

Optimizer User Guide

Comprehensive reference: models, options, continuation and integration

Patodia Infotech Private Limited

16 October 2026

Contents

1. Getting started and choosing a service	4
2. Write mathematical input: variables, constants and functions	10
3. Time limits, precision and interpreting results	17
4. General equations, including inclusive conditions	19
5. Optimizer as a sophisticated calculator	23
6. General linear optimization and mixed-integer models	32
7. General nonlinear and mixed-integer optimization	36
8. Numerical global optimization and objective gaps	41
9. Supplied binary exact-cover and partition patterns	45
10. City tours: traveling salesman problems	48
11. Capacitated vehicle routing	52
12. Vehicle routing with time windows	55
13. Cutting stock: one stock length	58
14. Cutting stock: multiple stock types	61
15. One-dimensional bin packing	63
16. Rectangular packing and cutting	65
17. Production and job scheduling	68
18. Staff rostering	71
19. School timetabling	74
20. Worker-job assignment	78
21. Transportation and shipment allocation	80
22. Minimum-cost network flow	82
23. Resource placement and rebalancing	84
24. Import existing LP, MPS and NL files	86
25. AMPL and GAMS modeling-language workflows	91
26. Python modeling interfaces and imported files	93
27. Multiple seeds and reproducible experiments	96
28. DLL integration: C, C++, Python, Java and C#	98
29. Using the PHP front end	105

30. Troubleshooting and model checks	107
31. Complete command-line option reference	109
32. Weighted blending and Pareto sampling	116
33. Priority and lexicographic linear objectives	120
34. Deterministic bounded work, checkpoints and continuation	122
35. Backend selection, diagnostics and additional numerical controls	126
36. Typed C core API, progress and cancellation	129
37. New service controls, compatibility and reproducible evaluation	133
38. Updated option index, migration and troubleshooting	136

Edition: 16 October 2026 - comprehensive options update

Equations, scientific calculations, optimization, operational models and application integration.

This guide explains how to construct a problem, choose a service, set its options, interpret its result and call Optimizer from an application. It includes the supported LP/MPS/NL imports, AMPL/GAMS runtime workflows, updated nonlinear/global controls and a scientific-calculator chapter. Every model chapter retains plain-English record explanations, so that you can replace example data with real inputs without guessing what each field means.

Earlier companion releases supply model files, calculator worksheets and language callers. The current source release supplies modular build scripts, public DLL/core headers, matching adapters and tests. This documentation delivery contains the Markdown and PDF manual, not a new companion source archive or external modeling runtime. Use matching source/binaries for the documented options.

A verified feasible result satisfies the represented requirements at the reported tolerance. A heuristic candidate is not automatically a proven optimum. Read the stopping reason and any objective bound/gap before drawing an optimality conclusion. Numerical precision, feasibility and global optimality are different properties.

Long commands may wrap visually in the PDF. Enter each shell command on one line; Markdown and companion files preserve complete source. The PDF contents and bookmarks navigate chapters, and the Markdown headings make the same material searchable.

What changed in this edition

This edition retains the existing 31 model and integration chapters and adds detailed Chapters 32-38. It documents weighted blending, two-objective Pareto sampling, affine lexicographic priorities, scalar NLP/MINLP checkpoint continuation, sparse nonlinear controls, MILP memory/branching/portfolio controls, affine IIS/feasibility-relaxation diagnostics, prebuilt backend selection and the typed C/Python core API. The complete option reference now cross-references those detailed controls.

Practical positioning: seek verified useful decisions within a controllable computational budget, inspect diagnostics and continue supported saved MINLP searches when useful. Heuristic search does not automatically establish global optimality or root completeness. Feasibility, objective quality and proof status remain separate. Time limits are cooperative, not hard real-time deadlines.

Important current limitation: cold-start work/checkpoint search is not a quality replacement for ordinary timed search on every model. The user-reported Specialized heat-exchanger comparison found a worse objective after 160 work units than an ordinary 30-second run. Chapters 34 and 37 document this regression and recommend the successful timed path for that case. No optimizer source correction is represented as completed in this documentation update.

Build migration: the engine is modular. The old single-file g++ commands are superseded by Chapter 1's optimizer-only build.py/Windows wrapper commands. Optional HiGHS is reused as a prebuilt dependency; this manual contains no commands to compile HiGHS.

Reading route: use Chapters 1-3 for setup, syntax and status; the relevant model chapter for its grammar; Chapter 31 plus 38 for option lookup; Chapters 32-36 for new mathematical/integration workflows. Single-objective timed, checkpoint work, Pareto and lexicographic modes have different contracts.

1. Getting started and choosing a service

How to use this manual

The executable has several services. They share a program name but do not share every input grammar or every option. Mathematical models use expressions and declarations. Operational services use named records. Tour and capacity-routing services accept standard instance files.

Commands and paths Long commands may wrap visually in the PDF; enter each command on one line. Complete runnable programs are included in the companion files.

Commands assume that optimizer.exe is in the current directory and that the companion models folder is beside it. In PowerShell, use .optimizer.exe if the executable is not on PATH. In Command Prompt, optimizer.exe is sufficient. Replace paths with your actual locations. Quote paths containing spaces.

Examples and expected answers Small examples have stated answers where these can be checked directly. Heuristic services may return different valid plans with the same or a different objective. The companion validator checks the supported sample models; a single successful example is not a performance guarantee for a much larger model.

Precision and language examples The guide documents the GCC/libquadmath binary128 build. A portable double-only build is also supported. Python examples use the supplied OptimizerDLL class, not a .pyd module. Java uses JNA. C and C++ use the supplied C header.

Reading conventions Uppercase words in service formats are record keywords. Names such as Alice, M1 and math1 are identifiers chosen by you. A bracketed field in a syntax description is optional; do not type the brackets unless the actual grammar requires them. In a bound declaration, the square brackets are required.

Start here For a first test, use the five-minute quick start. For service selection, use the service selection table. For all command-line settings, use the complete command-line option reference. For direct library calls, begin with the DLL integration chapter.

Model-building method For each problem, first identify the unknown choices, fixed data, hard restrictions and preferred objective. Then use the model-specific record explanations and annotated worked file. Replace actual values, not just names; keep units and the meaning of every field consistent. A solver can only enforce requirements you represent.

Five-minute quick start

Create a plain UTF-8 file called roots.txt containing:

```
x^2 = 4
```

Run:

```
optimizer.exe --file roots.txt --seconds 5
```

The real roots are $x=-2$ and $x=2$. The residual measures how closely the original equation is satisfied. The command line has no interactive prompts.

Next create production.txt:

```
int x,y
maximize 5*x+4*y
2*x+y<=8
x+2*y<=8
bound x=[0,8]
```

```
bound y=[0,8]
```

Run:

```
optimizer.exe --linear --file production.txt --seconds 10
```

The optimum is $x=3$, $y=2$, objective=23. This is a mixed-integer linear model. Integer declarations affect the allowed values; the objective specifies what is being improved.

Check three things before using a result - Was a complete feasible candidate returned? - Does the reported objective mean what your model intends? - Is optimality established, or is this a best feasible result from a limited search?

Run optimizer.exe with no arguments to display help. Run optimizer.exe --solver-help for equations or --services-help for operational records. Use --file explicitly whenever you also pass settings.

Choose the service that matches your problem

Need	Select	Input
Find real/integer equation solutions	Default equation mode	Expressions and declarations
Minimize/maximize an affine model	--linear, or automatic selection	Native text, LP or MPS
General nonlinear objective/constraints	Default optimization mode	Expressions and declarations
Select supplied exact-one patterns	--crew	Binary equations and cost
Visit all cities and return	--tsp	Symmetric TSPLIB .tsp
Deliver with identical capacities	--cvrp	VRPLIB .vrp
Deliver within time windows	--vrptw	TYPE VRPTW records
Cut one stock length	--cutting-stock	TYPE: CUTTING_STOCK
Cut multiple stock types	--cutting-stock-multi	TYPE CUTTING_STOCK_MULTI
Pack bins / rectangular sheets	--bin-packing / --packing2d	Named item records
Schedule jobs and machines	--schedule	TYPE SCHEDULING
Assign staff to shifts	--roster	TYPE ROSTERING
Place lessons into periods	--timetable	TYPE TIMETABLE
Match workers to jobs	--assignment	TYPE ASSIGNMENT
Allocate shipments	--transport	TYPE TRANSPORTATION
Route bounded flow at minimum cost	--mincost-flow	TYPE MIN_COST_FLOW

Do not force a routing problem into thousands of generic binary variables when the dedicated service covers the actual requirements. Conversely, do not use a dedicated service if it cannot represent a required rule. Unsupported side constraints are not automatically inferred.

For example, CVRP does not model delivery windows or heterogeneous vehicles; use VRPTW. Crew mode selects already legal patterns; roster mode assigns individual employees with availability, skills and workload constraints.

For multi-resource objects and bins, use --placement --file PATH with TYPE RESOURCE_PLACEMENT. The placement chapters explain resource columns, eligibility, migration charges and separation groups.

Install the executable and DLL

For Windows, place optimizer.exe in a working directory. For library use, also keep optimizer.dll and the supplied language adapter files in known locations. The calling process and DLL must have matching architecture: an x64 DLL requires x64 Python, Java or another x64 application.

Runtime dependencies A dynamically linked MinGW build may require libquadmath and MinGW runtime DLLs. Use the dependencies generated by the same toolchain as the build. Put them beside the executable/DLL, or use the documented runtime-directory setting in the Python adapter. Do not mix unrelated runtime DLL versions.

Basic checks

```
optimizer.exe --help
optimizer.exe --solver-help
optimizer.exe --lp-help
```

For Python, place optimizer_ctypes.py beside your script, then run:

```
from optimizer_ctypes import OptimizerDLL
engine = OptimizerDLL(r"C:\optimizer\optimizer.dll")
print("Binary128 available:", engine.has_binary128)
print(engine.solve("x^2=4", seconds=5)["values"])
```

The guide's runnable Python programs take the DLL path as their first argument. Paths displayed here are examples; change them to your actual directory.

Other operating systems A Windows DLL is not a Linux shared object. Build the source with a suitable native Linux toolchain for a .so. Match platform, architecture and runtime dependencies. The C interface is portable; compiled binaries are platform-specific.

No Python-version-specific extension is required for the ctypes adapter. The supplied Python code targets Python 3.9 or later, subject to the matching machine architecture and native dependencies.

Build the command-line program and DLL

The current engine is modular: optimizer.cpp is a thin entry point, and the implementation is in src/. The old command that compiles optimizer.cpp alone no longer builds the complete product. Use Python 3.9+ and a compatible C++17 GCC/MinGW compiler. Extended builds require quadmath. These commands build OUR optimizer only, including its executable and shared library; they do not configure or compile HiGHS.

Windows native build, from the extracted source directory

```
python build.py --jobs 4 --opt-level 1
```

This produces optimizer.exe, optimizer.dll and the MinGW import library. The native wrapper is also available:

```
build_windows_native.cmd
```

The wrapper defaults to the build script's optimization level; pass --opt-level1 if you want that tested level. Supported build.py optimization levels are 0,1,2,3 (default 3). --double-only omits quadmath/binary128, --clean forces a rebuild, --debug uses -O0, --cxxPATH selects a compiler, and --jobsN controls parallel compilation. It is a build tool, not a solver command.

Reuse an already built HiGHS SDK

```
set "OPTIMIZER_HIGHS_ROOT=C:\PannaLaIPatodia\highs-install"
build_windows.cmd --opt-level 1
```

The SDK needs matching headers and prebuilt libraries. This script builds only the optimizer and links that existing dependency. It does not rebuild HiGHS. For custom layouts:

```
python build.py --jobs 4 --opt-level 1 --with-highs --highs-include
"C:\Dependencies\highs\include\highs" --highs-library "C:\Dependencies\highs\lib\libhighs.a"
```

Supply additional existing archives/includes only when your SDK needs them. Runtime DLL dependencies must be discoverable by the calling process. Do not mix architectures or incompatible toolchains. --lp-enginehighs is unavailable when that backend was not linked.

Linux native build

```
python3 build.py --jobs 4 --opt-level 1
```

This produces optimizer and liboptimizer.so. Match the target's glibc/compiler/runtime baseline; a build on a newer Linux host is not automatically CentOS 8 compatible. A Windows DLL cannot be loaded as a Linux shared object. ctypes callers do not require a Python-version-specific extension, but still require the correct architecture and runtime libraries.

Incremental builds and checkpoints

The build caches compatible objects and reuses core objects for CLI/DLL. A no-change build need not compile the whole engine. Rebuilding code can invalidate existing work checkpoints because their identity includes compilation details; keep the original executable for continuation. For ordinary application use, installation of a matching supplied binary does not require recompiling every run.

After building

```
optimizer.exe --help
optimizer.exe --solver-self-test
optimizer.exe --lp-self-test
optimizer.exe --milp-self-test
python tests\dll_smoke_test.py --library optimizer.dll
```

The Windows wrappers compile; they do not implicitly execute every test mentioned in historical release notes. Run relevant current test scripts explicitly. On Linux replace optimizer.exe/optimizer.dll with ./optimizer/liboptimizer.so as appropriate. Windows/MinGW commands are provided for local verification; the documentation smoke checks used the available Linux build.

Files, encoding and paths

Use plain UTF-8 text. A UTF-8 byte-order mark is accepted by mathematical input. Avoid a Word document renamed to .txt: formatted document bytes are not valid model text.

Single-argument behavior

```
optimizer.exe equations.txt
optimizer.exe "x^2=4"
```

One existing positional argument is treated as a file. If it does not exist, it is treated as equation text. To make a missing file fail clearly as a missing file, use --file. With several arguments or options, always prefer explicit --file or --eq.

Reliable file command

```
optimizer.exe --file "C:\Models\monthly plan.txt" --seconds 30
```

Mathematical --file - reads standard input. Many named services can also read text through their supported --eq interface. TSP, CVRP and homogeneous cutting wrappers are file-oriented; use actual instance files for these services.

Model file extensions Native equations/optimization and record services can use .txt. A tour usually uses .tsp, capacity routing .vrp, and homogeneous cutting .cstock. LP and MPS imports use explicit --lp-file and --mps-file; do not rely on a filename extension alone to select an importer.

Application paths A file argument passed to the DLL is resolved in the calling application's working directory. Use absolute paths in services, notebooks, scheduled jobs and web applications. A successful CLI call from one directory does not imply that a background process has the same directory.

Keep the input file, chosen options and resulting plan together. They provide a reproducible record of the represented problem and the accepted solution.

Command lines, quoting and repeated statements

Shells interpret characters such as `<`, `>`, `*`, spaces and parentheses. Quote every inline model statement, especially inequalities. The solver needs each equation as a complete argument or as part of one `--eq` text argument.

```
optimizer.exe "x+y=6" "x*y=8"
optimizer.exe --eq "x+y=6; x*y=8" --seconds 5
optimizer.exe --minimize "(x-3)^2" "x>=0" "x<=5"
```

These are argument lists, not mathematical strings joined without separators. A newline-separated file avoids most shell-escaping problems.

Capturing output in Command Prompt

```
optimizer.exe --file model.txt > result.txt 2> progress.txt
```

Without `--progress`, reports show settings, results and conclusions. `--progress` adds discoveries and verified feasible-result updates. Error messages go to `stderr`. The DLL captures output separately rather than requiring shell redirection.

PowerShell

```
.\optimizer.exe --file .\models\lp_production.txt --seconds 10
$LASTEXITCODE
```

Use PowerShell's own quoting rules for paths and literal input. Long build commands can be placed on one line in a `.bat` file; do not paste PowerShell backtick continuation into Command Prompt.

Calling from an application Pass a list of strings. Do not join everything into one shell command and pass that as a single DLL argument. For Python:

```
result = engine.run_capture([
    "--file", "models/lp_production.txt", "--seconds", "10"
])
```

The DLL argument list excludes the executable name. No shell expansion is performed by `optimizer_run`. A string containing `--seed` inside an equation argument remains part of that input, not a separate setting.

The standard command form for every model

Use this form for dedicated services:

```
optimizer.exe --MODE --file PATH [options]
```

Replace `--MODE` with a real service flag and `PATH` with your model file. `MODE` is a placeholder, not a flag to type. Options follow the path, for example `--seconds 20`. Quote a path containing spaces.

```
optimizer.exe --packing2d --file "C:\Models\panels.txt" --seconds 20
optimizer.exe --cutting-stock --file cutting_stock_wiki.cstock
optimizer.exe --cutting-stock-multi --file stock.txt
optimizer.exe --crew --file pairings.txt --seconds 30
```

Both `--cutting-stock` and `--file` may be used together. This is the recommended consistent form, including for services that also accept a legacy positional filename.

Model	Service flag
General equations, including conditions	No mode flag: use <code>--file PATH</code>
General linear optimization	<code>--linear</code>
General nonlinear optimization	<code>--sqp</code> , or automatic selection from the objective
Supplied binary covering/partition patterns	<code>--crew</code>
City tour; capacity vehicle routing	<code>--tsp</code> ; <code>--cvrp</code>
Routing with time windows	<code>--vrptw</code>

Model	Service flag
Cutting stock: one or multiple stock types	--cutting-stock; --cutting-stock-multi
Assignment; transportation; network flow	--assignment; --transport; --mincost-flow
Scheduling; staff rostering; school timetable	--schedule; --roster; --timetable
One-dimensional bins; rectangular sheets	--bin-packing; --packing2d
Multi-resource object placement	--placement

LP/MPS/NL imports also use one --file PATH, with --input-format lp, --mps or --nl when an explicit selector is needed. The older --lp-file and --mps-file shorthand forms remain supported.

Supported file families and the right entry point

Your file	Standard submission	What it represents
Native .txt equations	optimizer.exe --file model.txt	Equalities and optional $\leq$ / $\geq$ conditions; no objective means root search.
Native .txt optimization	optimizer.exe --linear --file model.txt or --sqp --file model.txt	Expressions, declarations, one objective and constraints.
Dedicated service model	optimizer.exe --MODE --file model.txt	Service-specific records, explained in its chapter.
.tsp / .vrp / .cstock	--tsp / --cvrp / --cutting-stock --file PATH	Supported tour, capacity-routing or single-stock instances.
.lp	--input-format lp --file model.lp	Supported affine LP/MILP exchange file.
.mps	--mps --file model.mps	Supported linear/integer MPS exchange file.
.nl	--nl --file model.nl	Supported generated algebraic model, ASCII or binary.
AMPL .mod + .dat	solve_ampl_file in Python, or select Optimizer inside AMPL	The installed AMPL runtime creates the generated model.
GAMS .gms	solve_gams_file in Python	The installed GAMS runtime and CONVERT export the generated model.

MODE is a placeholder for a real selector, for example --packing2d. It is not an option to type literally. Use exactly one model input and one compatible service/format selector. --cutting-stock --file PATH is valid; the same standard form applies to the other dedicated services.

LP/MPS/NL extensions are detected case-insensitively with --file. If an LP exchange file is named model.txt, supply --input-format lp explicitly. An AMPL or GAMS source file is not a native equation file: do not pass it directly to --file and expect the language to be compiled. The import chapters explain each supported subset and workflow.

2. Write mathematical input: variables, constants and functions

Statements, comments and identifiers

Mathematical input accepts one statement per line or semicolon-separated statements. Parentheses and bound brackets may span lines. Blank lines are ignored. An equation with no equal sign means that its expression equals zero.

```
/* Parameters for today's run */
constant capacity = 40
int units
bound units = [0, capacity]
2units + spare = capacity; // balance equation
```

Block comments continue from `/*` to `*/` and may contain newlines. A `//` comment ends at the newline. Close every block comment; an unfinished comment is an input error. Do not put the next statement after `//` on the same line. Legacy whole-line `#` comments also exist, but use C/C++ comments for new mathematical files.

Variable names begin with a letter or underscore, followed by letters, digits or underscores. They are case-sensitive: `R1` and `r1` are different. `xy` is one variable name; it is not `x` times `y`. Function names and built-in constants should not be used as declaration names.

Structured service identifiers Named operational records use single-word identifiers, including letters, digits, underscore and hyphen where supported. Use `Teacher_John` rather than an identifier containing spaces. Keep a separate mapping to a human-readable name in your application.

Fixed-decimal operational fields accept at most six decimal places and absolute values up to 1,000,000. VRPTW fields use double.

Service records are not general equations. A record such as ITEM panel 3 5 supplies numeric fields in a prescribed order. Do not insert mathematical constant declarations, Python code or arbitrary expressions into a service file unless its grammar explicitly supports them.

Unknown fields and unsupported options are errors. That is useful: a misspelled operational rule should not silently disappear from the model.

Variable types and declarations

Declaration	Allowed values
No declaration	Real-valued unknown
int a,b / integer a,b	Integers, including zero and either sign
binary selected	0 or 1
positive int trucks / pos int trucks	Integers ≥ 1
negative int returned / neg int returned	Integers ≤ -1
nonnegative int stock / nonneg int stock	Integers ≥ 0
nonpositive int delta / nonpos int delta	Integers ≤ 0

Signed forms also accept integer instead of int. Comma-separated lists are supported. These declarations work in both equation mode and general optimization. CLI `--int a,b` and `--integer a,b` declare unrestricted integer names; signed/binary declarations are written in the model text.

Domain intersections Repeated declarations intersect their domains. binary b together with positive int b allows only b=1. Incompatible declarations are rejected. Bound overrides do not remove declaration restrictions. binary b plus --bound b -5 5 still allows only 0 or 1.

Useful distinctions Positive excludes zero; nonnegative includes it. Use nonnegative int production for a quantity that may be zero. A declaration supplies a sign/domain, not an automatic finite upper limit. Provide a meaningful upper bound when available.

Integer arithmetic in expressions still uses real division. If male=280 and female=200, male/female is 1.4, not 1. Restricting variables does not change operator semantics.

The general root/integer search documents exact integer representation up to magnitude $2^{53}-1$. Do not model arbitrary-size exact integer arithmetic as floating-point equations. Dedicated packing/cutting services have their own count and size limits.

Constants, pi and Euler's number

Declare fixed parameters with constant or const. They are excluded from the unknown variables and must be defined once by expressions that depend only on other constants and built-ins.

```
constant a1=40, a2=sqrt(a1), R1=2*a2;
constant price=3.75;
x=R1
```

Definitions may appear in any order. Forward references among declared constants are permitted. A missing definition, repeated definition, dependency cycle or invalid real-valued expression is an error. Multiple inline initializers are supported: constant a=3, b=max(2,4), c=sqrt(a+b); Commas inside function arguments do not separate declarations. Declared names may also be initialized later.

Compact example

```
constant a=3, b=2;
c=sqrt(a+b)
```

Only c is unknown. Its value is approximately 2.23606797749979. Separate declaration and assignment statements remain supported.

Built-in constants - pi and #pi both mean the circle constant. - #e means Euler's number. - Bare e is an ordinary variable.

```
e = 2
x = exp(e)
y = #e^2
```

Here e is an unknown constrained to 2, while #e is fixed. exp(2) and #e^2 evaluate the same mathematical quantity. Scientific notation such as 2e3 means 2000; use 2*e or 2 e to multiply by the variable e.

Parameters versus equations Writing R1=10 without declaring R1 constant makes it an equation involving an unknown R1. It may still be solvable, but it increases the model's variable/equation count. Use constant declarations when the value is a known parameter rather than a decision.

Constants may be used in objectives, constraints and bound endpoints. Dedicated service records normally require numeric literals instead; resolve those parameters in your spreadsheet or Python code before generating records.

Bounds and one-sided limits

Mathematical file bounds use inclusive finite endpoints:

```
constant capacity=40
bound x=[0,capacity]
bound y=[-sqrt(capacity),sqrt(capacity)]
bound fixed=[2,2]
```

Endpoints may depend only on constants and built-ins. Equal endpoints fix a variable. Unknown bound names, duplicate file bounds, reversed endpoints and nonfinite values are errors. Bounds may appear before constant definitions.

Override precedence Per-variable CLI `--bound NAME LOW HIGH` overrides CLI `--bounds LOW HIGH`. Global CLI `--bounds` overrides file bounds. The effective interval still intersects any binary or signed integer declaration. Overrides are replacements of lower-priority bounds, not merely extra intersections with the original file bound.

```
optimizer.exe --file model.txt --bounds -20 20 --bound x 0 5
```

Do not apply global bounds casually: they may change every decision's physical domain. For instance, they can replace a positive file lower bound on a continuous variable. Declaration restrictions remain active.

One-sided constraints Use $x \geq 0$ or $x \leq \text{capacity}$ in optimization, especially LP/MILP. Write two statements for a two-sided inequality or use a bound. The form $0 \leq x \leq 10$ is not a native chained constraint.

Unspecified bounds are mode-dependent In native LP/MILP, variables are free unless constrained; no artificial box is added. Equation and general nonlinear searches use finite automatic ranges. An automatic range is a search aid, not evidence of a physical limit or proof of unboundedness. Supply known engineering limits whenever possible.

Integer intervals must contain at least one allowed whole value. `int x` with bound $x = [0.2, 0.8]$ has no allowed value and is rejected.

Operators and precedence

Operator	Meaning	Example
+ and -	Addition, subtraction; unary signs	$x-y$; $-x$
*	Multiplication	$3*x$
/	Real division	male/female
^ or **	Exponentiation	x^2 ; $x**2$
()	Grouping	$(x+y)/2$
=	Equality statement	$x+y=10$
<= and >=	Inclusive optimization inequality	$x^2 \leq 25$

Powers associate to the right: $2^3 2$ means $2^{(3 2)} = 512$. A unary minus precedes the powered expression in meaning: $-x^2$ means $-(x^2)$. Use $(-x)^2$ when the base itself should be negative.

Implicit multiplication Supported examples include $2x$, $3y^2$, $2(x+1)$, $x(y+1)$, $(x+1)(y-1)$, $2\sin(x)$, 2π and $x y$. Use explicit `*` for clarity in generated models. Adjacent letters form a name: xy is not multiplication.

Multiplication and division have equal precedence and are evaluated left to right. $x/2y$ means $(x/2)y$. Use $x/(2y)$ for division by the product. This is a common source of incorrect models even when the expression parses successfully.

Numbers Decimals and scientific notation are accepted: `0.12`, `.5`, `1e-9`, `2e+3`. In extended models, write important coefficient digits as text literals; converting them to Python float first may already lose precision.

There are no general Boolean operators, programming loops, ternary expressions, strict `<` or `>` constraints, or arbitrary user-defined functions in the native expression grammar. Generate repeated statements in Python and use supported mathematical constraints to represent relationships.

Arithmetic, roots and logarithm functions

Function	Meaning and real domain
abs(x)	Absolute value; all real x
sqrt(x)	Nonnegative square root; $x \geq 0$
cbrt(x)	Real cube root; supports negative x
pow(a,b)	a raised to b; requires a real result
exp(x)	Exponential, base Euler's number
log(x), ln(x)	Natural logarithm; $x > 0$
log10(x)	Base-10 logarithm; $x > 0$
log1p(x)	$\log(1+x)$; $x > -1$
expm1(x)	$\exp(x)-1$

All functions require parentheses. A two-argument function uses a comma. Functions are available in mathematical equations, objectives, general constraints and constant expressions, not as arbitrary record-field expressions in dedicated services.

Stable forms near zero Use `log1p(x)` instead of `log(1+x)` when x is very small, and `expm1(x)` instead of `exp(x)-1` when cancellation matters. They express the same mathematical quantities with better small-argument numerical behavior.

Negative bases `pow(-2,3)` has a real result; a negative base with a noninteger exponent usually does not. Use `cbrt(x)` for a real cube root of a potentially negative x. Writing $x^{(1/3)}$ is not a reliable substitute for `cbrt(x)` on negative real inputs.

Example

```
constant k=log(2)
bound x=[0,10]
exp(-k*x)=0.25
```

The solution is $x=2$. Use meaningful domains to keep search points away from undefined logarithms, negative square roots and overflow. A function returning a nonfinite value does not become an allowed complex result; the engine searches real solutions.

Trigonometric and hyperbolic functions

Function	Meaning
sin(x), cos(x), tan(x)	Trigonometric functions; radians
asin(x), acos(x)	Inverse functions, real input in $[-1,1]$
atan(x)	Principal inverse tangent
atan2(y,x)	Quadrant-aware angle from coordinates
sinh(x), cosh(x), tanh(x)	Hyperbolic functions

Arguments and returned angles are in radians. Convert degrees using `angle_degrees#pi/180`. Convert a returned angle to degrees with `angle_radians180/#pi`.

```
constant degrees=30
s=sin(degrees*#pi/180)
```

This equation fixes s to 0.5. In an equation-only model s is the unknown; degrees is constant.

Periodicity $\sin(x)=0$ has infinitely many real solutions. The solver can report a finite sample, usually within automatic or explicit ranges. To find roots within an operational interval, add bound $x=[-2\#pi,2\#pi]$. Increasing the range creates more possible roots; decreasing the tolerance does not by itself explore farther.

Singularities and branch choices $\tan(x)$ is undefined at odd multiples of $\pi/2$. `atan2(y,x)` is sensitive near the origin and has an angular branch boundary. Principal `asin/acos/atan` values are functions, not instructions to enumerate all inverse branches of an equation.

Practical modeling For rotating equipment, coordinates and AC power models, document whether every angle is radians or degrees before creating the file. A factor-of- $180/\pi$ mistake often produces a feasible answer to the wrong model. Large `sinh/cosh` arguments can overflow;

realistic bounds help avoid meaningless search regions.

Minimum, maximum, hypot and supported functions

Function	Meaning
<code>min(a,b)</code>	Smaller of two expression values
<code>max(a,b)</code>	Larger of two expression values
<code>hypot(a,b)</code>	Euclidean magnitude $\sqrt{a^2+b^2}$
<code>atan2(a,b)</code>	<code>atan2</code> first argument, second argument
<code>pow(a,b)</code>	Exponentiation

`min` and `max` have exactly two arguments. To combine three values, nest them: `max(a,max(b,c))`. These functions evaluate an expression; they do not set an optimization objective.

Objective versus pointwise function

```
minimize x^2+y^2
```

asks for the smallest objective value. In contrast:

```
max(x,0)=x
```

is an equality encoding $x \geq 0$. Prefer the direct inequality $x \geq 0$ for optimization. A bare `max(z,0.10)` is interpreted as `max(z,0.10)=0`, which cannot express $z \geq 0.10$.

Complete callable function list `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `log`, `ln`, `log10`, `exp`, `sqrt`, `abs`, `cbrt`, `log1p`, `expm1`, `pow`, `atan2`, `min`, `max` and `hypot`.

Functions not on this list, such as `floor`, `ceil`, `round`, `sign`, `sum`, `if` and user-defined function calls, are not part of the current native expression language. Write explicit sums or generate them from Python. Integer declarations often provide the correct modeling alternative to rounding decision variables inside an expression.

Smoothness `abs`, `min` and `max` can have kinks. They are allowed, but difficult nonsmooth models may need a reformulation or a larger search budget. Their availability is not a guarantee that every discontinuous model will converge.

Domains, units and safe model construction

The solver works with real-valued expressions. A trial with `log(x)` at $x \leq 0$, `sqrt(x)` at $x < 0$, division by zero or an overflowing exponential is invalid; it is not a complex-number calculation.

Make valid domains explicit

```
bound pressure=[0.1,100]
bound ratio=[0.01,10]
log(pressure)+sqrt(ratio)=target
```

If `target` is known, declare and define it as a constant. If it is an unknown, it remains a decision variable. For optimization, $x \geq 0$ allows zero, which is not enough for `log(x)`; use a meaningful strictly positive finite lower bound instead.

Choose consistent units Use one unit per physical quantity throughout each model. In routing, coordinates determine distance units; speed must agree with distance and time. In cutting, stock lengths, kerf and trim use the same unit. In rostering, hours and shift timestamps must share a consistent time scale.

Absolute residuals Feasibility tolerance is in the original constraint units. A balance measured in rupees and another measured in millions of rupees have different numerical scales. Formulate well-scaled equations using declared constants, and choose a tolerance meaningful for the original measurement precision.

Structural checks before search Confirm that every identifier exists, every required quantity

is present, bounds do not contradict declarations, total supply balances total demand where required, and all mandatory assignments have at least one eligible option.

The engine checks returned candidates against the represented input. It cannot verify requirements that were never modeled: labor rules, road restrictions, school policy, taxes and engineering safety factors must be supplied explicitly in a supported form.

Objectives, conditions and mode selection

For a traditional scalar model, write one objective as minimize EXPRESSION or maximize EXPRESSION. Named blend/Pareto objectives and affine priorities use the separate grammars in Chapters 32-33. The objective tells the program what to improve. A condition tells it which values are acceptable. These are different decisions.

Equations with conditions, without an objective Default equation mode accepts $LHS \geq RHS$ and $LHS \leq RHS$ alongside equations. It searches for distinct solutions of the equations that also satisfy the conditions. Adding $x \geq 0$ does not, by itself, request minimization. Strict $<$ and $>$ are unsupported.

```
x^2=4
x>=0
```

The acceptable root is $x=2$. Without the condition, both -2 and 2 are roots. A condition-only model describes a feasible region; returned points do not enumerate that whole region.

Explicit feasibility services `--linear` requests linear feasibility/optimization. `--sqp` requests nonlinear feasibility/optimization. With no objective these explicit services use objective zero and seek a feasible candidate, rather than reporting multiple equation roots. `--crew` retains its supplied-pattern semantics.

With an objective An affine objective and affine constraints normally select LP/MILP. A nonlinear expression requires nonlinear optimization. Integer declarations restrict permitted variable values. Do not combine `--linear` and `--sqp`.

Use inequalities directly: $0.12w_1 + 0.08w_2 \geq 0.10$. A bare `max()` expression does not mean “enforce this lower limit”; it is interpreted as an equation to zero. Dedicated operational services express their rules through records, not arbitrary appended algebraic constraints.

Primal starting-point hints for mathematical optimization

The current native optimizer grammar accepts numeric coordinate hints:

```
minimize (x-3)^2
bound x=[0,10]
start x = 2.5
```

`startNAME=NUMBER` requires spaces around the equals sign and a numeric literal as shown. It is guidance, not a fixed value or an extra constraint. Names must already belong to the mathematical model, and duplicate/nonfinite starts are rejected. Every accepted candidate must still satisfy the original requirements. These model hints, imported NL primal values, typed named starts and the complete work-mode incumbent file are different interfaces. A hint is not a saved random/search state or an optimality certificate. It does not add this syntax to operational record grammars or guarantee a feasible candidate. See Chapters 24, 34 and 36 for the applicable entry paths.

Precision and tolerance

Normal mathematical mode uses double precision, with default final/feasibility tolerance $1e-9$. A quad-enabled build supports `--extend` with native binary128 arithmetic and approximately 33 decimal significant digits. The default final/feasibility tolerance becomes $1e-24$ in applicable mathematical modes.

```
optimizer.exe --file roots_system.txt --extend --tol 1e-24 --seconds 30
```

Extended nonlinear/root mode still uses ordinary-precision exploration before higher-precision checking/refinement. Extended LP/MILP and network modes use the extended numerical path directly. The report shows Precision: standard or Precision: extended. optimizer_has_binary128() reports whether the DLL supports binary128.

Three distinct optimization tolerances - --tol or --ctol: original constraint feasibility. - --gtol: scaled first-order/stationarity test, nonlinear optimization only. - --ftol: relative objective stagnation, nonlinear optimization only.

Double defaults are 1e-9, 1e-6 and 1e-12 respectively. Extended nonlinear defaults are 1e-24, 1e-18 and 1e-28. Equation mode accepts --tol, not --ctol.

Tighter is not always more useful A smaller numerical tolerance is stricter. 1e-30 is stricter than 1e-24. It can reject otherwise useful candidates or consume more time refining them. Precision does not guarantee more roots, global optimality or unlimited accuracy. Conditioning and coefficient accuracy matter.

Dedicated services TSP/CVRP accept --extend but do not perform quad refinement. Operational services reject it; hard quantities use their documented decimal rules, while VRPTW uses double. Homogeneous cutting and dedicated network modes do support extended controls. Do not apply one precision recipe to every service.

3. Time limits, precision and interpreting results

Time, work budgets and reproducibility

--seconds sets an approximate wall-time search budget. Zero disables the clock limit; it does not disable independent work limits. Defaults depend on mode: 20 seconds for mathematical, crew, network and operational services; 30 seconds for TSP, CVRP and homogeneous cutting. Multi-seed defaults to a total 20 seconds unless overridden.

Parsing, preparation, final verification and an indivisible operation can make elapsed time slightly exceed the chosen limit. The program is synchronous: it returns when the call finishes. Do not assume --seconds is a hard operating-system kill deadline.

Work caps are separate Root/nonlinear modes have starts and iterations. MILP has nodes and pivots. Crew has moves and local repair limits. Operational services have starts and iterations. Increasing --seconds alone may not help if an independent work cap is reached first.

Change one setting at a time Keep a baseline file, time allowance and seed. Try a larger time allowance; then try a different seed or multi-seed strategy where supported. Compare feasible objective values and validation, not merely the number of diagnostic messages.

A fixed seed controls pseudorandom choices. Equal seeds under clock-limited runs can still produce different outcomes because different machines, loads and budgets permit different amounts of work. It is not a cross-machine bit-for-bit result guarantee.

Budget recommendations Use short sample runs to check parsing and feasibility. Use operationally meaningful limits for actual planning. For a difficult heuristic model, retain the best verified result across attempts. Avoid imposing an impractically tight tolerance when input data itself is measured only to a few decimal places.

Native multi-seed trials and the final phase share one total time allowance; see the relevant service option reference.

Timed search versus checkpoint work mode

--seconds0 removes the wall-time ceiling; independent effort caps still stop search. With a checkpoint/resume flag you must provide --work-limitN, and N means ADDITIONAL completed attempts per call. An easy attempt can complete very quickly. Checkpoint mode has a different schedule from the established timed solver and can give worse objective quality. Read Chapter 34 before selecting it; the Specialized cold-start regression remains unresolved in this source candidate.

Read and accept a result

Public reports Every invocation begins with "Optimizer by Patodia Infotech Private Limited. All rights reserved." Equation reports show Precision, Variables, Residual tolerance, the time allowance and relevant bounds warnings. Solution summaries give a count and elapsed time without processing counters.

Optimization and operational reports identify the model type when applicable. Result values, constraint violations, objective bounds/gaps, optimality qualifications and stopping reasons remain available. --progress reports discoveries and feasible-result improvements on stderr. Supported modes can also expose work-unit counters, native-search audits and polishing diagnostics; do not assume every service has identical report fields.

A typical general optimization result contains a feasible minimum/maximum heading, an objective, variable values, maximum constraint violation, convergence information and a stopping reason. These fields answer different questions.

Feasible means the candidate meets the represented constraints and domains within the documented checks. **First-order convergence achieved** is a local continuous check; it is not a global minimum/maximum certificate. For integer problems, it refers to the continuous subproblem, not proof that the integer choices are best.

Objective bound and gap For minimization, a valid lower bound is no greater than the optimum; the incumbent is an achievable upper value. For maximization the directions reverse. Absolute gap measures the difference. LP/MILP relative gap uses $\max(1, |\text{incumbent}|)$ as its denominator. A missing bound is not zero and is not evidence of zero gap.

Service plans Read routes, operation start/end times, staff assignments, lesson slots and cutting patterns as plans, not just scalar objectives. A smaller route cost is useful only if capacity and customer coverage are complete. A timetable is useful only if every lesson occurrence has been assigned without clashes.

Search limits A time-limited feasible result remains usable if it meets your operational acceptance criteria. Optimality may remain unproven. No candidate under a limit is not proof that the model has no feasible solution.

Precision Printed digits are not a guarantee that every digit is physically meaningful. Preserve high-precision text when transferring an extended result to another language. Python's ordinary float representation is not binary128; the supplied adapter also retains numeric text fields.

Exit codes and failure handling

There are two status layers for a DLL call: the C-ABI call status and the solver's exit code. `optimizer_run` returning 0 means the API call completed. Inspect `optimizer_exit_code(result)` to determine what the solver reported.

Mode	Exit 0	Exit 1	Other outcomes
Equations	Successful run; inspect solution count	Input/build error	Zero roots can still be a successful completed search
General nonlinear optimization	Verified feasible candidate	Input/build error	2: no accepted feasible candidate
LP/MILP and network	Verified feasible candidate	Input error	2: numerical infeasibility; 3: unbounded/unresolved/limited without candidate
Crew	Verified schedule	Input error	3: no verified schedule
TSP/CVRP	Verified route result	Input error	CVRP 2: no result or stated necessary infeasibility
Cutting / operational services	Verified plan	Input/build error	2: supported infeasibility conclusion; 3: limited without a plan

Always read the accompanying status/message. Exit 0 does not always mean an optimum, and in root mode does not mean at least one root exists. Exact interpretation is mode-specific.

Handling errors in Python `solve()` normally raises `ValueError` for solver exit 1. Use `raise_on_error=False` if you want to show the structured error in your application. An API or DLL-loading failure raises a different exception. Resource-limited absence of a solution is a result to handle, not necessarily an exception.

Do not display an empty plan as an optimal zero-cost plan. Keep feasible, optimality status and termination reason separate in the user interface. For scheduled batch jobs, save `stderr` alongside `stdout` and the model file.

4. General equations, including inclusive conditions

General equations: file format

A root model contains equations, optional constants, variable-domain declarations bounds and optional $\geq$ or $\leq$ conditions. There is no objective. Each expression must equal zero, either explicitly or by being a bare expression.

```
constant total=6
bound x=[-10,20]
bound y=[-10,20]
x+y=total
x*y=8
```

Both x and y are real unless declared integer. The equations can be listed in any order; the solver detects variable names. User-chosen names do not require x1, x2 numbering.

Invocation

```
optimizer.exe --file equations.txt --seconds 20
optimizer.exe --eq "x+y=6; x*y=8" --seconds 5
```

Equation mode searches for distinct approximate real solutions. --max-solutions limits how many are retained. --merge controls relative deduplication of real coordinates. Different integer assignments are not merged by this real tolerance.

What is returned Each Solution record contains one complete variable vector and max |residual| for the original equations. The final residual must meet --tol. With fewer independent equations than unknowns, infinitely many solutions may exist; reported points only sample that set.

Scope The program need not find every solution. Periodic functions may have infinitely many roots. A root outside the explored automatic ranges may be missed unless you supply suitable bounds or expand the ranges. Known physical bounds make results more relevant and the search more focused.

Do not multiply the original equations by factors vanishing at an already known root to exclude it. That keeps or introduces zero residuals at that root; it does not express exclusion of the known solution.

Equation modeling: each statement explained

Decide what is known and what must be found. Declare known values as constants; leave unknowns undeclared unless they need an integer domain. An equals sign is an equation, not a sequential assignment to an unknown. Equation order does not change the intended system.

Statement	Meaning in simple English
constant a1, a2, R	Declare three known parameters, not unknown variables. Define them in the following statements.
a1=40	Set known parameter a1 to 40.
a2=sqrt(a1)	Calculate known a2 as the positive square root of a1, about 6.32455532.
R=2*a2	Calculate known R as twice a2, about 12.64911064.
bound x=[0,20]	Unknown x may be any real value from 0 through 20 inclusive.
x=R	Require the unknown to equal the calculated parameter. The equation determines x.

Complete companion file

```
optimizer.exe --file models/roots_constants.txt --seconds 10
```

Equation example: polynomial roots

```
x^2=4
```

Run the supplied example:

```
optimizer.exe --file models/roots_polynomial.txt --seconds 5
```

Expected roots are $x=-2$ and $x=2$, both with zero mathematical residual. The order of reported roots is not a ranking of quality.

A higher-degree example

```
(x-1)*(x-2)*(x-3)*(x-4)*(x-5)=0
```

Its real roots are 1 through 5. Factorized input expresses the intended coefficients directly. Expanded high-degree polynomials can be poorly conditioned: rounded coefficient data may define a different polynomial from the intended exact one.

The release has a dedicated single-variable polynomial handling path up to degree 64 within the configured range. This helps suitable problems, but does not turn floating-point root reporting into an exact completeness certificate. Bounds, time and solution-count settings still apply.

Wilkinson-style testing For a product with integer roots 1 through 20, make the input explicit and set a range covering all 20 roots. Compare count and residual against the original expression. Do not infer missing-root impossibility from a short search.

Useful settings `--tol` changes final acceptance accuracy. `--max-solutions` must be large enough for the requested count. `--bound x 0 25` restricts x to the desired interval. `--extend` requests tighter polishing in a compatible build. More time is useful only if the current stopping reason is not another work or solution limit.

Equation example: three roots of a system

```
x*y^2-x=6*y
x+y=6
```

Run:

```
optimizer.exe --file models/roots_system.txt --seconds 10
```

The system has three real solutions: - $x=4$, $y=2$. - $x=4-\sqrt{7}$, $y=2+\sqrt{7}$, approximately 1.354248689 and 4.645751311. - $x=4+\sqrt{7}$, $y=2-\sqrt{7}$, approximately 6.645751311 and -0.645751311.

These values follow by substituting $x=6-y$ into the first equation. Every reported pair must satisfy both original equations. Numerical output can differ in the last digits while representing the same approximate root.

Try a bounded run

```
optimizer.exe --file models/roots_system.txt --bound x -10 20 --bound y -10 20 --seconds 10
```

CLI bounds name the variable explicitly, so there is no reliance on a detected variable ordering. Keeping named bounds with the problem also avoids stale bounds from an unrelated model.

Try extended checking

```
optimizer.exe --file models/roots_system.txt --extend --tol 1e-24 --seconds 20
```

Use a reasonable acceptance tolerance first. A much stricter tolerance may spend more time polishing each candidate and return fewer accepted roots within the same allowance. Retain the model text and settings when comparing runs.

Equation example: integer student counts

```
int male, female, students
bound male=[0,1000]
bound female=[1,1000]
bound students=[1,2000]
male+female=students
5*male+4*female=2200
male+female+male/female=481.4
```

Run:

```
optimizer.exe --file models/roots_integer.txt --seconds 10
```

The intended whole-number solution is female=200, male=280 and students=480. The second equation gives $5280 + 4200 = 2200$. The third gives $480 + 280/200 = 481.4$ because division is real.

Without int declarations, the real-valued model also has female=11, male=431.2 and students=442.2. That is a valid solution to different domains, but not a count of people. Declarations are therefore part of the problem, not a display preference.

The bounds exclude female=0, where the ratio is undefined, and supply finite meaningful limits for count search. Integers remain integers during final validation. A displayed near-integer real number is not a substitute for declaring the variable integer.

CLI equivalent declaration --int male,female,students can be supplied instead of the first file line. The signed domain forms must be written as model declarations; do not invent a --positive-int option.

Equation example: fixed parameters

```
constant a1, a2, R
a1=40
a2=sqrt(a1)
R=2*a2
bound x=[0,20]
x=R
```

Run:

```
optimizer.exe --file models/roots_constants.txt --seconds 5
```

Only x is an unknown. The constants evaluate to a1=40, a2=sqrt(40) and R=2*sqrt(40). Therefore x is approximately 12.649110640673518.

Why declare constants? Engineering models often contain known material coefficients, scale factors, prices and measured totals. Declaring them prevents the engine from treating their definitions as extra equations with extra unknowns. It also makes the distinction between supplied data and decisions clear to the reader.

Longer dependency chains

```
constant base=40
constant scale=sqrt(base)
constant R3=0.0025970/scale
constant R5=0.00001799/base
```

Initializers can share a declaration: constanta=3,b=2; Separate statements are also supported. A constant cannot depend on an unknown decision. If a coefficient is meant to vary during optimization, leave it as a variable with constraints instead.

Parameter studies Use Python to generate this text for several supplied values, call the DLL for each and store the results. The language's loop runs outside the expression grammar. Do not write a Python for statement into the equation file.

For extended precision, keep exact coefficient strings or Decimal values when generating input.

Formatting an already rounded float with extra digits does not recover the original intended decimal data.

Difficult equation searches: a practical workflow

Begin with a physically reasonable range and the default 1e-9 final tolerance. Check that the example parses and that at least one returned point satisfies every original equation. Then increase exploration or refine accuracy, depending on what you need.

If too few roots are found - Increase `--seconds` and, if necessary, `--starts` or `--iterations`. - Try additional seeds or native multi-seed search. - Expand relevant automatic ranges using `--radius`, `--stages` and `--growth`, or supply known bounds. - Check `--max-solutions` and `--merge`; the count may be limited or nearby real roots may be deduplicated.

If refinement is difficult Try `--extend` with a achievable tolerance, such as 1e-24, before attempting 1e-30. A larger numerical tolerance is less strict; it can accept more approximate roots. A smaller one requests stronger residual accuracy. These changes do not guarantee that all difficult roots will be discovered.

If the model has invalid domains Supply positive lower bounds for logarithm arguments, keep denominators away from zero where mathematically justified, and review fractional powers of negative quantities. Do not change the mathematical domain merely to hide a modeling error.

Controlled comparisons Keep the input text unchanged and vary one setting per experiment. Compare original residuals and complete solution vectors. For a singular or nearly singular system, a tiny residual does not necessarily imply a correspondingly tiny error in the variable values.

`--single-stage`, `--no-deflation`, `--no-inference` and `--no-slices` are optional comparison controls. Defaults should normally remain enabled. They are not required to write a valid equation model, and disabling helpful search behavior can reduce the number of roots found.

5. Optimizer as a sophisticated calculator

Run the numbered worksheets from the companion calculator folder, or prefix their filenames with calculator from the companion root. Use `--file` explicitly when also providing settings.

Named results turn equations into a calculator

Use Optimizer for reusable scientific worksheets, chained calculations, unit conversions and solving a formula backwards. Name the quantities you want displayed. The companion calculator folder contains the worksheets; the current executable was used to check the examples for this edition.

Start with named answers

```
optimizer.exe --eq "constant a=3,b=7; c=a+b; d=a/b"
--extend --seconds 1 --max-solutions 1
```

Join the two displayed command lines into one line before running in Command Prompt. Wrapped command blocks in this chapter follow that convention; join them with spaces, without adding a newline. The first equation asks for c; the second asks for d. The expected answers are $c = 10$ and d approximately 0.428571428571428571....

How calculator-style input works

Use constant for fixed inputs and ordinary named variables for answers you want displayed. An equality is a mathematical relationship, not a programming assignment: $c=a+b$ becomes an equation that the program must satisfy. Related equations can therefore appear in any order, although writing them in dependency order is easier to read.

A bare expression has a different meaning. Input such as $3+7$ means $3+7=0$; it does not request the value 10. Write $\text{answer}=3+7$ to calculate and display it. Do not add minimize or maximize to ordinary calculator worksheets.

Use a worksheet for repeated work

```
optimizer.exe --file 01_basic.txt --extend
--seconds 1 --max-solutions 1
```

The accompanying examples archive contains all 18 numbered worksheets. Edit fixed inputs, save the file, and run it again. Use `--file` explicitly when supplying other options.

Write clear, reusable calculations

Fixed inputs, intermediate constants and visible results

```
/* Fixed inputs and named intermediate values */
constant radius=5, diameter=2*radius;
constant square_area=diameter^2;
circle_area=#pi*radius^2;
area_ratio=circle_area/square_area;
constant force=12, extension=3;
```

```
e=force/extension;
euler_value=#e;
```

A constant initializer can use other declared constants and built-in functions. Multiple definitions share a declaration, for example constant $a=3, b=\max(2,4)$. You can also declare names first and define them separately. Dependencies may be written out of order, but circular definitions are invalid. A constant cannot depend on an unknown answer variable. To display an intermediate constant, add an ordinary result variable equal to it.

Operators and grouping

Expression	Meaning / result
$2+3*4$	Multiplication first: 14
$(2+3)*4$	Explicit grouping: 20
2^3 or 2^{32}	Right-associative power: 512
-3^2 / $(-3)^2$	Negative of a square: -9 / square of -3: 9
$x/2y$	Equivalent to $(x/2)y$; write $x/(2y)$ if intended
$2e3$ / $2*e$	Scientific notation 2000 / twice variable e

Explicit multiplication is the clearest style for calculator worksheets. The program also accepts $2x$, $2(x+1)$, $2\sin(x)$ and $x y$. The token xy is one variable, not $x*y$. Supported number formats include 0.25, .25, 1e-6 and 2E3. Names are case-sensitive; use letters or an underscore first, then letters, digits or underscores.

Statement separators and comments

Put one statement per line in a file, or separate statements with semicolons. Use `//` for a comment to the end of the line and `/* ... */` for a block comment. In a single-line command, a `//` comment would also hide the remaining statements, so prefer block comments or omit comments there.

Undeclared answers are real numbers. Even with integer inputs, division is real: `int a,b; a=3; b=7; ratio=a/b` gives a fractional ratio. Declaring ratio as an integer would forbid that result. `#e` is Euler's constant; bare `e` remains an ordinary variable. Both `pi` and `#pi` mean the circle constant.

Complete calculator function reference

Functions are lowercase and require parentheses. Every listed function takes exactly one or two arguments as shown. Angles are measured in radians. Formula values are real. Here, sophisticated or complex computations means involved real-valued formulas. Complex-valued arithmetic such as $\sqrt{-1}$ is not supported.

Functions	Purpose and important domain
$\sin(x)$, $\cos(x)$, $\tan(x)$	Trigonometric values. Avoid tangent poles at $\pi/2+k\pi$.
$\operatorname{asin}(x)$, $\operatorname{acos}(x)$	Inverse sine/cosine; $-1 \leq x \leq 1$. Results are radians.
$\operatorname{atan}(x)$	Inverse tangent; real x ; result in radians.
$\operatorname{atan2}(y,x)$	Direction of the vector (x,y) , with quadrant information. $(0,0)$ is invalid.
$\sinh(x)$, $\cosh(x)$, $\tanh(x)$	Hyperbolic sine, cosine and tangent; large magnitudes can overflow.
$\log(x)$, $\ln(x)$	Natural logarithm; $x > 0$. $\ln$ is an alias for $\log$.
$\log_{10}(x)$	Base-10 logarithm; $x > 0$.
$\exp(x)$	Exponential e^x ; very large positive x can overflow.
$\log_{1p}(x)$	Natural logarithm of $1+x$; $x > -1$; useful near zero.
$\expm1(x)$	$\exp(x)-1$; useful near zero.
$\sqrt{x}$, $\operatorname{cbt}(x)$	Square root needs $x \geq 0$. Cube root also accepts negative x .
$\operatorname{pow}(a,b)$	Real power a^b . Negative bases with noninteger exponents can be invalid.
$\operatorname{abs}(x)$	Absolute value.
$\min(a,b)$, $\max(a,b)$	Smaller/larger of two values. Nest calls for additional values.
$\operatorname{hypot}(a,b)$	Euclidean length $\sqrt{a^2+b^2}$.

Roots, limits and distance

```
cube_root=cbrt(-27);
fourth_root=pow(16,0.25);
clamped=max(0,min(1,1.2));
distance=hypot(5,12);
```

Expected results: -3, 2, 1 and 13. For a negative cube root, use `cbrt(-27)` rather than assuming a fractional power of a negative base will work. `min/max` are pointwise functions; they do not request optimization.

This expression language is not the whole of C++, Python or Excel. Do not assume `floor`, `ceil`, `round`, `factorial`, complex arithmetic, arrays or `if/then` expressions are built-in calculator functions. Percentage notation is also absent: write `18/100` or `0.18`.

Angles, trigonometry and coordinates

Convert degrees to radians explicitly

```
constant degrees=30, radians=degrees*#pi/180;
sine=sin(radians);
cosine=cos(radians);
tangent=tan(radians);
```

Expected values are 0.5, approximately 0.8660254038 and approximately 0.5773502692. `sin(30)` means the sine of 30 radians, not 30 degrees. To convert a radian result back to degrees, multiply by `180/#pi`.

Inverse angles and the correct quadrant

```
arcsine_degrees=asin(0.5)*180/#pi;
arccosine_degrees=acos(0.5)*180/#pi;
arctangent_degrees=atan(1)*180/#pi;
direction_degrees=atan2(3,4)*180/#pi;
```

The first three answers are 30, 60 and 45 degrees. The last is approximately 36.86989765 degrees. `atan2(3,4)` uses vertical component 3 and horizontal component 4. Prefer it over `atan(y/x)` when the quadrant matters or `x` may be zero. Its principal-angle convention is approximately $-\pi$ to π , with the endpoint sign depending on the inputs.

Resolve a length into Cartesian components

```
constant length=10, angle=60*#pi/180;
x_component=length*cos(angle);
y_component=length*sin(angle);
```

The components are 5 and approximately 8.660254038. They retain the same length unit as the input. The reverse calculation uses `length=hypot(x,y)` and `angle=atan2(y,x)`.

Avoid ambiguous angle calculations

`asin` and `acos` return principal inverse values; they do not list every periodic angle that could satisfy a sine or cosine equation. To find angles satisfying a relationship, use an equation and a meaningful bounded interval, as discussed in the inverse-calculation examples below. Evaluate `tan` only away from its poles. A large numerical value near a pole is not a reliable way to request infinity.

Worksheet: `06_angles.txt` includes all calculations on this page. Change degrees and length for surveys, vector components, geometry or rotations. Keep the degree-to-radian conversion visible so the worksheet remains understandable when shared.

Logarithms, growth and hyperbolic values

Natural, common and custom-base logarithms

```
natural_log=log(#e^2);
ln_alias=ln(#e^2);
base10_log=log10(1000);
base2_log=log(32)/log(2);
exponential=exp(1);
```

The results are 2, 2, 3, 5 and approximately 2.718281828459045. For any valid base b , $\log(x)/\log(b)$ calculates the logarithm of x to base b . Require $x>0$, $b>0$ and $b\neq 1$. These are mathematical domain requirements, not extra calculator keywords.

Small changes without avoidable cancellation

```
small_log=log1p(1e-10);
small_exp=expm1(1e-10);
```

The results are approximately $9.9999999995e-11$ and $1.00000000005e-10$. The dedicated functions are preferable to $\log(1+x)$ and $\exp(x)-1$ when x is tiny: adding 1 or subtracting nearly equal values can lose useful digits. Extended precision helps, but choosing a suitable expression still matters.

Hyperbolic functions

```
hyperbolic_sine=sinh(1);
hyperbolic_cosine=cosh(1);
hyperbolic_tangent=tanh(1);
```

Approximate results are 1.175201194, 1.543080635 and 0.761594156. These functions arise in catenaries, growth/decay relationships and scientific formulas. Their names are distinct from $\sin$, $\cos$ and $\tan$. This build does not expose inverse hyperbolic functions as calculator functions.

Understand invalid inputs

$\log(0)$, $\log(-1)$, $\sqrt{-1}$, $\arcsin(2)$ and division by zero do not have finite real values in this calculator. Do not interpret a failed solve as a valid numerical answer. If a function becomes invalid only for some unknown inputs, restrict those inputs to a meaningful domain, such as a positive interval for an argument of $\log$.

Worksheet: 07_logs_and_hyperbolic.txt exercises every function on this page. Avoid deriving Euler's constant from a guessed decimal when $\#e$ already supplies the built-in value. Do not use e without $\#$ unless it is deliberately your own variable.

Invoices, compounding and loan arithmetic

A transparent invoice worksheet

```
constant price=1000, discount=0.10, tax_rate=0.18;
net_price=price*(1-discount);
tax_amount=net_price*tax_rate;
invoice_total=net_price+tax_amount;
```

The arithmetic gives net price 900, illustrative tax 162 and total 1062. The tax rate is a supplied input, not a built-in legal rule. Use decimal rate fractions: 0.18 means 18%. Monetary outputs are not automatically rounded to currency minor units.

Compounding with units declared in the comments

```
// Values are in lakhs; rates are decimal fractions.
constant principal_lakhs=1.5, annual_rate=0.05, years=5;
future_lakhs=principal_lakhs*(1+annual_rate)^years;
gain_lakhs=future_lakhs-principal_lakhs;
```

The result is approximately 1.914422344 lakhs, a gain of 0.414422344 lakhs. This assumes annual compounding at a constant illustrative rate, with no contributions or withdrawals. A stated period must match the rate period. The worksheet predicts no actual investment return.

A fixed-rate, end-of-month loan-payment model

```
constant principal=150000, annual_rate=0.12, months=60;
constant monthly_rate=annual_rate/12;
payment=principal*monthly_rate/(1-(1+monthly_rate)^(-months));
total_paid=payment*months;
interest_paid=total_paid-principal;
bound payment=[0,10000];
bound total_paid=[0,300000];
bound interest_paid=[0,200000];
```

Approximate answers: payment 3336.67, total paid 200200.03 and interest paid 50200.03. The formula assumes 60 equal end-of-month payments, monthly rate 0.12/12 and no fees. At exactly zero rate, use principal/months instead; the displayed formula divides by zero there.

Bounds are supplied for the large monetary results. They are valid answer ranges, not currency formatting. The program uses finite search limits even for calculator worksheets; providing generous, meaningful bounds prevents a large answer from falling outside those limits. Example files 08-10 are ready to edit.

Engineering and small statistical worksheets

Power, energy and a geometric volume

```
constant voltage=230, current=5, hours=4;
power_watts=voltage*current;
energy_kwh=power_watts*hours/1000;
constant radius_m=2, height_cylinder_m=3;
volume_m3=#pi*radius_m^2*height_cylinder_m;
```

Results: 1150 W, 4.6 kWh and approximately 37.69911184 cubic metres. $P=V*I$ is appropriate for DC or an ideal unity-power-factor single-phase example; it is not a complete three-phase or general AC power model. Units are carried by your formulas and names, not checked by the executable.

Logarithmic scientific quantities

```
constant concentration=1e-3;
pH=-log10(concentration);
```

This simplified calculation gives 3. It treats the supplied positive number as the relevant dimensionless activity approximation. More detailed chemistry must supply its own model and corrections. Optimizer evaluates the relationships you write; it does not infer the physical assumptions.

Mean, population variance and standard deviation

```
constant v1=10, v2=12, v3=14;
mean_value=(v1+v2+v3)/3;
```

```
population_variance=((v1-mean_value)^2+(v2-mean_value)^2+(v3-mean_value)^2)/3;
standard_deviation=sqrt(population_variance);
```

Results: mean 12, population variance 2.666666667 and population standard deviation 1.632993162. The denominator is 3 because the formula defines a population variance. For the usual unbiased sample variance of these three observations, divide the squared-deviation sum by 2 instead.

Build larger calculations deliberately

Use named intermediate results to keep formulas readable. Use constants for fixed conversion factors, measurements and coefficients. Add result bounds when outputs can be very large. This calculator has scalar expressions rather than spreadsheet cell ranges or arrays; use Python to generate equations for a longer table of observations.

Worksheets: 11_engineering.txt also includes a simple mass/height index calculation; 12_small_statistics.txt contains the statistics example. Values in examples are illustrative mathematical inputs, not a medical, electrical-design or compliance recommendation.

Go beyond evaluation: find an unknown input

An equation-based calculator can solve backwards. Write the known measurements as constants, leave the required input as an ordinary variable, and use bounds to select the physically meaningful branch. A direct inverse formula and an equation solve are both possible; choose the clearer representation.

How long until a decaying quantity reaches 20%?

```
constant fraction=0.2, time_constant=5;
exp(-t/time_constant)=fraction;
bound t=[0,100];
```

The answer is approximately $t=8.047189562$ in the same time unit as `time_constant`. You can also evaluate $t=-\text{time_constant} \cdot \log(\text{fraction})$ directly. The bounded equation is useful when the relationship becomes more complicated and no convenient inverse formula is available.

Recover an illustrative growth rate

```
constant initial=100, final_value=121, periods=2;
initial*(1+r)^periods=final_value;
bound r=[0,1];
```

The answer is $r=0.1$, or 10% per period. Without the positive-rate bound, a square can introduce an unintended negative-base branch. The bounds express the desired mathematical range; they do not establish whether a future return is achievable.

One relationship may have multiple answers

```
optimizer.exe --file 15_two_roots.txt --extend
--seconds 2 --max-solutions 2
```

The file contains $x^2=169$ and bound $x=[-20,20]$. Both -13 and 13 are valid. To request only the nonnegative root, use 16_positive_root.txt, which bounds x between 0 and 20. `--max-solutions 1` is suitable for a worksheet with a single intended answer; it does not prove that an equation has only one root.

Enough relationships, but not conflicting definitions

Every unknown needs enough independent information. $a+b=10$ alone has infinitely many real pairs. Adding $a*b=21$ gives two pairs, (3,7) and (7,3); an extra range can select one. Likewise, $x=3$ and $x=4$ are conflicting equations, not two sequential assignments. Integer declarations restrict admissible inputs; they do not provide automatic rounding.

For ordinary formulas, compute fixed inputs first as constants. For inverse problems, start with sensible bounds and a modest time budget, and inspect the original-equation residual. A missing answer is a failed search or invalid model, not proof that no answer exists.

Precision, messages and Python automation

Option / message	Practical interpretation
--extend	Request extended precision if the executable was built with support.
--tol T	Maximum allowed absolute equation residual. Defaults: 1e-9 standard, 1e-24 extended.
--seconds S	Approximate time budget. Start with 1-2 seconds for simple worksheets, not 0.01.
--max-solutions N	Stop after N accepted answers; use 1 for a unique calculator worksheet.
--bound name LOW HIGH	Specify an answer interval on the command line. File form: bound name=[LOW,HIGH].
Time budget reached	Accepted displayed answers remain valid at the requested residual tolerance; completeness is separate.
No explicit bounds ...	Search limits are finite. Bound large or branch-sensitive answers explicitly.

Extended precision is finite numerical arithmetic, not symbolic exact arithmetic or unlimited digits. A long printed decimal does not guarantee that every digit is correct. Residual tolerance measures how closely the equations are satisfied, not a universal number of correct answer digits. For a simple $d=3/7$ equation, residual and answer error closely agree; nonlinear sensitivity can change that relationship. Check meaningful digits and units before using a result.

Use Python loops with the command-line calculator

```
import subprocess
for radius in [2, 5, 10]:
    model = f"constant r={radius}; area=#pi*r^2"
    run = subprocess.run(
        ["optimizer.exe", "--eq", model, "--extend",
         "--seconds", "1", "--max-solutions", "1"],
        capture_output=True, text=True, check=True)
    print(run.stdout)
```

Use an argument list, so Python passes the complete worksheet as one argument. With the supplied optimizer_ctypes.py adapter and a matching optimizer.dll, the equivalent in-memory call is:

```
from optimizer_ctypes import OptimizerDLL
engine = OptimizerDLL("optimizer.dll")
result = engine.solve("constant a=3,b=7; d=a/b",
    extend=True, seconds=1, max_solutions=1)
if not result["feasible"]:
    raise RuntimeError(result["message"])
print(result["stdout"])
```

The adapter's ordinary numeric values can be Python floats; preserve the printed decimal text when extended digits matter rather than converting it back to float. This DLL example was checked with the matching native shared library. Windows requires matching Python/DLL architecture and the DLL's runtime dependencies.

Final checklist: name each result; define fixed inputs; use radians; respect domains; provide answer bounds where needed; inspect residuals; keep a reusable worksheet. Included: 18 checked worksheets, Python examples and a validation record.

A long scientific worksheet: an AC circuit

This worksheet uses fixed inputs, intermediate constants, square-root distance, quadrant-aware angles and chained results. It represents a simple series resistance-inductance-capacitance circuit at one frequency. It does not check physical units automatically.

```
/* AC circuit: volts, ohms, henries, farads and hertz. */
constant voltage=230, resistance=40;
constant inductance=0.12, capacitance=0.00008, frequency=50;
constant omega=2*pi*frequency;
constant xl=omega*inductance, xc=1/(omega*capacitance);
constant reactance=xl-xc;
impedance=hypot(resistance,reactance);
current=voltage/impedance;
phase_degrees=atan2(reactance,resistance)*180/#pi;
power_factor=cos(atan2(reactance,resistance));
real_power=voltage*current*power_factor;
reactive_power=voltage*current*sin(atan2(reactance,resistance));
bound impedance=[0,1000];
bound current=[0,100];
bound phase_degrees=[-90,90];
bound power_factor=[0,1];
bound real_power=[0,2000];
bound reactive_power=[-2000,2000];
```

Input or result	Meaning in simple English
voltage=230	RMS supply voltage, in volts.
resistance=40	Resistance, in ohms.
inductance=0.12	Inductance, in henries.
capacitance=0.00008	Capacitance, in farads; this is 80 microfarads.
frequency=50	Frequency, in hertz.
omega=2pifrequency	Convert frequency to angular frequency, in radians per second.
xl and xc	Inductive and capacitive reactances, in ohms. They are fixed intermediate constants.
reactance=xl-xc	Net reactance. A negative value means the capacitive contribution is larger here.
impedance	Magnitude of resistance and net reactance, in ohms.
current	RMS current, in amperes.
phase_degrees	Impedance angle, converted from radians to degrees.
power_factor	Cosine of the impedance angle.
real_power and reactive_power	Active power in watts and signed reactive power in volt-amperes reactive.
bound ...	Inclusive allowed result intervals, provided as physical search domains. They do not perform unit conversion.

```
optimizer.exe --file calculator\17_ac_circuit.txt --extend --seconds 2 --max-solutions 1
```

Expected approximate values are impedance 40.05454 ohms, current 5.74217 amperes, phase -2.99045 degrees, power factor 0.998638, active power 1318.90061 watts and reactive power -68.90016 VAR. These rounded numbers help check the worksheet; they are not a substitute for the full returned values. Constants are not automatically displayed. To display omega too, add omega_result=omega.

Solve an involved formula backwards: two triangle shapes

Known sides are 4 and 5, and the required area is 8. The third side c is unknown. Heron's formula uses peri, the semiperimeter. Unlike a one-way calculator, Optimizer can find both permitted shapes.

```
/* Solve backwards for a triangle side; all lengths in the same unit. */
constant a=4, b=5;
bound c=[1,9];
bound peri=[4,9];
peri=(a+b+c)/2;
sqrt(peri*(peri-a)*(peri-b)*(peri-c))=8;
c>=1;
```

```
c<=9;
```

```
optimizer.exe --file calculator\18_inverse_triangle.txt --extend --seconds 2 --max-solutions 2
```

There are two solutions: $c=\sqrt{17}$, approximately 4.12310562561766, and $c=\sqrt{65}$, approximately 8.06225774829855. Their semiperimeters are $(9+c)/2$. The square root requires a nonnegative radicand, and the side must satisfy the triangle inequality. The inclusive conditions restrict admissible roots; they do not request a minimum or maximum. The companion validator recomputes the area from each solution.

Display high-precision computations in an application

```
from decimal import Decimal, localcontext
from optimizer_ctypes import OptimizerDLL

engine = OptimizerDLL("optimizer.dll")
for a, b in [(3, 7), (22, 7), (1, 3)]:
    result = engine.solve(
        f"constant a={a}, b={b}; quotient=a/b",
        extend=True, tol="1e-24", seconds=2, max_solutions=1)
    if not result["feasible"]:
        print("No accepted answer:", result["message"])
        continue
    text = result["best"]["values_text"]["quotient"]
    print(f"{a}/{b} = {text}")
    with localcontext() as ctx:
        ctx.prec = 40
        decimal_value = Decimal(text)
        print("100 times that value:", decimal_value * 100)
```

The loop, formatting and branching belong to Python; the equations use Optimizer's expression language. Construct `Decimal` from `values_text`, not from the convenience float field. Binary128 has 113 significand bits, roughly 33 significant decimal digits; extra printed digits and a small residual do not guarantee exact arithmetic or correct digits for an ill-conditioned inverse calculation. Requested time is an allowance, not a promise that every worksheet completes within a few milliseconds.

6. General linear optimization and mixed-integer models

General linear optimization: model format

A linear program uses an affine objective and affine constraints. An affine expression is a constant plus a sum of constant coefficients times variables. Variable products, variable denominators and nonlinear functions of decisions are not affine.

```
constant price1=3
constant price2=2
maximize price1*x+price2*y
x+y<=4
x<=2
x>=0
y>=0
```

Native input accepts the same comments, constants, implicit multiplication and bound declarations as general mathematical input. `--linear` forces the linear service and rejects nonlinear expressions. `--lp` and `--milp` are aliases; declarations determine whether integer search is needed.

Variables and constraints Undeclared continuous variables are free, not automatically non-negative. Explicitly write `x>=0` when the quantity cannot be negative. Constraints accept `=`, `<=` and `>=`. For linear feasibility without an objective, use `--linear` explicitly.

Commands

```
optimizer.exe --linear --file models/lp_production.txt --seconds 10
optimizer.exe --lp-file model.lp --seconds 20
optimizer.exe --mps-file model.mps --seconds 20
```

The import formats have their own defaults and supported subsets; see the relevant service option reference. Do not paste LP section headers into a native expression file.

When to choose LP/MILP Use it for production quantities, inventory balances, resource allocation, selection and shipment models that really are linear. It can report numerical objective bounds and gaps. A resource-limited feasible incumbent can be useful without a completed optimality conclusion.

Use `--mip-seed` for a single-seed linear heuristic run. Root/nonlinear controls such as `--starts` are not LP tuning parameters.

LP example: production with two resources

```
maximize 3*x+2*y
x+y<=4
x<=2
x>=0
y>=0
```

A workshop produces x and y . Each unit contributes profit 3 or 2. Total production cannot exceed four units, and at most two units of x can be produced. Quantities are continuous for this first model.

```
optimizer.exe --linear --file models/lp_production.txt --seconds 10
```

The optimum is $x=2$, $y=2$, objective=10. Profit increases with either variable, so the total production limit is active; the higher-profit product reaches its separate upper limit.

Change the represented problem carefully Adding $\text{int } x, y$ would restrict quantities to whole numbers. In this example the continuous optimum is already integral, so the best objective stays 10. In another model, the relaxation can be fractional and its objective becomes a bound rather than an implementable integer plan.

Validate the answer Check $x \geq 0$, $y \geq 0$, $x \leq 2$ and $x + y \leq 4$, then recompute $3x + 2y$. A correctly reported objective cannot compensate for a missing physical constraint.

Parameters Replace numeric prices with declared constants if several rows share them. Supply measured prices in consistent currency units. Scaling a cost by 100 changes the objective units and the meaning of an absolute gap target, but not the relative ordering of feasible plans.

The native LP conclusion is numerical within the selected tolerances. Save the bound/gap and stopping reason alongside the variables when an audit trail is needed.

Linear production: each coefficient explained

Define the decision quantities, objective units and resource use per unit. Products are divisible in this continuous LP. To require whole units, add an integer declaration and use the MILP formulation. This example reaches $x=2$, $y=2$, profit 10.

Statement	Meaning in simple English
maximize $3x + 2y$	Choose production x and y to maximize profit. Profit per unit is 3 for x and 2 for y .
$x + y \leq 4$	Each product uses one unit of a shared resource; only four resource units are available.
$x \leq 2$	At most two units of product x may be produced, for example because of demand.
$x \geq 0$	Production of x cannot be negative.
$y \geq 0$	Production of y cannot be negative.

Complete companion file

```
optimizer.exe --linear --file models/lp_production.txt --seconds 10
```

MILP example: integer production

```
int x,y
maximize 5*x+4*y
2*x+y<=8
x+2*y<=8
bound x=[0,8]
bound y=[0,8]
```

Run:

```
optimizer.exe --linear --file models/milp_production.txt --seconds 10
```

The best integer plan is $x=3$, $y=2$, objective=23. Resource use is 8 and 7, respectively. The continuous relaxation has $x=y=8/3$ and objective=24, but those are not whole production quantities.

Integer restrictions int and signed integer declarations are hard domains. binary represents a yes/no choice. The solver checks rounded integer candidates against the original constraints, so simply displaying a relaxed value with zero decimals would not make it a valid integer plan.

Useful controls `--mip-nodes` limits integer-search nodes. `--lp-pivots` limits linear numerical work. `--mip-gap` and `--mip-abs-gap` allow termination at a requested gap when a valid bound is available. `--mip-seed` selects the randomized heuristic seed without switching normal linear dispatch.

```
optimizer.exe --linear --file models/milp_production.txt --mip-gap 0.01 --seconds 30
```

A 1% gap target is a stopping criterion, not a promise that an arbitrary model will reach it. If

no incumbent is found within a limit, the model is not thereby proved infeasible.

For very large binary models, natural finite bounds and meaningful formulations matter. Dedicated exact-cover pattern search can be helpful only when its stricter model scope is satisfied; additional inequalities belong in the general MILP.

Integer production: each statement explained

Use integer declarations for indivisible products, staff counts or on/off decisions. Bounds do not themselves make a variable integer. The verified example has $x=3$, $y=2$, profit 23.

Statement	Meaning in simple English
int x,y	Both product quantities are integers. They cannot take values such as 2.5.
maximize $5x+4y$	Each unit of x contributes 5 profit units; each unit of y contributes 4.
$2x+y \leq 8$	Resource 1: an x uses 2 units, a y uses 1; capacity is 8.
$x+2y \leq 8$	Resource 2: an x uses 1 unit, a y uses 2; capacity is 8.
bound $x=[0,8]$	Allow whole-number x values from 0 through 8.
bound $y=[0,8]$	Allow whole-number y values from 0 through 8.

Complete companion file

```
optimizer.exe --linear --file models/milp_production.txt --seconds 10
```

Binary choices and logical linear constraints

```
binary buy1,buy2,buy3
nonnegative int reserve
4*buy1+3*buy2+2*buy3+reserve=5
maximize 8*buy1+6*buy2+5*buy3
```

The three buy variables select items; reserve is unused integer capacity. The best selection is $buy1=0$, $buy2=1$, $buy3=1$, $reserve=0$, $objective=11$.

```
optimizer.exe --linear --file models/binary_selection.txt --seconds 10
```

Common binary rules | Requirement | Linear constraint | |---|---| | Exactly one of a,b,c | $a+b+c=1$ | | At most one | $a+b+c \leq 1$ | | At least one | $a+b+c \geq 1$ | | Choosing a requires b | $a \leq b$ | | a and b cannot both be selected | $a+b \leq 1$ | | Selection controls a bounded quantity q | $q \leq M*a$; $q \geq 0$ |

For the last rule M must be a justified finite upper limit. An unnecessarily enormous value can weaken a model and worsen numerical scaling. A tight bound is useful data, not a tuning trick.

Pattern coverage versus general MILP The --crew service supports exact-one coverage equations only. General covering (≥ 1), packing (≤ 1), prerequisites and resource limits are valid MILP constraints, but they are not accepted as arbitrary side rules by --crew.

No general if/then programming statement is supported in native text. Express the permitted logic with binary variables and supported constraints, or use an appropriate dedicated service. Validate that the algebra actually matches the intended business rule before solving.

LP/MILP bounds, gaps and stopping

Native linear variables are free unless constrained. binary and signed integer declarations introduce their hard domains. --bounds and --bound remain available, with the documented override precedence. Do not confuse an unrestricted linear variable with an automatically bounded nonlinear search variable.

Interpret a minimization report If a feasible incumbent costs 57,000 and a valid lower bound is 55,200, the remaining absolute gap is 1,800. This does not say the unknown optimum equals either value. It places it between them, subject to the report's numerical qualification.

For maximization, a valid upper bound is compared with a feasible lower objective. The native gap calculation respects objective direction. Relative LP/MILP gap uses $\max(1, |\text{incumbent objective}|)$.

Status is more informative than runtime A node limit can leave a useful feasible plan with optimality unestablished. An unresolved numerical result must not be shown as a certified infeasible or unbounded MILP. An unbounded integer relaxation is not by itself a proof that the integer model is unbounded.

Tuning without changing the problem Increase `--seconds`, `--mip-nodes` or `--lp-pivots` only when the stopping reason indicates that cap. `--mip-heuristic-moves` and `--mip-dive-nodes` affect feasible-candidate discovery. `--no-mip-heuristics` disables those aids. Optional `--mip-clique-cuts` applies only to eligible packing/partition structures; no bound or objective improvement is guaranteed.

Numerical accuracy `--tol/--ctol` controls original constraint feasibility. `--mip-int-tol` controls integer recognition before rounded verification. A looser recognition tolerance does not permit fractional declared integer values in the final candidate. `--extend` changes applicable arithmetic, not the underlying requirements.

see the relevant service option reference for every linear and crew setting and its default.

7. General nonlinear and mixed-integer optimization

General nonlinear optimization: format

For the scalar workflow in this chapter, use the mathematical grammar with one objective and any supported equalities, inequalities, constants, bounds and integer declarations. A continuous nonlinear model is NLP; adding discrete domains makes it a mixed-integer nonlinear model.

```
minimize (x-3)^2+(y-2)^2
x+y>=4
bound x=[0,5]
bound y=[0,5]
```

```
optimizer.exe --file nonlinear.txt --seconds 20
```

The objective is minimized while every original constraint remains active. A bare nonlinear expression without a comparison is an equality to zero, not an automatically interpreted objective or a free-standing rule.

Useful applications Curve fitting, engineering design, bounded parameter calibration and mathematically formulated allocation problems. Pricing, finance, traffic and engineering models are not built in: supply the actual equations and units required by your application.

Result interpretation The best feasible objective is an achievable result, not necessarily a global optimum. First-order convergence is a local property. Integer feasibility does not certify the best integer assignment. Nonconvex or nonsmooth objectives can have many local solutions.

Bounds and precision Meaningful finite bounds help exploration and invalid-domain avoidance. With unspecified bounds the engine explores finite automatic boxes. --extend requests the supported higher-precision path but does not certify global optimality or remove conditioning difficulties.

Use --sqp explicitly when forcing the nonlinear service for an affine test. For a genuinely nonlinear expression it is normally selected automatically. --gtol and --ftol are nonlinear controls; do not send them to a dedicated route or operational service.

NLP example: a two-asset risk model

```
minimize 0.04*w1^2+0.09*w2^2
w1+w2=1
0.12*w1+0.08*w2>=0.10
bound w1=[0,1]
bound w2=[0,1]
```

Run:

```
optimizer.exe --file models/nlp_portfolio.txt --seconds 10
```

Here w_1 and w_2 are fractions, sum to one and cannot be negative. The expected return must be at least 0.10. The illustrative objective has no covariance term.

The optimum is $w_1=9/13$, $w_2=4/13$, approximately 0.6923076923 and 0.3076923077. Objective is 0.0276923076923 and modeled expected return is 0.1076923076923. The return requirement is satisfied without being tight.

Check the represented assumptions A real portfolio risk model normally needs the full appropriate covariance structure and accurately estimated inputs. This example demonstrates

syntax; it is not an investment recommendation. The engine minimizes exactly the expression supplied.

Extended run

```
optimizer.exe --file models/nlp_portfolio.txt --extend --tol 1e-24 --seconds 20
```

Use precision appropriate to the input coefficients. Reporting many digits cannot improve uncertain risk estimates.

Alternative example models/nlp_product.txt maximizes $x*y$ with $x+y=10$ and both variables in $[0,10]$. Its optimum is $x=y=5$, objective 25. This illustrates maximization with a nonlinear objective and a linear equality.

Nonlinear portfolio model: each expression explained

Use decimal fractions: 0.10 means 10%, not 10. Decide whether weights must total one or whether cash is represented separately. This mathematical example has no share rounding, trading charges, estimated-price uncertainty or built-in financial data.

Statement	Meaning in simple English
minimize $0.04w_1^2 + 0.09w_2^2$	Minimize this simplified squared-risk measure. Coefficients are variance-like inputs. There is no covariance term in this example.
$w_1 + w_2 = 1$	Invest all modeled wealth across the two assets; weights add to 100%.
$0.12w_1 + 0.08w_2 \geq 0.10$	Require modeled return of at least 10%, using input returns 12% and 8%.
bound $w_1 = [0,1]$	Asset 1 weight is between zero and 100%.
bound $w_2 = [0,1]$	Asset 2 weight is between zero and 100%.

Complete companion file

```
optimizer.exe --sqp --file models/nlp_portfolio.txt --seconds 10
```

NLP example: fit exponential decay

```
minimize (c*exp(-k*0)-10)^2+(c*exp(-k*1)-5)^2+(c*exp(-k*2)-2.5)^2
bound c=[0.1,20]
bound k=[0,3]
```

The observations are 10, 5 and 2.5 at times 0, 1 and 2. The model $c\exp(-kt)$ is fitted by minimizing the sum of squared residuals. c is positive and k is constrained to a finite nonnegative interval.

```
optimizer.exe --file models/nlp_fit.txt --seconds 10
```

The exact fit is $c=10$ and $k=\log(2)$, with mathematical objective zero. A tiny positive reported objective can be a numerical approximation to that fit. There are no separate equality constraints forcing each observation to match; the least-squares objective performs the fit.

Noisy measurements Replace the observations with measured data and retain the objective. Do not create exact equations for every noisy point unless exact interpolation is truly intended. With more observations, generate a sum of terms in Python rather than writing an unsupported `sum()` function into native input.

Weights and scaling If observations have different uncertainty or scales, provide justified residual weights. Units of k must agree with time. If timestamps are hours, k is per hour; if days, it is per day.

Application checks Recompute predictions from the returned c and k , inspect residuals against each observation and keep the fitted domain bounds. The engine does not supply statistical confidence intervals, outlier diagnosis or automatic model selection. Those tasks belong in the surrounding application.

For a parameter sweep, use the loop example on the corresponding option reference and pre-

serve the best feasible results rather than merely printing unverified numbers.

Curve fitting: building the residual objective

Collect observed (time,value) pairs and write one squared-error term per observation. Measurements are data, c and k are unknown parameters. Exact fit is $c=10$ and $k=\log(2)$. Do not equate each observation individually unless exact interpolation, rather than best fit, is intended.

Statement	Meaning in simple English
minimize (cexp(-k0)-10) ² +(cexp(-k1)-5) ² +(cexp(-k2)-2.5) ²	At times 0, 1 and 2, observations are 10, 5 and 2.5. Model prediction is cexp(-kt). Square each prediction error and sum them; minimize that total.
bound c=[0.1,20]	Initial amplitude c may range from 0.1 to 20 in observation units.
bound k=[0,3]	Decay rate k is nonnegative and at most 3 per chosen time unit.

Complete companion file

```
optimizer.exe --sqp --file models/nlp_fit.txt --seconds 10
```

MINLP example: choose trucks and load

```
positive int trucks
bound trucks=[1,5]
bound load=[0,10]
trucks*load>=12
minimize 4*trucks+trucks*load^2
```

A positive integer trucks specifies how many vehicles are used. Each has the same continuous load. Total delivered amount must be at least 12. The illustrative objective combines a fixed charge with a nonlinear load cost.

```
optimizer.exe --file models/minlp_trucks.txt --seconds 15
```

For fixed n trucks the lowest feasible load is $12/n$. The objective becomes $4*n+144/n$. For $n=1,...,5$, the minimum occurs at $n=5$, $load=2.4$, $objective=48.8$. This can be checked independently by enumerating five choices.

Model scope This is a small mathematical demonstration, not a complete delivery-routing model. It does not decide routes, vehicle-specific loads or delivery windows. Use dedicated routing services when those are the requirements.

Domains and bounds positive int excludes zero, avoiding an unusable zero-vehicle choice. bounds provide finite limits. A real relaxed solution is not an implementable truck count; the reported candidate must satisfy the declared integer domain.

Search expectations Small integer boxes may be enumerated subject to --integer-enumeration. Larger discrete spaces use heuristics; global integer optimality is not generally certified. --integer-neighbors controls additional local integer alternatives, while --integer-nodes and --integer-radius apply to the documented integer seed search.

These are nonlinear/integer controls, not the native MILP --mip-nodes option. Choose settings that match the selected service.

Mixed-integer nonlinear model: every statement explained

This simplified model assumes equal load across identical trucks. It does not describe delivery routes or individually different vehicle loads. The example minimum is trucks=5, load=2.4, cost=48.8.

Statement	Meaning in simple English
positive int trucks	Choose a whole number of trucks, at least one.
bound trucks=[1,5]	At most five trucks are available.
bound load=[0,10]	Each used truck carries the same modeled continuous load, from zero to ten.
trucks*load>=12	Total carried load must be at least twelve. A variable multiplied by another variable makes this nonlinear.
minimize 4trucks+trucksload^2	Pay 4 per truck plus the stated quadratic load cost for each truck.

Complete companion file

```
optimizer.exe --sqp --file models/minlp_trucks.txt --seconds 10
```

Conditions without an optimization objective

A useful condition restricts acceptable equation solutions without introducing a cost to minimize or maximize.

```
constant a=4, b=5;
peri=(a+b+c)/2
sqrt(peri*(peri-a)*(peri-b)*(peri-c))=8
c>=0
```

Run the default equation service:

```
optimizer.exe --file models/triangle_conditions.txt --extend --seconds 2
```

Here a and b are known sides. peri is the semiperimeter, and the square-root formula is the triangle area. Two acceptable third sides are $c=\sqrt{17}$, approximately 4.123105626, and $c=\sqrt{65}$, approximately 8.062257748. The corresponding peri values are $(9+c)/2$.

Syntax LHS<=RHS and LHS>=RHS include equality. Write a separate statement for each condition. Do not write $0 \leq x \leq 3$: use bound $x=[0,3]$ or the two statements $x \geq 0$ and $x \leq 3$. Strict > and < are not supported. Conditions are checked against the original model with the requested tolerance.

Choose the result you need Default equation mode seeks distinct acceptable roots. Explicit -sqp asks for nonlinear feasibility and may return only one acceptable candidate. Explicit --linear is suitable only for affine expressions. Add minimize or maximize when an actual preference matters.

An equation system can have infinitely many acceptable solutions. Neither multiple returned roots nor a long search guarantees completeness. If no acceptable point is found under a search limit, that alone does not prove the conditions are impossible.

Tune nonlinear optimization sensibly

Start with default feasibility tolerance and realistic bounds. Confirm that the objective and every hard rule are represented correctly. A highly precise solution to an incomplete model is not useful.

Exploration controls --starts increases the restart allowance; --iterations increases the local iteration cap. --radius, --stages and --growth affect unspecified search ranges. --seed chooses a single exploratory seed. --max-solutions retains multiple distinct feasible candidates rather than certifying several optima.

Acceptance controls --tol/--ctol concerns constraint feasibility. --gtol concerns local stationarity. --ftol concerns relative objective stagnation. Tightening one does not automatically tighten the others. In extended mode, avoid requesting accuracy beyond what conditioning and coefficient data support.

Integer controls --integer-enumeration sets the size of a small assignment box eligible for enumeration. --integer-neighbors controls additional nearby integer alternatives. --integer-nodes and --integer-radius affect the extra all-integer seed search where applicable. These improve

opportunities, not guarantees.

When progress stalls Try a different seed or multi-seed search, review the bounds, simplify badly scaled expressions and check nonsmooth terms such as `abs/min/max`. A model with several separated feasible regions may need more exploration rather than stricter polishing.

A useful comparison Record feasible objective, maximum original violation, convergence status and elapsed time for each run. Preserve the best accepted candidate. A lower objective from an infeasible point is not an improvement.

If an automatic boundary is active, inspect whether the physical model needs a wider range or a true bound. Automatic boxes do not certify that an otherwise unbounded optimization problem has a finite global optimum.

New nonlinear controls and continuation

`--nlp-sparse` and `--nlp-memory` tune the nonlinear algebra/history path; Chapter 35 gives scope and ranges. Blend/Pareto and affine priority objectives have separate Chapters 32-33. The bounded checkpoint scheduler is documented in Chapter 34, including its current cold-start quality limitation. Treat default timed search as your quality baseline; neither more work units nor tighter feasibility tolerance guarantees a better objective.

8. Numerical global optimization and objective gaps

Use `--global-opt` when you need a numerical objective bound as well as a feasible candidate. It is optional; ordinary nonlinear solving remains available for a wider range of functions. This mode concerns only the explicitly declared finite domain, and its result must be read with the printed numerical qualifications.

Model format and requirements

Use the standard `--MODE--filePATH` form, here `--global-opt--filePATH`. The file uses the existing general equation/optimization syntax. It must contain exactly one minimize or maximize objective and explicit finite bounds on every variable. Bounds can be file declarations, command-line bounds, or bounded declarations such as binary.

Example:

```
bound x = [-2, 2]
bound y = [-2, 2]
minimize x^4 + y^4 - 2*x^2 - 2*y^2
x + y >= 0
```

`boundx=[-2,2]` allows x to be any real value between -2 and 2 , including both endpoints. The objective asks for the smallest value of the displayed expression. The condition requires the sum to be nonnegative. The known global minimum for this example is -2 , attained at several feasible points, including $x = y = 1$.

```
optimizer.exe --global-opt --file models\global\quartic.txt --seconds 5
```

An omitted bound does not silently become a global domain. The solver gives an error naming the variable that needs a bound. Supply bounds that follow from the actual application: a global claim concerns only the declared domain. Giving an excessively narrow artificial box changes the problem; giving an excessively wide box can make bounding much slower.

Command-line bounds still override file bounds:

```
optimizer.exe --global-opt --file model.txt --bound x -10 10 --seconds 60
```

Strict `<` and `>` remain unsupported. Use inclusive `<=`, `>=`, and `=`. General equations with conditions and no objective continue to use the existing equation-solving behavior; they are not accepted by `--global-opt`.

New options

Option	Default	Meaning
<code>--global-opt</code>	Off	Enable the numerical global bounding mode.
<code>--global-gapR</code>	1e-4	Relative objective-gap target, a finite number ≥ 0 . 1e-4 means 0.01%.
<code>--global-abs-gapA</code>	0	Absolute objective-gap target, a finite number ≥ 0 , measured in objective units.
<code>--global-nodesN</code>	10000	Maximum number of processed spatial search nodes; integer ≥ 1 .
<code>--global-pivotsN</code>	1000000	Total LP iteration limit for the global search; integer ≥ 1 .
<code>--global-help</code>	Not applicable	Explain this mode, its supported functions, and its limits.

A gap or work-limit option above also enables global mode. Either the relative target **or** the absolute target can stop the search. Increase work limits as well as time if the output reports a work limit. Zero node/iteration limits are rejected; they do not mean unlimited.

Existing important options:

Option	Use in this mode
--secondsS	Total time allowance shared by feasible-candidate discovery, refinement, and bounding. Zero disables the time limit, but work limits remain.
--tolT, --ctoIT	Tolerance used to verify original constraints. This is separate from the objective gap.
--extend	Refine feasible candidates using binary128 when the build supports it.
--seedN	Seed for the incumbent discovery search. It can affect which feasible solution is found and subsequent progress.
--startsN, --iterationsN	Controls for initial feasible-candidate discovery.
--boundNAMELOWHIGH	Override one variable's finite bounds.
--boundsLOWHIGH	Supply a common finite domain for variables without a more specific command-line override.
--progress	Report improving candidates and occasional node-bound progress. The final reported bound covers the entire unresolved frontier, whereas a progress line can describe one node.

--global-search, --global-boxes, and --no-global-search control the existing **heuristic** discovery/exploration phase. They do not calculate a global bound by themselves. They are distinct from --global-opt.

Do not combine global nonlinear controls with --linear, --milp, LP/MILP-specific controls, or --crew. The existing LP/MILP backend remains the preferred dedicated path for linear models. Dedicated routing, cutting, assignment, scheduling, and other services retain their own controls.

Multi-seed calls may use this mode. Each trial performs a fresh bounded search of the entire declared domain. Feasible incumbent coordinates can be retained, but trial progress is not combined into a single continued proof search. Multi-seed search is consequently not a guaranteed improvement in proof time; start with a single global run.

Supported expressions

Category	Available in --global-opt
Arithmetic	+, -, *, /, parentheses; existing implicit multiplication syntax.
Powers	^, **, pow with an integer constant exponent from 0 through 16.
Trigonometry	sin, cos, with argument ranges within $\pm 10^{12}$. Arguments are radians.
Exponentials	exp, expm1.
Logarithms	log, ln, log10, log1p.
Roots	sqrt.
Piecewise expressions	abs, min, max. These can have weaker bounds.
Constants	Existing constant declarations, including constanta=3,b=2;, #pi, pi, and #e.
Variable domains	Existing real, int/integer, binary, positive/negative/nonnegative/nonpositive integer declarations and their aliases.

Division denominators must stay away from zero over their interval bounds. Logarithm arguments must have a strictly positive interval lower bound. A square-root domain is restricted to nonnegative arguments. Expression ranges must stay finite and reasonably scaled; an excessive range produces a clear error suggesting tighter bounds.

Functions not in the table, variable exponents, negative powers, and fractional powers are not supported by this first global mode. Use sqrt for a square root. The ordinary solver continues to support its wider function set. Constants containing functions are resolved before expressions reach the global mode, so a fixed numeric constant may use a function not otherwise supported in variable expressions.

If interval arithmetic cannot establish a safe logarithm or division domain even though you know the expression is safe, reformulate or tighten its bounds, or use the ordinary solver. The implementation prefers an explicit limitation to a misleading bound.

Example: a bounded nonlinear maximum

```
bound x = [0, 1]
bound y = [0, 1]
maximize x*y
x + y <= 1
```

The product must be made as large as possible while the two nonnegative values have a combined limit of 1. The global maximum is 0.25 at $x = y = 0.5$.

```
optimizer.exe --global-opt --file models\global\bilinear.txt --seconds 5 --global-gap 1e-4
```

For a maximum, the best objective bound is an **upper** bound. For a minimum, it is a **lower** bound. Both directions report a nonnegative gap.

Example: a bounded integer model

```
int units
bound units = [-5, 5]
minimize (units - 1.3)^2
```

The variable can take only integers -5 through 5 . The best value is $\text{units}=1$, with objective 0.09 .

```
optimizer.exe --global-opt --file models\global\integer.txt --seconds 5
```

Every returned candidate must satisfy the original integer declaration. The reported bound and gap describe remaining uncertainty; a feasible integer alone is not a proof of the integer optimum. Large combinatorial domains may still be very expensive.

Reading the result

Typical fields:

```
Global status: gap target met
Best feasible minimum found (... seconds).
Candidate 1: objective = ...
x = ... y = ...
Maximum constraint violation: ...
First-order convergence: ...
Best objective bound: ...
Absolute optimality gap: ...
Relative optimality gap: ...
Optimality: established within numerical tolerances
Bounds use conservative floating-point guards; exact-arithmetic certification is not provided.
Stopped: requested global gap reached.
```

The candidate is independently checked by the engine against the original expressions, declared bounds, and integer domains at `--tol`. A local first-order result is not the global result. Global status depends on the objective bound and the remaining gap.

For a minimum with feasible objective (U) and lower bound (L):

```
[ \text{absolute gap} = \max(0, U-L), \text{relative gap} = \frac{\text{absolute gap}}{|U|}. ]
```

For a maximum, subtract the feasible objective from the upper bound instead. A printed relative gap of 0.0009275 means approximately **0.09275%**, not 0.0009275% .

If the feasible objective is zero, the relative gap is infinite unless the absolute gap is also zero. Supply a practical absolute tolerance, for example `--global-abs-gap1e-6`, for zero-valued objectives. Setting both targets to zero demands a zero numerical gap and is usually unsuitable for nonlinear floating-point models.

On a time, node, or LP iteration limit, the best feasible candidate is retained. The reported

bound applies to the unresolved part of the declared problem. If the target was not met, Optimality: not established is printed. No feasible candidate is not a proof of infeasibility. Exit 0 means a verified feasible candidate was returned; it does **not** mean the gap target was met. Exit 1 means an input/unsupported-model error; exit 2 means no verified candidate was returned.

Choosing between exploration and a bound

Your need	Control	What the result establishes
Look in more regions for a better candidate	--global-search / --global-boxes	Better discovery may occur; no global objective bound follows from these controls alone.
Stop that additional exploration	--no-global-search	Ordinary restarts and local refinement remain; this does not change the physical model.
Obtain a numerical global bound and gap	--global-opt	Available only for the supported functions and finite domains. Read the final bound, gap and status.
Accept a practical relative gap	--global-gap 0.01	A 1% target, subject to available time and work. It does not promise a 1% result on every instance.
Accept an absolute gap near a zero-valued objective	--global-abs-gap 1e-6	Objective-unit target; avoids relying on a relative percentage near zero.

For a more difficult bounded nonlinear model, start with the ordinary service and a realistic feasibility tolerance, then try numerical global mode if the model is supported. Compare verified objectives and reported gaps, not only runtime. Tightening --tol does not request a tighter objective gap; --global-gap and --global-abs-gap do that. Increasing --seconds alone may not help when a separate work cap has already been reached.

9. Supplied binary exact-cover and partition patterns

Supplied binary patterns: what --crew accepts

The --crew service selects pre-generated patterns so that every required task is covered exactly once at minimum total cost. A binary variable represents selecting one legal pattern. Although named crew, the format can represent other exact-cover pattern selections.

Accepted model Affine minimization or feasibility; binary variables, or integer variables with hard domains inside $[0,1]$; coverage coefficients zero/one; right-hand side one; exact-one equalities. Fixed selections/exclusions, objective constants and unconstrained binary objective variables are retained.

```
binary p1,p2,p3
minimize 7*p1+6*p2+10*p3
p1+p3=1
p2+p3=1
```

The patterns themselves must already satisfy individual legality rules. For airline use this includes any required duty/rest/qualification rules used during pairing generation. The engine does not derive legal pairings from an employee database in --crew mode.

Not general covering/packing At-least-one covering inequalities, at-most-one packing inequalities, maximization, nonlinear terms and additional arbitrary resource constraints are rejected. Use --linear for those general binary models. The exact-cover title sometimes includes 'partition'; it does not mean all binary programs are supported here.

Other uses Select supplied delivery-trip patterns, machine schedule patterns, maintenance-team patterns or task groups, provided the actual decision is exact-one coverage. Do not discard a practical side constraint just to make the model eligible.

Input uses the mathematical text grammar, including comments, constants and bounds, rather than a separate employee-shift record grammar. --crew is explicit; it does not replace default MILP dispatch globally.

Supplied patterns: constructing the mathematical model

--crew selects among patterns you supply; it does not invent flight pairings or employee shifts. Declare one binary variable per eligible complete pattern. Its objective coefficient is the cost of choosing that pattern. A row lists which patterns cover one required task.

```
binary p1,p2,p3
minimize 7*p1+5*p2+9*p3
p1+p3=1
p2+p3=1
```

Here task 1 is covered by exactly one of p1 or p3; task 2 by exactly one of p2 or p3. The supported --crew service selects exact-cover/partition patterns with =1 rows and a minimization objective. General at-least-one covering (≥ 1), at-most-one packing (≤ 1), or other side constraints must use --linear instead. Do not append those inequalities to an exact-cover --crew model.

Build patterns only from operationally valid combinations. Include travel, qualifications and rest rules when generating them; the selection model does not infer those facts from a name. Use --crew --file PATH. General expressions with unsupported side constraints should use a general mathematical service instead.

Pattern-selection example

```
binary p1,p2,p3,p4
minimize 7*p1+6*p2+10*p3+5*p4
p1+p3=1
p2+p3=1
p1+p4=1
p2+p4=1
```

Run:

```
optimizer.exe --crew --file models/crew_patterns.txt --seconds 10 --seed 38
```

The four equations represent exact-one coverage of four tasks. Pattern p1 covers tasks 1 and 3; p2 covers tasks 2 and 4; p3 covers tasks 1 and 2; p4 covers tasks 3 and 4.

A cheapest complete selection is p1=p2=1, p3=p4=0, cost 13. The alternative p3=p4=1 costs 15. The returned plan is verified against all four original coverage equations.

Map the answer to operations The companion models/pattern_mapping.csv records names, costs and covered tasks. In an application, selected p1 and p2 are looked up in that table to recover the actual trips, pairings or groups. The generic equation variables do not themselves contain travel or staffing details.

Fixing a choice Adding bound p3=[1,1] forces that pattern. In this example it requires p4 as well, producing cost 15. A bound restricting a binary variable outside its declared domain is an input error.

A larger model can contain thousands of patterns. The useful result is a complete feasible selected set, its recomputed original cost and the reported bound/gap if available. A short heuristic run does not certify that no cheaper pattern combination exists.

Exact-cover patterns: interpreting every coverage row

Create each eligible pattern from your actual operational rules first. A column contains every task covered by that complete pattern. The cheapest example chooses p1=p2=1 and p3=p4=0, cost 13. --crew accepts this exact-one selection scope; general >= covering or <= packing uses --linear.

Statement	Meaning in simple English
binary p1,p2,p3,p4	Each supplied pattern is either chosen (1) or not chosen (0).
minimize 7p1+6p2+10p3+5p4	Pattern costs are 7, 6, 10 and 5. Sum the costs of chosen patterns.
p1+p3=1	Task 1 must be covered exactly once, by pattern 1 or 3.
p2+p3=1	Task 2 must be covered exactly once, by pattern 2 or 3.
p1+p4=1	Task 3 must be covered exactly once, by pattern 1 or 4.
p2+p4=1	Task 4 must be covered exactly once, by pattern 2 or 4.

Complete companion file

```
optimizer.exe --crew --file models/crew_patterns.txt --seconds 10
```

Build exact-cover patterns from operational data

Separate pattern generation from pattern selection. Generate only candidates that are individually legal, keep their task membership and cost in a table, and emit one binary variable per candidate. Emit one exact-one equation per required task.

Example generator idea

```
patterns = {
    "p1": (7, ["F1", "F3"]),
    "p2": (6, ["F2", "F4"]),
    "p3": (10, ["F1", "F2"]),
```

```

    "p4": (5, ["F3", "F4"]),
}
lines = ["binary " + ",".join(patterns)]
lines += ["minimize " + "+".join(
    f"{cost}*{p}" for p, (cost, tasks) in patterns.items())]
for task in ["F1", "F2", "F3", "F4"]:
    names = [p for p, (_, tasks) in patterns.items() if task in tasks]
    if not names:
        raise ValueError("No legal pattern for " + task)
    lines.append("+".join(names) + "=1")

```

The complete runnable program is in `languages/patterns.py`. It uses the supplied DLL adapter. Python may read the pattern table from a database or CSV before generating input.

Preserve the meaning of legality If a rule involves two selected patterns together, individual pattern generation alone may not represent it. Either represent that rule in general MILP or use a dedicated service with the required supported constraints. For example, an employee's total workload across several patterns is not automatically checked by a task-only cover.

Retain original identifiers outside the native variable names so selected variables can be converted back into understandable operational plans.

Crew controls, bounds and practical acceptance

Use `--crew-starts` for the initial feasible-seed target and `--crew-pool` for the retained diverse schedule count. `--crew-moves` sets the overall move cap. `--crew-neighborhood` limits how many selected patterns may be released in a local repair, and `--crew-block-nodes` limits that repair work. These are work controls, not promises of improvement.

Optional `--crew-connected` enables connected, stall-adaptive repair; it is off by default. Compare it against the baseline on your own model. `--extend` changes applicable cost/verification arithmetic; it does not make this a complete exact integer optimizer.

```
optimizer.exe --crew --file models/crew_patterns.txt --seconds 20 --seed 38 --crew-connected
```

For native seed trials:

```
optimizer.exe --crew --file model.txt --multseeds --seed_numbers 5 --seed-seconds 7 --seconds 60
--seed 38
```

Report interpretation An independently evaluated bound can quantify the remaining possible improvement. A nonzero gap means optimality is not established. A good incumbent is valuable even if the bound remains weak. Do not interpret a numerical search score as a valid objective bound.

No result Limited search without a schedule does not prove that the exact cover is impossible. Check that every task has legal patterns and that fixed selections do not conflict. Increase work/time or consider a different seed only after checking the formulation.

Do not combine `--crew` with `--linear`, `--sqp` or their dedicated search controls. Read `--crew-help` for accepted settings. This service does not accept native covering inequalities or general binary side constraints.

10. City tours: traveling salesman problems

City tours / TSP: supported file format

The dedicated TSP service finds a closed tour visiting each city once. Input is symmetric TSPLIB TYPE TSP. Use the original coordinates or matrix; do not expand a large tour into a generic binary formulation unless additional requirements truly demand it.

Coordinate metrics EUC_2D uses nearest-integer Euclidean distances; CEIL_2D rounds up-ward; ATT uses TSPLIB pseudo-Euclidean rounding. These costs are not interchangeable with unrounded geometric length.

Required coordinate records include TYPE, DIMENSION, EDGE_WEIGHT_TYPE and NODE_COORD_SECTION. Node labels and coordinates follow. Coordinates must lie within the supported magnitude (up to $1e14$). DIMENSION must be at least three.

Explicit matrices EDGE_WEIGHT_TYPE: EXPLICIT with EDGE_WEIGHT_FORMAT: FULL_MATRIX is supported up to 5000 cities. The matrix must be symmetric and its costs finite, within the supported magnitude. Other matrix layouts and directed ATSP are not supported.

```
optimizer.exe --tsp --file models/city_square.tsp --seconds 5
optimizer.exe --file models/city_square.tsp --seconds 5
```

Recognized TSP files dispatch automatically through --file or the single-file form. Explicit --tsp is the clearest command.

Scope The service has no capacity limits, service times, delivery windows or extra equations. All cities are visited, and the last edge returns to the starting city. Rotating or reversing a symmetric tour can describe the same route and cost.

Heuristic tour quality depends on instance and budget. A target cost or an excellent short-run result is not a universal 1% guarantee. Exact small-tour completion is identified separately.

City tours: explaining a coordinate file

NAME is your instance label. TYPE TSP means visit every city once and return to the start. DIMENSION is the number of cities, not the number of coordinates in a row. EDGE_WEIGHT_TYPE defines how travel lengths are obtained.

Under NODE_COORD_SECTION each row is city_id x y. The identifier labels a city; the numbers are its coordinates. They do not specify a visiting order. Under EUC_2D, distances follow the supported rounded Euclidean convention. ATT uses its different pseudo-Euclidean convention; do not change ATT to EUC_2D merely because both use coordinates.

For EXPLICIT distance input, EDGE_WEIGHT_FORMAT and EDGE_WEIGHT_SECTION specify a supported matrix layout. Each matrix entry is the cost of travel between its corresponding cities. Use that for measured travel costs rather than geographical coordinates when appropriate.

The TSP service has no demands, vehicle capacities, time windows or optional customers. It returns one closed tour containing every city exactly once. Those additional routing requirements belong in CVRP or VRPTW. Standard command: optimizer.exe --tsp --file cities.tsp --seconds 30.

Tsp worked file: every line explained

Visit four city locations once and return to the first city. Collect coordinates and choose the correct travel metric; coordinates are not necessarily driving distances.

Input line	Meaning in simple English
NAME: square	Gives this instance the name square. It is a label, not a numeric parameter.
TYPE: TSP	Selects the TSP input grammar. Keep this consistent with the service flag in your command.
DIMENSION: 4	Declares 4 nodes. For CVRP this includes the depot; for TSP these are all cities to visit.
EDGE_WEIGHT_TYPE: EUC_2D	Chooses EUC_2D travel costs. EUC_2D uses the supported rounded Euclidean distances; EXPLICIT reads the supplied cost matrix.
NODE_COORD_SECTION	Begins rows of node ID, x coordinate and y coordinate. The order is data order, not a route.
1 0 0	Node 1 is at coordinate (0, 0). This row does not prescribe route order.
2 10 0	Node 2 is at coordinate (10, 0). This row does not prescribe route order.
3 10 10	Node 3 is at coordinate (10, 10). This row does not prescribe route order.
4 0 10	Node 4 is at coordinate (0, 10). This row does not prescribe route order.
EOF	Ends this standard-format example. It is a file terminator, not an object, item or decision.

Save and run The complete file is models/city_square.tsp.

```
optimizer.exe --tsp --file models/city_square.tsp --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

TSP example: four-city square

```
NAME: square
TYPE: TSP
DIMENSION: 4
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 0 0
2 10 0
3 10 10
4 0 10
EOF
```

Run:

```
optimizer.exe --tsp --file models/city_square.tsp --seconds 5
```

A perimeter tour 1-2-3-4-1 has cost 40. The opposite direction or a different starting label is equivalent. Every city occurs once before closing the route.

Read the returned tour Check that all input labels are present, none is duplicated, and the closing edge is included in the objective. A list of four cities without return-cost accounting would describe an open path, not the requested model.

Try construction diversity

```
optimizer.exe --tsp --file models/city_square.tsp --tsp-pool 4 --seconds 5
```

Tiny instances may complete exactly before heuristic diversity matters. For larger tours, --tsp-pool ranks several starting tours within the same time allowance. This does not guarantee that the best construction produces the best final local result.

If you scale coordinates, TSPLIB rounding can affect distances nonlinearly at small scales. Re-compute costs using the declared metric rather than unrounded map lengths. For a road network with asymmetric travel costs or missing edges, this symmetric complete-tour service is

not the correct model.

TSP example: an explicit cost matrix

```
NAME: four_costs
TYPE: TSP
DIMENSION: 4
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
0 10 14 10
10 0 10 14
14 10 0 10
10 14 10 0
EOF
```

Run:

```
optimizer.exe --tsp --file models/city_matrix.tsp --seconds 5
```

The perimeter cost is 40. Cross-diagonal routes cost more. This example shows how supplied pairwise costs can replace coordinates while keeping the same symmetric complete-tour semantics.

Each matrix row corresponds to one city in order. Supply all entries, including the diagonal, in FULL_MATRIX order. A triangular layout is not accepted merely because it represents the same mathematical information. Keep costs symmetric.

When a matrix is appropriate Use it for measured symmetric travel costs or other pairwise movement penalties. Coordinate modes can be much more memory-efficient for large instances. FULL_MATRIX is limited to 5000 cities; it is not the input route for a 14,000-city matrix.

Unsupported constraints Do not add service windows, forbidden-city transitions or vehicle assignments to this file and expect the dedicated TSP solver to enforce them. Use a service or general formulation that explicitly supports your requirements.

--tsp-target stops at a requested feasible cost threshold. Meeting it is not an optimality proof. --extend is accepted but does not provide binary128 tour-cost refinement; generic residual tolerance flags are not tour settings.

Matrix worked file: every line explained

Use measured symmetric travel costs rather than coordinates. Supply the full four-by-four matrix, maintaining the same city order for every row and column.

Input line	Meaning in simple English
NAME: four_costs	Gives this instance the name four_costs. It is a label, not a numeric parameter.
TYPE: TSP	Selects the TSP input grammar. Keep this consistent with the service flag in your command.
DIMENSION: 4	Declares 4 nodes. For CVRP this includes the depot; for TSP these are all cities to visit.
EDGE_WEIGHT_TYPE: EXPLICIT	Chooses EXPLICIT travel costs. EUC_2D uses the supported rounded Euclidean distances; EXPLICIT reads the supplied cost matrix.
EDGE_WEIGHT_FORMAT: FULL_MATRIX	FULL_MATRIX means supply every entry of the square distance matrix in row order, including diagonal entries.
EDGE_WEIGHT_SECTION	Begins distance/cost matrix rows; entry at row i and column j is the travel cost between those cities.
0 10 14 10	One row of the four-city cost matrix, in city order 1, 2, 3, 4. Diagonal zero means no travel from a city to itself.
10 0 10 14	One row of the four-city cost matrix, in city order 1, 2, 3, 4. Diagonal zero means no travel from a city to itself.
14 10 0 10	One row of the four-city cost matrix, in city order 1, 2, 3, 4. Diagonal zero means no travel from a city to itself.
10 14 10 0	One row of the four-city cost matrix, in city order 1, 2, 3, 4. Diagonal zero means no travel from a city to itself.

Input line	Meaning in simple English
EOF	Ends this standard-format example. It is a file terminator, not an object, item or decision.

Save and run The complete file is `models/city_matrix.tsp`.

```
optimizer.exe --tsp --file models/city_matrix.tsp --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

TSP options and quality tuning

Current automatic-policy distinction: establish a baseline without manual tuning flags. Supplying an actual `--tsp-*` tuning control, except `--tsp-target`, selects the legacy manual search. The parameter discussion below describes that manual path; it does not guarantee improvement over today's automatic defaults. Chapter 37 and `--tsp-advanced-help` explain the distinction.

The default time budget is 30 seconds. `--tsp-pool` can rank up to 1000 construction starts, limited by the city count; zero disables the pool. `--tsp-elite` selects how many top-ranked starts receive additional improvement. Pool construction and all improvement share the configured time.

Practical command

```
optimizer.exe --tsp --file large.tsp --tsp-pool 50 --tsp-elite 3 --seconds 30 --seed 38
```

This is a tuning example, not a claim of guaranteed superiority. More starts can leave less time to improve each one.

Other controls `--tsp-starts` controls non-pool construction starts. `--tsp-neighbors` sets candidate breadth. `--tsp-iterations`, `--tsp-stall` and `--tsp-passes` cap improvement work. `--tsp-3opt` sets a supported breadth or zero disables that option. `--tsp-depth` controls compound-move depth where applicable. `--tsp-patch` enables bounded fragment improvement. `--tsp-exact` sets the exact-small-instance threshold. Full ranges/defaults appear in the city-tour option reference.

Large versus small runs Large-tour defaults are driven by time rather than an implicit short stall cap. With `--seconds 0`, finite default work caps still apply. An explicit small iteration/stall cap can stop a run before its larger time allowance is used.

Acceptance Compare verified tour cost against a known benchmark, if one exists, under the same metric. An external known optimum is useful for measuring a gap but is not automatically known or certified by this run. Preserve the tour itself, not just the scalar cost, so it can be checked and used.

11. Capacitated vehicle routing

CVRP: capacity-routing format

CVRP serves every customer once from one depot, using identical vehicle capacities. Deliveries are unsplit; every route begins and ends at the depot. Objective is total travel cost.

Input is TYPE CVRP with DIMENSION including the depot, CAPACITY, a supported EDGE_WEIGHT_TYPE, coordinates or a symmetric FULL_MATRIX, DEMAND_SECTION for every node, and DEPOT_SECTION containing one depot label followed by -1.

Fleet semantics --vehicles K is the maximum fleet available, not a requirement to use all K vehicles. A VEHICLES header is supported; CLI overrides it. If neither is supplied, up to one vehicle per customer is allowed. A number embedded in the instance filename or COMMENT is not a fleet declaration.

Quantities Capacity and demand are nonnegative integers within supported limits, up to 1e12; capacity must be usable and depot demand zero. Scale fractional quantities into consistent integer units, such as kilograms or tenths of a unit, before writing this format.

Metrics and sizes EUC_2D, CEIL_2D, ATT and symmetric EXPLICIT/FULL_MATRIX are supported. Coordinate dimensions run from 2 to 100001 including the depot; FULL_MATRIX is limited to 5000 nodes. Labels are preserved in output.

Not represented here Time windows, pickups, split deliveries, multiple depots, asymmetric costs, heterogeneous fleets, route-duration limits and vehicle fixed costs. VRPTW covers several of those additional delivery requirements. Unknown constraint sections are rejected rather than ignored.

```
optimizer.exe --cvrp --file models/deliveries.vrp --vehicles 2 --seconds 10
```

Capacity routing: understanding the supplied data

In a CVRP file, DIMENSION counts all nodes including the depot. CAPACITY is the load each supported identical vehicle may carry. NODE_COORD_SECTION gives node_id x y; DEMAND_SECTION gives node_id required_load. Use matching IDs in both sections.

DEPOT_SECTION identifies the depot, followed by the format terminator. The depot has zero customer demand. Every non-depot customer is mandatory and must be served once; a route starts and finishes at the depot. Use the supported vehicle-limit option when a hard route count applies; do not infer a fleet limit from DIMENSION.

Keep every demand in the same load unit as CAPACITY: kilograms, pallets or another consistent measure. Distance units are independent of load units. A customer's demand exceeding vehicle capacity is not repaired by placing the customer on two routes; split deliveries are not represented here.

EDGE_WEIGHT_TYPE sets the travel metric, as in the tour service. Customer coordinates are not delivery windows or service durations. Use VRPTW for those fields. Standard command: optimizer.exe --cvrp --file deliveries.vrp --seconds 30.

Cvrp worked file: every line explained

Deliver four orders, each requiring two load units, using at most two vehicles of capacity four. Collect depot, customer locations, demand and fleet capacity before writing the file.

Input line	Meaning in simple English
NAME: deliveries	Gives this instance the name deliveries. It is a label, not a numeric parameter.
TYPE: CVRP	Selects the CVRP input grammar. Keep this consistent with the service flag in your command.
DIMENSION: 5	Declares 5 nodes. For CVRP this includes the depot; for TSP these are all cities to visit.
CAPACITY: 4	Each vehicle can hold at most 4 load/size units. This is not the number of vehicles/bins.
VEHICLES: 2	At most 2 vehicles/routes are available. Each route begins and ends at the depot.
EDGE_WEIGHT_TYPE: EUC_2D	Chooses EUC_2D travel costs. EUC_2D uses the supported rounded Euclidean distances; EXPLICIT reads the supplied cost matrix.
NODE_COORD_SECTION	Begins rows of node ID, x coordinate and y coordinate. The order is data order, not a route.
1 0 0	Node 1 is at coordinate (0, 0). This row does not prescribe route order.
2 1 0	Node 2 is at coordinate (1, 0). This row does not prescribe route order.
3 2 0	Node 3 is at coordinate (2, 0). This row does not prescribe route order.
4 -1 0	Node 4 is at coordinate (-1, 0). This row does not prescribe route order.
5 -2 0	Node 5 is at coordinate (-2, 0). This row does not prescribe route order.
DEMAND_SECTION	Begins rows of node ID and required load. Load uses the same unit as vehicle capacity.
1 0	Node 1 requires load 0. The depot row has zero load.
2 2	Node 2 requires load 2. The depot row has zero load.
3 2	Node 3 requires load 2. The depot row has zero load.
4 2	Node 4 requires load 2. The depot row has zero load.
5 2	Node 5 requires load 2. The depot row has zero load.
DEPOT_SECTION	Begins the depot identifier list; -1 ends that section.
1	Node 1 is the depot. Routes must return to it.
-1	Ends the depot list; it is not a negative node identifier.
EOF	Ends this standard-format example. It is a file terminator, not an object, item or decision.

Save and run The complete file is models/deliveries.vrp.

```
optimizer.exe --cvrp --file models/deliveries.vrp --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

CVRP example: four deliveries

```
NAME: deliveries
TYPE: CVRP
DIMENSION: 5
CAPACITY: 4
VEHICLES: 2
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 0 0
2 1 0
3 2 0
4 -1 0
5 -2 0
DEMAND_SECTION
1 0
2 2
3 2
4 2
5 2
DEPOT_SECTION
1
-1
EOF
```

Run:

```
optimizer.exe --cvrp --file models/deliveries.vrp --seconds 5
```

A feasible two-route plan is 1-2-3-1 and 1-4-5-1. Each carries load 4. The total rounded travel cost is 8. All customers are visited exactly once; label 1 is the depot.

A one-vehicle limit would violate total demand 8 versus capacity 4. Here that necessary capacity check establishes the problem with one vehicle cannot be served. Other failed heuristic searches do not generally prove infeasibility.

--vehicles is a maximum. If one vehicle can serve an entire different instance, a larger allowed fleet does not force unnecessary routes. Service times and time windows cannot be added to this VRPLIB sample as arbitrary extra rows; use the supported VRPTW records instead.

CVRP options, feasibility and practical use

Use the fleet cap and capacity data first; no search setting can overcome a genuine capacity contradiction. Then allow enough time for construction and improvement.

```
optimizer.exe --cvrp --file deliveries.vrp --vehicles 12 --seconds 30 --seed 38 --cvrp-starts 50
```

--cvrp-starts controls construction starts. --cvrp-neighbors sets candidate breadth. --cvrp-iterations caps improvement attempts; without a finite clock, a finite default still applies. --cvrp-ruin specifies the number of customers released in an improvement attempt; zero selects the adaptive default. --cvrp-target is a feasible cost threshold, not a proof request.

Check the complete plan Every customer must occur once, every route must return to the declared depot, each load must fit capacity and the number of routes must not exceed the fleet cap. Recompute the objective using the declared rounded metric.

Quality Heuristic outcomes can differ by seed, runtime and instance. A valid route may be useful even without a bound. More seconds may improve it, but no universal closeness-to-optimum guarantee is given. Native multi-seed provides diversification within one total allowance.

Precision --extend is accepted for compatibility but route geometry/cost search remains on its documented ordinary/rounded numerical representation. General nonlinear tolerance flags are not routing tuning controls.

Operational choice If drivers must return by a deadline, customers have delivery windows or vehicles have different capacities/costs, use VRPTW and provide those rules. Do not accept an unconstrained CVRP result as if the omitted rules had been checked.

12. Vehicle routing with time windows

VRPTW: records and field meanings

Use `--vrptw` with TYPE VRPTW. This service supports delivery windows, service times and a finite named heterogeneous fleet from one depot.

```
TYPE VRPTW
DEPOT depot X Y READY DUE
CUSTOMER c1 X Y DEMAND READY DUE SERVICE
VEHICLE van CAPACITY START END MAX_DURATION FIXED_COST DISTANCE_COST SPEED
ALLOW c1 van
MATRIX depot c1 TRAVEL_TIME DISTANCE
```

DEPOT defines geometry and its time window. CUSTOMER defines demand and the interval in which service must start. SERVICE is the duration spent there. VEHICLE defines capacity, availability, maximum full route duration, use cost, per-distance cost and speed.

Time interpretation Waiting is allowed. Service start, rather than service end, must be within the customer's window. Route duration includes travel, waiting and service. A feasible later depot departure can reduce unnecessary waiting, but still must satisfy all availability limits.

Eligibility and travel data ALLOW creates an optional vehicle whitelist for a customer. Without it, eligible vehicle selection follows the general fleet model. Default geometry is Euclidean, with travel time distance/SPEED. If any MATRIX travel data is supplied, provide every off-diagonal ordered pair and use speed 1.

Scope and limits Delivery only; no pickups, split deliveries, breaks, traffic model or multiple depots. Numeric fields use double and `--tol` controls absolute VRPTW feasibility checks. Up to 2000 customers and 500 vehicles are supported. All costs are nonnegative. Operational `--extend` is rejected.

Use one consistent distance/time unit throughout the data. Coordinates in kilometers with speed kilometers/hour require windows and service durations in hours.

Routing with windows: reading a stop and vehicle

DEPOT depot X Y READY DUE gives a named depot, coordinates and its opening/closing times. CUSTOMER c1 X Y DEMAND READY DUE SERVICE gives a mandatory stop: position, required load, earliest and latest permitted service start, and time spent serving it.

For CUSTOMER c1 2 0 3 8 12 0.5, deliver 3 load units at coordinate (2,0); start service between time 8 and 12 inclusive; spend 0.5 time unit there. Early arrival can require waiting. DUE is a service-start deadline, not a requirement to finish service by that time.

VEHICLE van CAPACITY START END MAX_DURATION FIXED_COST DISTANCE_COST SPEED describes its load limit, working interval, maximum route duration, charge if used, cost per distance unit and travel speed. Route duration includes travel, service and waiting. Return must respect the vehicle and depot closing limits.

ALLOW c1 van restricts the stop to that eligible vehicle; repeated supported ALLOW records describe eligibility. MATRIX from to TRAVEL_TIME DISTANCE supplies directed travel data. Provide the complete required off-diagonal matrix and use the supported speed convention. Keep time, distance, load and cost units individually consistent. Standard command: `--vrptw --file PATH`.

Vrptw worked file: every line explained

Deliver four orders with different permitted service windows and two available vans. Record loading capacity, working times, service durations, distance charges and eligibility separately.

Input line	Meaning in simple English
TYPE VRPTW	Selects the VRPTW input grammar. Keep this consistent with the service flag in your command.
DEPOT depot 0 0 0 100	Depot depot is at (0, 0), open from time 0 to 100. Routes start and return there.
CUSTOMER A 1 0 2 0 20 1	Stop A is at (1, 0), needs load 2, allows service START from 0 to 20, and requires service duration 1.
CUSTOMER B 2 0 2 0 20 1	Stop B is at (2, 0), needs load 2, allows service START from 0 to 20, and requires service duration 1.
CUSTOMER C -1 0 2 20 50 1	Stop C is at (-1, 0), needs load 2, allows service START from 20 to 50, and requires service duration 1.
CUSTOMER D -2 0 2 20 50 1	Stop D is at (-2, 0), needs load 2, allows service START from 20 to 50, and requires service duration 1.
VEHICLE van1 4 0 100 30 5 1 1	Vehicle van1: capacity 4; start/end availability 0/100; maximum route duration 30; used-vehicle charge 5; cost per distance 1; speed 1.
VEHICLE van2 4 0 100 30 5 1 1	Vehicle van2: capacity 4; start/end availability 0/100; maximum route duration 30; used-vehicle charge 5; cost per distance 1; speed 1.
ALLOW A van1	A may use only these listed vehicles: van1. This is eligibility, not a cost.
ALLOW B van1	B may use only these listed vehicles: van1. This is eligibility, not a cost.

Save and run The complete file is models/vrptw.txt.

```
optimizer.exe --vrptw --file models/vrptw.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

VRPTW example: delivery windows and eligibility

```
TYPE VRPTW
DEPOT depot 0 0 0 100
CUSTOMER A 1 0 2 0 20 1
CUSTOMER B 2 0 2 0 20 1
CUSTOMER C -1 0 2 20 50 1
CUSTOMER D -2 0 2 20 50 1
VEHICLE van1 4 0 100 30 5 1 1
VEHICLE van2 4 0 100 30 5 1 1
ALLOW A van1
ALLOW B van1
```

Run:

```
optimizer.exe --vrptw --file models/vrptw.txt --seconds 10
```

Customers A and B are restricted to van1 and have early windows. C and D have later windows. Each vehicle has capacity 4, so two demand-2 customers fit per route. Costs include a fixed 5 for each used van plus traveled distance.

A plan using A/B on van1 and C/D on van2 has total geometric distance 8 and fixed cost 10, for objective 18. Waiting or delayed departure may be needed to meet the later window. Read the returned stop times, not merely the route labels.

A vehicle's maximum duration is distinct from its latest permitted end time. Both matter. Increasing the cost budget does not relax hard windows or capacity limits.

VRPTW matrix travel and acceptance checks

```

TYPE VRPTW
DEPOT D 0 0 0 20
CUSTOMER A 0 0 1 0 10 1
CUSTOMER B 0 0 1 0 10 1
VEHICLE van 2 0 20 20 0 1 1
MATRIX D A 2 4
MATRIX A D 2 4
MATRIX D B 3 6
MATRIX B D 3 6
MATRIX A B 1 2
MATRIX B A 1 2

```

Run:

```
optimizer.exe --vrptw --file models/vrptw_matrix.txt --seconds 5
```

The geometry is replaced by supplied directed travel-time/distance pairs. All six off-diagonal ordered pairs for three locations are present. Vehicle speed is 1 as required for this mode. A complete route D-A-B-D has travel time 6 and distance 12, plus two service-time units; it is feasible within the stated limits.

Why separate travel time and distance? A road segment can have a short distance but a long travel time. Distance drives the distance-cost term, while travel time affects schedule feasibility. Do not interchange them.

Before accepting a route Check customer coverage, eligibility, route loads, service starts, vehicle start/end availability and full duration. Include waiting and service, and respect depot limits. The returned candidate is checked against these represented rules.

--service-starts and --service-iterations are the operational search controls, not --cvrp-starts. A time-limited result is heuristic unless the report establishes otherwise. No returned candidate under a search limit does not prove that all window-feasible assignments are impossible.

13. Cutting stock: one stock length

Homogeneous cutting stock: format and conventions

Use --cutting-stock for one unlimited stock length, minimizing the number of stock bars or rolls. All lengths share one unit and accept fixed decimals up to six places.

```
TYPE: CUTTING_STOCK
STOCK_LENGTH: 110
KERF: 0
TRIM: 0
DEMAND_MODE: EXACT
ITEM_SECTION
A 20 48
B 45 35
C 50 24
D 55 10
EOF
```

Each item row is a unique identifier, finished length and positive integer demand. NAME and COMMENT headers are optional; C/C++ comments are supported. MAX_STOCK: N can impose a positive hard availability limit.

Loss conventions TRIM is total reserved length per used bar, combining both ends. KERF is lost length between adjacent pieces. A bar containing n pieces uses TRIM + sum(piece lengths) + (n-1)*KERF. There is no extra automatic end-cut allowance. Adapt TRIM if your workshop uses a different convention.

Demand EXACT is the default and produces exactly the requested quantities. AT_LEAST permits surplus production. Do not assume surplus is free in a real business problem: this service's objective counts stock bars, not storage or disposal costs.

Scope Up to 200 item types and one million requested pieces, with supported lengths up to one million units. No multiple stock prices, defects, setup costs, knife limits or reusable remnant inventory. Use the multistock service for multiple available stock types.

A smaller restricted pattern set is not automatically a global optimum. Read the returned bound and optimality statement.

Cut worked file: every line explained

Cut two length-6 pieces and two length-4 pieces from identical length-10 bars. Collect finished dimensions and quantities, then measure actual trim and saw loss.

Input line	Meaning in simple English
TYPE: CUTTING_STOCK	Selects the CUTTING_STOCK input grammar. Keep this consistent with the service flag in your command.
STOCK_LENGTH: 10	Each stock bar has nominal length 10, in your chosen common length unit.
KERF: 0	Reserve 0 length units between neighboring pieces. With n pieces in 1D, kerf loss is (n-1) times this value.
TRIM: 0	Reserve 0 total length units per used bar, combining both ends. This is the 1D convention; PACKING2D uses each-edge trim.
DEMAND_MODE: EXACT	EXACT means produce exactly each stated item count; AT_LEAST instead permits surplus when supported.
ITEM_SECTION	Begins rows of item identifier, finished piece length and exact required quantity.
A 6 2	Finished item A: length 6, required quantity 2. These are pieces to produce, not stock bars.
B 4 2	Finished item B: length 4, required quantity 2. These are pieces to produce, not stock bars.
EOF	Ends this standard-format example. It is a file terminator, not an object, item or decision.

Save and run The complete file is models/cutting_simple.cstock.

```
optimizer.exe --cutting-stock --file models/cutting_simple.cstock --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

Cutting examples: complementary pieces and saw loss

```
TYPE: CUTTING_STOCK
STOCK_LENGTH: 10
KERF: 0
TRIM: 0
DEMAND_MODE: EXACT
ITEM_SECTION
A 6 2
B 4 2
EOF
```

Run:

```
optimizer.exe --cutting-stock --file models/cutting_simple.cstock --seconds 5
```

Two stock bars suffice: each produces one A of length 6 and one B of length 4. The total requested length is 20, so at least two length-10 bars are necessary. This matches the plan and proves the bar count for this supported example.

Second example models/cutting_kerf.cstock uses length-12 stock, kerf 0.1, trim 0.2, exact demand A=3 repeated seven times and B=2 repeated six times, with at most four bars.

```
optimizer.exe --cutting-stock --file models/cutting_kerf.cstock --seconds 10
```

One feasible three-bar arrangement is 3A+B, 3A+B and A+4B. Used lengths are 11.5, 11.5 and 11.6 under the stated convention. Total finished length is 33, so two bars cannot suffice; three is the minimum bar count.

Changing kerf or trim changes feasibility even when every finished piece length is unchanged. Always use the workshop's actual allowances.

Read cutting patterns and tune the budget

A returned Pattern record supplies its repeat count, item multiplicities, used length and leftover. Multiply each item multiplicity by the repeats and sum across patterns to verify the required quantities.

Example A pattern with repeats=12 and pieces A:2 B:1 produces 24 A and 12 B. It is not one physical bar with 36 pieces. The repeats represent identical bars or rolls.

Waste and count Leftover is stock length minus modeled used length. The used length includes trim and between-piece kerf. Minimizing bar count is not an independently specified minimum-waste-cost objective. Among plans with the same count, the economically preferred plan may depend on costs not represented here.

Controls --cut-starts controls diversified initial plans. --cut-iterations limits additional pattern search. --cut-patterns caps retained patterns and must be at least the item-type count. --cut-nodes limits packing/integer completion work; zero disables those improvements. --cut-enumerate limits supported completion patterns. --cut-pricing-cells bounds the normalized size work. Exact ranges/defaults are listed on the corresponding option reference.

--extend supports extended numerical master/pricing operations, while the original length, demand, kerf and trim checks remain in exact scaled units. Tolerance is not a substitute for cutting allowance.

Optimality and availability A valid bound matching a feasible integer bar count establishes count optimality for the supported model. A restricted-master result alone does not. If a proved lower bound exceeds MAX_STOCK, the supply limit is infeasible; limited failure to find a plan meeting it is weaker and must be reported as such.

More seconds may not help if another cap has been reached. Check the termination message before increasing time alone.

14. Cutting stock: multiple stock types

Multiple stock lengths, prices and inventory

```
TYPE CUTTING_STOCK_MULTI
OBJECTIVE MIN_COST
KERF 0.1
TRIM 0.2
STOCK long 12 3 9
STOCK short 8 UNLIMITED 6
ITEM A 3 7
ITEM B 2 6
```

Run:

```
optimizer.exe --cutting-stock-multi --file models/cutting_multi.txt --seconds 20
```

STOCK fields are identifier, length, available quantity or UNLIMITED, and positive cost per used bar. ITEM fields are identifier, finished length and exact demand. OBJECTIVE MIN_COST chooses total purchase/use cost; MIN_BARS counts used bars regardless of type.

KERF and total TRIM use the same between-piece convention as homogeneous cutting. The example has three long bars available, unlimited shorter bars, and demand for seven A plus six B. Inventory limits are hard.

Different objectives can yield different plans The cheapest solution may use more shorter bars, while the fewest-bar solution may buy expensive long ones. Choose the objective matching the business decision. Costs are per used stock bar, not per finished piece or per unit length.

Scope Exact demand only; up to 100 stock types, 200 item types and one million demanded pieces. No defects, setup, reusable remnants or sequencing rules. A result optimal only within a restricted pattern set is not a global proof. --extend is rejected for this operational service.

Use --service-starts and --service-iterations with --seconds and --seed. Independently valid reported bounds can support a gap or a supported certificate; failed inventory-limited search alone does not prove infeasibility.

Multiple stock lengths: understanding a purchase record

Read STOCK long 12 3 9 as: “There is a stock type called long; every bar is 12 units long; at most three bars of this type are available; each used bar costs 9 money units.”

Field	Meaning
STOCK	Declares an available stock type.
long	Your name for this type; it is not a special keyword.
12	Nominal length of each bar of this type.
3	Available number of bars. It is an upper limit, not mandatory use.
9	Cost per bar used, in the same currency for all stock types.

STOCK short 8 UNLIMITED 6 means length-8 bars, with no explicit quantity limit, costing 6 each. ITEM A 3 7 means exactly seven finished length-3 pieces named A. ITEM B 2 6 means exactly six length-2 pieces named B.

For OBJECTIVE MIN_COST, minimize the sum of used-bar costs. For OBJECTIVE MIN_BARS, minimize the number of used bars. These objectives need not choose the same purchase plan.

With TRIM 0.2 and KERF 0.1, a bar containing n pieces must accommodate $0.2 + \text{their total finished length} + (n-1)*0.1$. Trim is combined reserved length for both ends. Four length-3 pieces require $12+0.2+0.3=12.5$ and therefore do not fit in a length-12 bar. Three such pieces

require 9.4 and do fit.

Use the same length unit for stock, items, trim and kerf. Do not multiply the cost by bar length: the record already gives the cost of one complete used bar.

Multi worked file: every line explained

Buy/use length-12 and length-8 bars to fulfill seven length-3 pieces and six length-2 pieces. Collect available quantities and cost per whole bar; minimize purchase/use cost.

Input line	Meaning in simple English
TYPE CUTTING_STOCK_MULTI	Selects the CUTTING_STOCK_MULTI input grammar. Keep this consistent with the service flag in your command.
OBJECTIVE MIN_COST	Minimize total cost of stock bars used.
KERF 0.1	Reserve 0.1 length units between neighboring pieces. With n pieces in 1D, kerf loss is (n-1) times this value.
TRIM 0.2	Reserve 0.2 total length units per used bar, combining both ends. This is the 1D convention; PACKING2D uses each-edge trim.
STOCK long 12 3 9	Stock type long: bar length 12; availability at most 3 bars; cost 9 per used bar.
STOCK short 8 UNLIMITED 6	Stock type short: bar length 8; availability unlimited; cost 6 per used bar.
ITEM A 3 7	Item A requires exactly 7 finished pieces of length 3. These are demand, not stock inventory.
ITEM B 2 6	Item B requires exactly 6 finished pieces of length 2. These are demand, not stock inventory.

Save and run The complete file is models/cutting_multi.txt.

```
optimizer.exe --cutting-stock-multi --file models/cutting_multi.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

15. One-dimensional bin packing

One-dimensional bin packing

```
TYPE BIN_PACKING
CAPACITY 10
ITEM A 6 2
ITEM B 4 2
ITEM C 2 5
```

Run:

```
optimizer.exe --bin-packing --file models/bin_packing.txt --seconds 10
```

CAPACITY is the identical bin capacity. ITEM fields are identifier, size and exact quantity. Objective is the number of bins. There is no kerf or trim in this bin-packing service.

The example contains two size-6 A, two size-4 B and five size-2 C, total size 30. Three capacity-10 bins are sufficient: two contain A+B, and the third contains five C. The total-size lower bound is also three, establishing the minimum for this example.

Suitable applications Identical box/volume allocations with a single scalar capacity, batching by weight or similar homogeneous capacity grouping. If both weight and volume limits matter, this one-dimensional service does not silently enforce the second limit. Use a supported general formulation.

Output and options Read the bin patterns and their repeats, verifying exact quantities and capacity. --service-starts and --service-iterations are the accepted operational controls. Do not send --cut-patterns or --cut-nodes to this wrapper; homogeneous cutting has a separate interface for those settings.

Units and scope Use the same size unit for every item and the bin capacity. Do not express size as an arbitrary Python function. Fixed decimal fields support up to six places. There is no heterogeneous bin price/supply model, 2D geometry or 3D loading in this format.

Optional supported exact-mode attempt

--bin-exact enables the supported small-instance branch-and-price path; see Chapter 37 for the ≤ 256 -copy and normalized-capacity limits. Use --service-iterations for its work allowance. A stopped exact-mode attempt is not a completed optimality proof, and larger inputs retain the heuristic path.

One-dimensional bin packing: reading the records

TYPE BIN_PACKING selects packing into identical capacity-limited bins. CAPACITY 10 says each bin can hold at most 10 size units. ITEM box 4 3 requires three separate objects called box, each size 4. The final field is a copy count, not a weight or price.

A bin can contain two such objects, using 8 units; three would use 12 and exceed capacity. The objective is to minimize bins used while placing every requested copy. Sizes must be positive and all share the capacity unit.

This service does not model saw kerf, border trim, shipping distance, multiple resource capacities or rectangular geometry. Use cutting stock for bar cutting, PACKING2D for rectangles or RESOURCE_PLACEMENT for multi-resource assignment. Standard command: optimizer.exe --bin-packing --file bins.txt --seconds 10.

Bin worked file: every line explained

Pack all requested objects into identical capacity-10 bins. Record indivisible object sizes and counts; the decision is which bin receives each copy.

Input line	Meaning in simple English
TYPE BIN_PACKING	Selects the BIN_PACKING input grammar. Keep this consistent with the service flag in your command.
CAPACITY 10	Each bin can hold at most 10 load/size units. This is not the number of vehicles/bins.
ITEM A 6 2	Object type A has size 6; place exactly 2 separate copies. Each copy consumes this size.
ITEM B 4 2	Object type B has size 4; place exactly 2 separate copies. Each copy consumes this size.
ITEM C 2 5	Object type C has size 2; place exactly 5 separate copies. Each copy consumes this size.

Save and run The complete file is models/bin_packing.txt.

```
optimizer.exe --bin-packing --file models/bin_packing.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

16. Rectangular packing and cutting

Rectangular packing: constructing the sheet model

Use --packing2d to place finished rectangular pieces on identical stock sheets. The built-in objective is to use as few sheets as possible. This is two-dimensional cutting/packing, not three-dimensional loading.

```
TYPE PACKING2D
SHEET 10 10
TRIM 0.2
KERF 0.1
METHOD GUILLOTINE
MAX_SHEETS 4
ITEM A 5 4 3 1
ITEM B 3 2 6 1
ITEM C 2 2 5 0
```

Save this as panels.txt and run:

```
optimizer.exe --packing2d --file panels.txt --seconds 10
```

Record	Meaning in plain English
TYPE PACKING2D	This file describes rectangular pieces to pack/cut from sheets.
SHEET 10 10	Each unused stock sheet is 10 units wide and 10 units high.
TRIM 0.2	Keep a 0.2-unit border unused on each of all four edges.
KERF 0.1	Allow 0.1 unit of material lost between pieces at a supported cut.
METHOD GUILLOTINE	Require the supported sequence of straight cuts across a current rectangular region.
MAX_SHEETS 4	At most four stock sheets are available. This does not force using all four.

Choose a common length unit, such as centimetres or metres, and use it for every dimension, trim and kerf. The program does not convert units automatically. Counts are numbers of pieces, not lengths. The item rows are explained on the following pages.

Rectangular packing: reading every ITEM field

The item syntax is:

```
ITEM identifier width height quantity rotation
```

Read ITEM A 5 4 3 1 as: “Make three finished pieces called A, each 5 units wide and 4 units high; rotating a piece by 90 degrees is permitted.”

Field in ITEM A 5 4 3 1	Meaning
ITEM	Introduces one type of required finished rectangle.
A	Your identifier for this piece type, used in results.
5	Finished width of each piece before any permitted rotation.
4	Finished height of each piece before any permitted rotation.
3	Exactly three copies of A are required.
1	Rotation by 90 degrees is allowed: 5-by-4 or 4-by-5.

ITEM B 3 2 6 1 means six 3-by-2 pieces, which may also be placed as 2-by-3. ITEM C 2 2 5 0 means five 2-by-2 pieces whose specified orientation is fixed. C is square, so a rotation would not change its geometric dimensions, but 0 still records the orientation rule.

Choosing the rotation field Use 0 for grain direction, printed labels, surface finish or another orientation requirement. Use 1 only if interchanging width and height is acceptable. Rotation is not an arbitrary-angle option. It does not allow resizing or altering piece quantities.

The example requires $3+6+5=14$ finished pieces. Their total finished area is $354 + 632 + 522 = 116$ square units. A returned plan must contain every required copy with the permitted dimensions, not merely some subset that fits.

Rectangular packing: trim, kerf and achievable sheet count

TRIM 0.2 reserves 0.2 unit on the left, right, top and bottom of each sheet. For SHEET 10 10, usable width and height are both $10-2 \times 0.2=9.6$. Usable area is $9.6 \times 9.6=92.16$ square units.

It is not a 9.8-by-9.8 usable sheet. Nor is the trim a 0.2 total allowance shared between two sides. In one-dimensional cutting stock, by contrast, TRIM is the total reserved length per bar. Always use the convention for the selected service.

Kerf is a cutting allowance KERF 0.1 represents material removed between neighboring pieces in the supported cutting partition. Finished A dimensions remain 5 by 4. Do not add kerf to every item's width and height yourself and also set KERF: that would double-count the allowance. If the saw requires extra end allowances, account for those consistently in the chosen trim convention.

What the sheet limit means MAX_SHEETS 4 permits a plan using one, two, three or four sheets. It is a hard upper supply limit, not a desired output count. Omit it when you do not want this explicit supply cap.

The example's area alone requires at least $\text{ceil}(116/92.16)=2$ sheets. Area is a lower bound, not a geometric packing proof: shapes, permitted rotation, kerf and guillotine cuts can require more. If a verified two-sheet plan is returned and the reported bound is two, the sheet count can be certified. Otherwise read the reported status rather than assuming optimality.

A failed limited search with MAX_SHEETS 4 does not, by itself, prove no four-sheet plan exists. Check data and limits before changing a real inventory restriction.

The companion example was tested for this edition: it returned two sheets, zero reported violation and a matching lower bound of two. Different valid placements can have the same sheet count.

Rectangular packing: methods and reading a plan

GUILLOTINE describes straight cuts that go across the current rectangular region being cut. After a cut, smaller rectangular regions can be cut again. Choose this method when the workshop requires that cut structure.

MAXRECTS describes an arrangement without a guillotine-cut guarantee. This method requires KERF 0. It is useful when placement matters and the supported no-kerf arrangement is appropriate; do not assume its layout supplies a valid saw sequence.

```
TYPE PACKING2D
SHEET 10 10
TRIM 0
KERF 0
METHOD MAXRECTS
ITEM A 6 4 2 1
ITEM B 4 4 2 0
```

```
optimizer.exe --packing2d --file models/packing2d_maxrects.txt --seconds 5
```

Result fields A placement identifies the piece, copy, sheet, x/y position and placed width/height. Coordinates are measured from the stock sheet origin; the usable rectangle begins at (TRIM, TRIM). A rotated A may be reported with width 4 and height 5. Sheet numbers separate pieces placed on different sheets.

Cut-region records describe the supported guillotine partition. Keep them with the placement report when planning production. Check that all 14 copies in the main example are present, no pieces overlap, all borders are respected and every rotation is permitted.

This service supports up to 10000 pieces and fixed decimal dimensions with up to six decimal places. It does not model sheet defects, arbitrary shapes, arbitrary angles, remnant inventory or 3D stacking. Use --seconds, --seed, --service-starts, --service-iterations and --progress as documented. --extend is not supported for this operational service.

17. Production and job scheduling

Production/job scheduling: record format

Use --schedule for nonpreemptive operations on unary machines, with alternative machines, precedence, calendars and supported cumulative resources.

```
TYPE SCHEDULING
OBJECTIVE MAKESPAN
MACHINE M1
RESOURCE crane 2
JOB job1 RELEASE DUE INTEGER_WEIGHT
OP op1 job1 ORDER
OPTION op1 M1 DURATION
USE op1 crane QUANTITY
PRECEDENCE previous_op next_op
DOWNTIME M1 START END
DEADLINE job1 HARD_TIME
```

OBJECTIVE can be MAKESPAN, TOTAL_TARDINESS or TOTAL_COMPLETION. A job's operations follow ascending unique ORDER; explicit PRECEDENCE adds cross-operation rules. Multiple OPTION rows give eligible machines and their different processing durations.

DUE is a soft timing target contributing to the appropriate tardiness objective; DEADLINE is a hard completion limit. RELEASE limits the start of a job. INTEGER_WEIGHT is the nonnegative/valid integral weight under the supported model; use a meaningful positive weight for a penalized job.

Time and resources Times are absolute and nonnegative. Intervals are half-open, so an operation ending at time 3 can be followed by another starting at 3. DOWNTIME removes machine availability. USE quantities may not exceed the named resource capacity across simultaneous operations.

Not inferred Working calendars, setup/changeover times, preemption, batching and sequence-dependent setup are not invented. Provide supported downtime explicitly or use a general formulation for unsupported rules.

Scheduling limits are 1000 machines, 10000 jobs and 10000 operations. Fixed time/resource data accept up to six decimal places. Read the operation plan and objective, not just a job order; alternative machines and calendars affect feasible start/end times.

Scheduling: translating a work order into records

MACHINE M1 declares one machine that processes at most one operation at a time. RESOURCE crane 2 declares a shared resource with capacity 2.

JOB job1 0 12 3 means release time 0, due time 12 and integer weight 3. Due time is a soft target when tardiness is penalized. DEADLINE job1 15 would separately impose a hard latest completion time.

OP cut1 job1 1 creates operation cut1 in job1 with order number 1. OPTION cut1 M1 4 allows it on M1 for duration 4. A second OPTION with another machine gives a genuine alternative; it is not an additional operation. USE cut1 crane 1 requires one crane-capacity unit while cut1 runs. PRECEDENCE cut1 finish1 requires cut1 to finish before finish1 starts. Job order already supplies its within-job sequence. DOWNTIME M1 5 7 makes M1 unavailable during [5,7). An operation ending at 5 can precede that closure; one starting at 7 can follow it.

Choose OBJECTIVE MAKESPAN for the time when all work finishes, TOTAL_COMPLETION for summed job completion times, or TOTAL_TARDINESS for weighted lateness. Use common time units throughout. Standard command: --schedule --file PATH.

Schedule worked file: every line explained

Complete two two-operation jobs on two machines while sharing one crane. Collect operation order, allowed machines, durations, resource use and downtime.

Input line	Meaning in simple English
TYPE SCHEDULING	Selects the SCHEDULING input grammar. Keep this consistent with the service flag in your command.
OBJECTIVE MAKESPAN	Minimize the completion time of all jobs: the latest job finish.
MACHINE M1	Declares machine M1. Only one operation can occupy it at a time.
MACHINE M2	Declares machine M2. Only one operation can occupy it at a time.
RESOURCE crane 1	Declares shared resource crane with capacity 1. Simultaneous operations must share that capacity.
JOB J1 0 8 1	Job J1: release time 0, soft due time 8, integer importance weight 1. The weight matters for the appropriate weighted objective.
JOB J2 0 8 1	Job J2: release time 0, soft due time 8, integer importance weight 1. The weight matters for the appropriate weighted objective.
OP a J1 1	Operation a belongs to job J1 at sequence position 1. Ascending positions define its within-job order.
OP b J1 2	Operation b belongs to job J1 at sequence position 2. Ascending positions define its within-job order.
OP c J2 1	Operation c belongs to job J2 at sequence position 1. Ascending positions define its within-job order.
OP d J2 2	Operation d belongs to job J2 at sequence position 2. Ascending positions define its within-job order.
OPTION a M1 3	Operation a may run on machine M1 and takes 3 time units there. Multiple OPTION rows are alternative machines.
OPTION a M2 4	Operation a may run on machine M2 and takes 4 time units there. Multiple OPTION rows are alternative machines.
OPTION b M2 2	Operation b may run on machine M2 and takes 2 time units there. Multiple OPTION rows are alternative machines.
OPTION c M2 2	Operation c may run on machine M2 and takes 2 time units there. Multiple OPTION rows are alternative machines.
OPTION d M1 2	Operation d may run on machine M1 and takes 2 time units there. Multiple OPTION rows are alternative machines.
USE a crane 1	While a runs, it consumes 1 units of shared resource crane.
USE d crane 1	While d runs, it consumes 1 units of shared resource crane.
DOWNTIME M1 7 9	Machine M1 is unavailable from time 7 up to 9. It becomes available at 9.
DEADLINE J1 12	Job J1 must finish by 12. This is a hard limit, unlike its soft due date.

Save and run The complete file is `models/flexible_jobshop.txt`.

```
optimizer.exe --schedule --file models/flexible_jobshop.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

Scheduling example: flexible machines and a crane

```
TYPE SCHEDULING
OBJECTIVE MAKESPAN
MACHINE M1
MACHINE M2
RESOURCE crane 1
JOB J1 0 8 1
JOB J2 0 8 1
OP a J1 1
OP b J1 2
OP c J2 1
OP d J2 2
OPTION a M1 3
OPTION a M2 4
OPTION b M2 2
OPTION c M2 2
OPTION d M1 2
USE a crane 1
```

```
USE d crane 1
DOWNTIME M1 7 9
DEADLINE J1 12
```

Run:

```
optimizer.exe --schedule --file models/flexible_jobshop.txt --seconds 10
```

Job J1 has a then b; J2 has c then d. Operation a can use M1 for three time units or M2 for four. The shared crane has capacity one and is required by a and d. M1 is unavailable over [7,9), and J1 has hard deadline 12.

A feasible makespan-5 arrangement is a on M1 at [0,3), c on M2 at [0,2), b on M2 at [3,5) and d on M1 at [3,5). Crane uses do not overlap. The longest job path has length five, so this example's makespan cannot be smaller.

The returned feasible schedule may use equivalent placement/order choices. Check machine identifiers and exact start/end records against the file. A job's due date alone is not a hard deadline.

Scheduling deadlines, tardiness and acceptance

```
TYPE SCHEDULING
OBJECTIVE TOTAL_TARDINESS
MACHINE mill
JOB A 0 3 2
JOB B 0 5 1
OP cutA A 1
OP cutB B 1
OPTION cutA mill 2
OPTION cutB mill 3
DEADLINE A 6
DEADLINE B 7
```

Run:

```
optimizer.exe --schedule --file models/schedule_deadlines.txt --seconds 5
```

A then B on the single mill completes A at 2 and B at 5. Both meet their due dates, so total weighted tardiness is zero. The hard deadlines also hold. If B runs first, A completes at 5 and incurs tardiness; that is feasible under the hard deadline but worse for this objective.

Distinguish objective choices MAKESPAN concerns the final completion time. TOTAL_COMPLETION concerns summed completion contributions; TOTAL_TARDINESS penalizes late completion relative to due dates with the supported weights. Changing an objective changes the preferred schedule, not the hard resource rules.

Before deployment Verify all operations appear once, job and explicit precedence hold, processing duration matches the chosen machine, no machine overlap crosses downtime, cumulative resources remain within capacity and hard deadlines are met.

Do not infer compliance with an unstated factory calendar, setup rule or operator break. The file is the complete represented model.

18. Staff rostering

Staff rostering: record format

Use `--roster` for assigning qualified employees to every demanded shift-role position. This is different from selecting already supplied crew patterns.

```
TYPE ROSTERING
SHIFT morning DAY ABSOLUTE_START ABSOLUTE_END
EMPLOYEE alice MIN_HOURS MAX_HOURS MIN_SHIFTS MAX_SHIFTS MIN_REST MAX_CONSEC_DAYS
SKILL alice nurse
DEMAND morning nurse COUNT
UNAVAILABLE alice morning
COST alice morning nurse COST
```

A SHIFT has a day label and absolute start/end on a consistent timeline. EMPLOYEE defines hard workload, assignment-count, rest and consecutive-day limits. SKILL permits role assignment. DEMAND gives the number of required positions for that role and shift.

UNAVAILABLE forbids an employee on a named shift. COST supplies the supported assignment preference cost; omitted ordinary preferences default to zero. Use nonnegative preference penalties and retain their units consistently.

Hard rules At most one shift per employee per day. All workload minima and maxima are hard. Minimum rest is checked between assigned shifts. MAX_CONSEC_DAYS 0 disables that particular limit. Every demanded position must be covered for a feasible result.

Time scale If time is in hours, day-2 starts might be 24, not 0 again. Reusing local clock time for different days makes rest comparisons incorrect. Choose a horizon origin and encode absolute times.

The service does not infer local labor law, weekend fairness, overtime rates or employee-specific policy. Represent supported rules explicitly. Limits are 1000 shifts and 2000 employees. Fixed workload/time/cost fields accept up to six decimal places. A partial roster is not reported as a complete feasible plan.

Staff rostering: translating staffing requirements

SHIFT morning 1 8 16 names a shift on day 1 that starts at absolute time 8 and ends at 16. For the next day, use absolute times consistently; do not restart the time clock without representing the day offset.

EMPLOYEE alice MIN_HOURS MAX_HOURS MIN_SHIFTS MAX_SHIFTS MIN_REST MAX_CONSEC_DAYS defines the permitted workload and rest limits. Hours, rest and shift endpoints share the same time unit. A maximum-consecutive-days value of 0 disables that particular limit; it does not mean no working days.

SKILL alice nurse says Alice is qualified for nurse work. DEMAND morning nurse 2 requires two qualified employees in that shift/skill role. UNAVAILABLE alice morning forbids assigning Alice to it. COST alice morning nurse 5 gives the cost of that eligible assignment.

Each employee may work at most one shift per day. Rest, hours, shift counts, qualifications and consecutive-day limits are hard rules. The cost ranks feasible schedules; it does not make a forbidden assignment legal. Supply every required shift and staffing role. Standard command: `--roster --file PATH`.

Roster worked file: every line explained

Cover three nursing shifts using Alice and Bob while respecting workloads, rest and availability. Collect demand per role and shift, qualifications and actual assignment costs.

Input line	Meaning in simple English
TYPE ROSTERING	Selects the ROSTERING input grammar. Keep this consistent with the service flag in your command.
SHIFT mon 1 0 8	Shift mon belongs to day 1, begins at absolute time 0, and ends at 8. Duration is end minus start.
SHIFT tue 2 24 32	Shift tue belongs to day 2, begins at absolute time 24, and ends at 32. Duration is end minus start.
SHIFT wed 3 48 56	Shift wed belongs to day 3, begins at absolute time 48, and ends at 56. Duration is end minus start.
EMPLOYEE alice 8 16 1 2 12 2	Employee alice: minimum/maximum hours 8/16; minimum/maximum shifts 1/2; minimum rest 12; maximum consecutive working days 2.
EMPLOYEE bob 8 16 1 2 12 2	Employee bob: minimum/maximum hours 8/16; minimum/maximum shifts 1/2; minimum rest 12; maximum consecutive working days 2.
SKILL alice nurse	Employee alice is qualified for role nurse. A name alone does not infer qualifications.
SKILL bob nurse	Employee bob is qualified for role nurse. A name alone does not infer qualifications.
DEMAND mon nurse 1	Shift mon needs 1 employees qualified for nurse.
DEMAND tue nurse 1	Shift tue needs 1 employees qualified for nurse.
DEMAND wed nurse 1	Shift wed needs 1 employees qualified for nurse.
UNAVAILABLE bob mon	Employee bob cannot work shift mon. A cheap assignment cannot override this restriction.
COST alice mon nurse 1	Assigning alice to shift mon in role nurse costs 1. Costs rank otherwise permitted staffing choices.
COST alice tue nurse 4	Assigning alice to shift tue in role nurse costs 4. Costs rank otherwise permitted staffing choices.
COST alice wed nurse 1	Assigning alice to shift wed in role nurse costs 1. Costs rank otherwise permitted staffing choices.
COST bob tue nurse 1	Assigning bob to shift tue in role nurse costs 1. Costs rank otherwise permitted staffing choices.
COST bob wed nurse 3	Assigning bob to shift wed in role nurse costs 3. Costs rank otherwise permitted staffing choices.

Save and run The complete file is models/rostering.txt.

```
optimizer.exe --roster --file models/rostering.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

Rostering example: three nursing shifts

```
TYPE ROSTERING
SHIFT mon 1 0 8
SHIFT tue 2 24 32
SHIFT wed 3 48 56
EMPLOYEE alice 8 16 1 2 12 2
EMPLOYEE bob 8 16 1 2 12 2
SKILL alice nurse
SKILL bob nurse
DEMAND mon nurse 1
DEMAND tue nurse 1
DEMAND wed nurse 1
UNAVAILABLE bob mon
COST alice mon nurse 1
COST alice tue nurse 4
COST alice wed nurse 1
COST bob tue nurse 1
COST bob wed nurse 3
```

Run:

```
optimizer.exe --roster --file models/rostering.txt --seconds 10
```

Each day needs one nurse for an eight-hour shift. Both employees are qualified. Bob cannot work Monday. Each must receive one or two shifts and eight to sixteen hours, with twelve hours of rest and at most two consecutive working days.

A good roster is Alice on Monday and Wednesday, Bob on Tuesday, total assignment cost 3. Each has adequate rest and workload; Alice's two days are not consecutive. This also achieves the minimum individual costs for the three assignments in this example.

The returned plan must include every demanded role position, not just employees who were assigned. Check hard minimum workloads as well as maxima; leaving an eligible employee unused can violate a minimum even when coverage is complete.

Rostering: prepare real data and verify rules

Start from separate tables for employees, qualifications, shifts, demand, unavailability and preference costs. Use stable identifiers so the result maps back to actual people. Convert all dates/times to one absolute time origin before generating records.

Input checklist - Every demand role has at least one qualified available employee. - The sum of demand can be covered under hard hour and shift limits. - Each employee's minimum is compatible with availability and the planning horizon. - Day labels and absolute times agree, including overnight shifts. - Consecutive-day and rest limits reflect the rules you intend to enforce.

Preference versus permission A high COST makes an assignment undesirable, not forbidden. UNAVAILABLE forbids it. A soft preference should not accidentally become a hard exclusion, and a mandatory restriction should not be represented merely as a small cost.

Output use Map employee and shift identifiers back to names and dates. Recompute coverage by role, total hours, shift count, day count and pairwise rest. Keep the solver's feasible/status fields with the roster.

Unsupported policies No country's labor legislation is automatically inferred. Rules such as detailed overtime, weekend balancing, fair rotation or complicated contract clauses may require a general model or an extension. Do not market a roster as policy-compliant without checking that each applicable rule is represented.

Tuning --service-starts and --service-iterations govern search; --seconds is the time allowance. Try multi-seed diversification where useful. A longer or more diverse run cannot make contradictory hard minima feasible. Limited failure is not proof of infeasibility unless the actual service report establishes a supported contradiction.

19. School timetabling

School timetable: record format

Use --timetable with a declared period grid, rooms, teachers, classes and lesson requirements.

```
TYPE TIMETABLE
SLOT mon1 DAY PERIOD
ROOM lab CAPACITY
TEACHER alice MAX_DAILY_PERIODS MAX_WEEKLY_PERIODS
CLASS class1 SIZE MAX_DAILY_PERIODS
QUALIFIED alice subject
LESSON math class1 subject COUNT DURATION_PERIODS
ROOM_FOR math lab
ALLOW math mon1
MAX_PER_DAY math COUNT
UNAVAILABLE_TEACHER alice mon1
UNAVAILABLE_CLASS class1 mon1
UNAVAILABLE_ROOM lab mon1
PREFERENCE math mon1 COST
GAP_COST alice NONNEGATIVE_COST
```

LESSON defines a requirement: COUNT occurrences, each occupying DURATION_PERIODS consecutive declared periods in one day. Teacher selection follows subject qualification, availability and workload. Class size must fit the room.

ROOM_FOR and ALLOW are optional whitelists for rooms and permitted start slots. They restrict the named lesson requirement; occupied periods beyond its start must also exist and be available. MAX_PER_DAY limits occurrences of that requirement per day.

Hard feasibility Teacher, class and room clashes are forbidden. Workloads count occupied periods, not just lesson starts. All required occurrences must be placed. A double-period lesson consumes two consecutive periods and contributes two to relevant workloads.

Preferences give soft placement costs. GAP_COST penalizes empty declared periods between a teacher's first and last teaching period. It does not invent lunch breaks or rules about undeclared periods.

Limits are 1000 declared slots, teachers, classes and rooms each. No combined classes, arbitrary break policies or unstated fairness rules are inferred. Keep the model within the documented scope.

School timetabling: reading teaching and room rules

SLOT mon1 1 1 creates a slot with day number 1 and period number 1. ROOM lab 30 declares room capacity 30. CLASS class1 25 6 declares 25 pupils and at most 6 occupied periods per day.

TEACHER t1 5 20 permits at most 5 occupied periods per day and 20 per week. QUALIFIED t1 maths makes that teacher eligible to teach maths. LESSON maths1 class1 maths 3 2 requires three occurrences for class1 in maths, each lasting two consecutive periods. COUNT is occurrences; duration is periods per occurrence.

ROOM_FOR maths1 lab makes the named room eligible. ALLOW maths1 mon1 restricts permitted start slots; ensure each chosen start can accommodate the consecutive duration. MAX_PER_DAY maths1 1 allows at most one occurrence that day.

UNAVAILABLE_TEACHER, UNAVAILABLE_CLASS and UNAVAILABLE_ROOM each forbid the named resource at the named slot. PREFERENCE maths1 mon1 2 gives a placement penalty for that lesson start. GAP_COST t1 1 gives the supported teacher-gap cost.

Teacher, class and room clashes are forbidden. A long lesson consumes every occupied period,

not just its start slot. Use matching IDs and enough eligible teachers, rooms and slots. Standard command: `--timetable --file PATH`.

Timetable worked file: every line explained

Schedule maths and science for two classes across two days. Collect occurrences, period duration, teacher qualifications, room capacities and unavailability; do not confuse occurrences with occupied periods.

Input line	Meaning in simple English
TYPE TIMETABLE	Selects the TIMETABLE input grammar. Keep this consistent with the service flag in your command.
SLOT mon1 1 1	Slot mon1 is day 1, period 1. These numbers describe the timetable position, not clock hours.
SLOT mon2 1 2	Slot mon2 is day 1, period 2. These numbers describe the timetable position, not clock hours.
SLOT mon3 1 3	Slot mon3 is day 1, period 3. These numbers describe the timetable position, not clock hours.
SLOT tue1 2 1	Slot tue1 is day 2, period 1. These numbers describe the timetable position, not clock hours.
SLOT tue2 2 2	Slot tue2 is day 2, period 2. These numbers describe the timetable position, not clock hours.
SLOT tue3 2 3	Slot tue3 is day 2, period 3. These numbers describe the timetable position, not clock hours.
ROOM R1 30	Room R1 holds at most 30 pupils and cannot host simultaneous lessons.
ROOM Lab 25	Room Lab holds at most 25 pupils and cannot host simultaneous lessons.
TEACHER Alice 3 6	Teacher Alice may teach at most 3 occupied periods per day and 6 per week.
TEACHER Bob 3 6	Teacher Bob may teach at most 3 occupied periods per day and 6 per week.
CLASS C1 20 3	Class C1 contains 20 pupils and may occupy at most 3 periods per day.
CLASS C2 22 3	Class C2 contains 22 pupils and may occupy at most 3 periods per day.
QUALIFIED Alice maths	Teacher Alice may teach subject maths. This is subject eligibility, not a particular assigned lesson.
QUALIFIED Bob science	Teacher Bob may teach subject science. This is subject eligibility, not a particular assigned lesson.
LESSON maths1 C1 maths 2 1	Lesson requirement maths1 is for class C1 in subject maths; schedule 2 occurrences, each lasting 1 consecutive periods.
LESSON maths2 C2 maths 2 1	Lesson requirement maths2 is for class C2 in subject maths; schedule 2 occurrences, each lasting 1 consecutive periods.
LESSON science1 C1 science 1 2	Lesson requirement science1 is for class C1 in subject science; schedule 1 occurrences, each lasting 2 consecutive periods.
LESSON science2 C2 science 1 2	Lesson requirement science2 is for class C2 in subject science; schedule 1 occurrences, each lasting 2 consecutive periods.
ROOM_FOR science1 Lab	Lesson science1 is permitted in room Lab; choose a room with sufficient capacity.
ROOM_FOR science2 Lab	Lesson science2 is permitted in room Lab; choose a room with sufficient capacity.
MAX_PER_DAY maths1 1	At most 1 occurrences of lesson maths1 may be scheduled on a day.
MAX_PER_DAY maths2 1	At most 1 occurrences of lesson maths2 may be scheduled on a day.
UNAVAILABLE_TEACHER Alice mon1	Teacher Alice cannot teach during slot mon1.
GAP_COST Alice 1	Apply cost 1 to the supported idle-gap measure for teacher Alice. This is a preference penalty, not a prohibition.
GAP_COST Bob 1	Apply cost 1 to the supported idle-gap measure for teacher Bob. This is a preference penalty, not a prohibition.

Save and run The complete file is `models/school_timetable.txt`.

```
optimizer.exe --timetable --file models/school_timetable.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

Timetable example: two classes and two subjects

```

TYPE TIMETABLE
SLOT mon1 1 1
SLOT mon2 1 2
SLOT mon3 1 3
SLOT tue1 2 1
SLOT tue2 2 2
SLOT tue3 2 3
ROOM R1 30
ROOM Lab 25
TEACHER Alice 3 6
TEACHER Bob 3 6
CLASS C1 20 3
CLASS C2 22 3
QUALIFIED Alice maths
QUALIFIED Bob science
LESSON maths1 C1 maths 2 1
LESSON maths2 C2 maths 2 1
LESSON science1 C1 science 1 2
LESSON science2 C2 science 1 2
ROOM_FOR science1 Lab
ROOM_FOR science2 Lab
MAX_PER_DAY maths1 1
MAX_PER_DAY maths2 1
UNAVAILABLE_TEACHER Alice mon1
GAP_COST Alice 1
GAP_COST Bob 1

```

Run:

```
optimizer.exe --timetable --file models/school_timetable.txt --seconds 10
```

There are two days with three periods each. Alice teaches maths, Bob science. Each class needs two single-period maths occurrences and one double-period science occurrence. Science requires Lab; maths is limited to one occurrence per day for each class. Alice is unavailable in mon1.

A returned feasible timetable must place all six occurrences, occupying eight class-periods in total, without teacher/room/class clashes. Equivalent schedules may have different lesson orders. Read the teacher, room, day and occupied periods for each occurrence.

The CLASS row describes its size and daily occupied-period limit. It does not supply a subject curriculum implicitly; every lesson requirement is explicit. A lesson count of two with duration two means two double-period occurrences, not two total occupied periods.

Timetables: data preparation and acceptance

Build tables for slots, rooms, teachers, classes, qualifications and lesson requirements. Then add hard unavailability and optional soft preferences. Use a fixed declared grid; all periods used by a multi-period lesson must exist consecutively in the same day.

Typical modeling mistakes A room smaller than the class; a subject with no qualified teacher; a duration crossing a missing period; daily workloads below the mandatory lesson volume; an ALLOW whitelist with too few starts; or treating preference cost as a hard prohibition.

Teacher choice Teachers are chosen from those QUALIFIED for the subject. Do not assume that a lesson is assigned to a specific teacher just because its identifier includes a person's name. To enforce a rule, it must be represented in the supported data; the current format's teacher eligibility is subject-based.

Read costs carefully PREFERENCE relates to a lesson's start slot. GAP_COST counts empty declared periods between a teacher's first and last teaching period. A gap cost does not prohibit every free period or guarantee an aesthetically preferred timetable.

Acceptance checklist Every required occurrence exists. Classes fit rooms. Teacher/class/room availability holds across all occupied periods. Workload and MAX_PER_DAY

limits hold. No resource is booked twice. Soft costs are recomputed from the selected starts and teacher gaps.

Search Use --seconds, --seed, --service-starts and --service-iterations. Native multi-seed may diversify candidates. A partially assigned timetable is not a feasible result. Limited failure to place every lesson does not prove the requirements are impossible; first review conflicting hard data.

Translate the identifiers into an understandable calendar in your application only after checking the complete feasible result.

20. Worker-job assignment

Worker-job assignment: format and example

```
/* Each worker gets one job; omitted worker/job pairs are forbidden. */
TYPE ASSIGNMENT
WORKER Alice
WORKER Bob
WORKER Carol
JOB Welding
JOB Packing
JOB Inspection
JOB Spare
COST Alice Welding 8
COST Alice Packing 3
COST Alice Inspection 12
COST Alice Spare 20
COST Bob Welding 4
COST Bob Packing 9
COST Bob Inspection 7
COST Bob Spare 20
COST Carol Welding 11
COST Carol Packing 6
COST Carol Inspection 2
COST Carol Spare 20
EOF
```

Run `--assignment --file models/assignment.txt`. Each worker receives exactly one job; each job receives at most one worker. Extra jobs are allowed. Omitted COST pairs are forbidden, not free.

The example's cheapest matching is Alice->Packing, Bob->Welding and Carol->Inspection, cost 9. Spare is an optional unassigned job. All workers are covered. IDs are case-sensitive unique words; repeated conflicting definitions are errors.

This dedicated service does not support job capacities above one, multiple assignments per worker, fixed activation charges or arbitrary side constraints. Use general LP/MILP if those rules are needed. Supported matrix size is at most four million worker-job cells.

Accepted controls are `--seconds`, `--tol`, `--extend`, `--progress` and `--network-iterations`. Multi-seed is rejected because this is a deterministic allocation routine. A complete verified matching can be returned under a limit if available; its status determines whether optimality is established.

Assignment: reading workers, jobs and costs

WORKER Alice creates a worker that must receive exactly one job. JOB Packing creates a job that may receive at most one worker. Extra jobs can remain unused; extra workers cannot simply be ignored.

COST Alice Packing 3 means assigning Alice to Packing costs 3. It also declares that pair eligible. An omitted pair is forbidden, not zero-cost. To permit a free assignment, explicitly write its COST as 0.

The objective is summed assignment cost. This service cannot assign several workers to a job or several jobs to a worker. For shipment quantities use transportation; for multiple resource demands use placement; for arbitrary side constraints use general LP/MILP.

Use one currency or cost scale for every COST row. The cost does not encode duration, worker capacity or qualification rules except through pair eligibility. Standard command: `optimizer.exe --assignment --file assignment.txt`.

Assignment worked file: every line explained

Assign each of three workers to one job, using distinct jobs. Collect eligible worker-job pairs and their costs; omit pairs that are truly forbidden.

Input line	Meaning in simple English
TYPE ASSIGNMENT	Selects the ASSIGNMENT input grammar. Keep this consistent with the service flag in your command.
WORKER Alice	Worker Alice must receive exactly one eligible job.
WORKER Bob	Worker Bob must receive exactly one eligible job.
WORKER Carol	Worker Carol must receive exactly one eligible job.
JOB Welding	Declares job Welding. At most one worker is assigned to it; an extra job can remain unused.
JOB Packing	Declares job Packing. At most one worker is assigned to it; an extra job can remain unused.
JOB Inspection	Declares job Inspection. At most one worker is assigned to it; an extra job can remain unused.
JOB Spare	Declares job Spare. At most one worker is assigned to it; an extra job can remain unused.
COST Alice Welding 8	Worker Alice may do job Welding, at cost 8. This row also permits the pair; omitted pairs are forbidden.
COST Alice Packing 3	Worker Alice may do job Packing, at cost 3. This row also permits the pair; omitted pairs are forbidden.
COST Alice Inspection 12	Worker Alice may do job Inspection, at cost 12. This row also permits the pair; omitted pairs are forbidden.
COST Alice Spare 20	Worker Alice may do job Spare, at cost 20. This row also permits the pair; omitted pairs are forbidden.
COST Bob Welding 4	Worker Bob may do job Welding, at cost 4. This row also permits the pair; omitted pairs are forbidden.
COST Bob Packing 9	Worker Bob may do job Packing, at cost 9. This row also permits the pair; omitted pairs are forbidden.
COST Bob Inspection 7	Worker Bob may do job Inspection, at cost 7. This row also permits the pair; omitted pairs are forbidden.
COST Bob Spare 20	Worker Bob may do job Spare, at cost 20. This row also permits the pair; omitted pairs are forbidden.
COST Carol Welding 11	Worker Carol may do job Welding, at cost 11. This row also permits the pair; omitted pairs are forbidden.
COST Carol Packing 6	Worker Carol may do job Packing, at cost 6. This row also permits the pair; omitted pairs are forbidden.
COST Carol Inspection 2	Worker Carol may do job Inspection, at cost 2. This row also permits the pair; omitted pairs are forbidden.
COST Carol Spare 20	Worker Carol may do job Spare, at cost 20. This row also permits the pair; omitted pairs are forbidden.
EOF	Ends this standard-format example. It is a file terminator, not an object, item or decision.

Save and run The complete file is models/assignment.txt.

```
optimizer.exe --assignment --file models/assignment.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

21. Transportation and shipment allocation

Transportation: balanced shipment allocation

```
TYPE TRANSPORTATION
UNITS INTEGER
SUPPLY FactoryA 20
SUPPLY FactoryB 15
DEMAND ShopA 10
DEMAND ShopB 25
ROUTE A_A FactoryA ShopA 2
ROUTE A_B FactoryA ShopB 5
ROUTE B_A FactoryB ShopA 4
ROUTE B_B FactoryB ShopB 1 15
EOF
```

Run:

```
optimizer.exe --transport --file models/transportation.txt --seconds 10
```

SUPPLY and DEMAND are exact quantities and total supply must equal total demand. ROUTE fields are ID, source, destination, unit cost and optional capacity. Omitted routes are forbidden; omitted capacity defaults to total supply.

A cheapest shipment sends 10 from FactoryA to ShopA, 10 from FactoryA to ShopB and 15 from FactoryB to ShopB, cost 85. UNITS INTEGER requires exact integral supply/demand/capacity data and returns integral flows. Default CONTINUOUS permits fractional flows.

For surplus, shortage or disposal, add an explicit dummy source/destination and its actual penalty. No automatic balancing is invented. Costs are per shipped unit. Fixed route-opening charges or multiple commodities require a different general model.

Options are the dedicated network controls on the corresponding option reference. --transportation is an alias for --transport. --extend is supported; multi-seed is not. Integer total supply and summed arc capacities must remain within $2^{53}-1$ for supported bookkeeping.

Transportation: reading shipment quantities and prices

SUPPLY FactoryA 20 means FactoryA supplies exactly 20 shipment units. DEMAND ShopA 10 means ShopA requires exactly 10 units. Total supply must equal total demand.

ROUTE A_A FactoryA ShopA 2 permits shipment from FactoryA to ShopA, at cost 2 per unit. A_A is your unique route name. ROUTE B_B FactoryB ShopB 1 15 costs 1 per unit and allows at most 15 units on that route. Omitting the final capacity defaults it to total supply. An omitted route is forbidden, not free.

UNITS INTEGER requires integral supplies/capacities and whole-number shipments. UNITS CONTINUOUS permits fractions. All quantities share a unit, and costs share a per-unit cost scale. This service allocates shipments; it does not generate truck tours or fixed route-activation charges.

Represent surplus/shortage explicitly with an appropriate dummy source or destination and its real penalty. No balancing is automatically invented. Standard command: optimizer.exe --transport --file shipments.txt.

Transport worked file: every line explained

Ship 35 units from two factories to two shops. Collect exact supply, exact demand, permitted shipping routes, per-unit prices and any route capacity.

Input line	Meaning in simple English
TYPE TRANSPORTATION	Selects the TRANSPORTATION input grammar. Keep this consistent with the service flag in your command.
UNITS INTEGER	INTEGER requires whole-number shipment/flow quantities with integral data; CONTINUOUS permits fractional quantities. Neither setting declares money amounts integer.
SUPPLY FactoryA 20	Source FactoryA supplies exactly 20 units. Total supply must equal total demand.
SUPPLY FactoryB 15	Source FactoryB supplies exactly 15 units. Total supply must equal total demand.
DEMAND ShopA 10	Destination ShopA requires exactly 10 shipment units.
DEMAND ShopB 25	Destination ShopB requires exactly 25 shipment units.
ROUTE A_A FactoryA ShopA 2	Route A_A goes from source FactoryA to destination ShopA, costing 2 per shipped unit; omitted capacity defaults to total supply.
ROUTE A_B FactoryA ShopB 5	Route A_B goes from source FactoryA to destination ShopB, costing 5 per shipped unit; omitted capacity defaults to total supply.
ROUTE B_A FactoryB ShopA 4	Route B_A goes from source FactoryB to destination ShopA, costing 4 per shipped unit; omitted capacity defaults to total supply.
ROUTE B_B FactoryB ShopB 1 15	Route B_B goes from source FactoryB to destination ShopB, costing 1 per shipped unit; capacity is at most 15 units.
EOF	Ends this standard-format example. It is a file terminator, not an object, item or decision.

Save and run The complete file is models/transportation.txt.

```
optimizer.exe --transport --file models/transportation.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

22. Minimum-cost network flow

Minimum-cost network flow

```
TYPE MIN_COST_FLOW
UNITS INTEGER
NODE Source 10
NODE Transfer 0
NODE Sink -10
ARC direct Source Sink 0 10 8
ARC first Source Transfer 0 10 2
ARC second Transfer Sink 0 10 3
EOF
```

Run:

```
optimizer.exe --mincost-flow --file models/mincost_flow.txt --seconds 10
```

NODE balance is outgoing minus incoming flow. Positive means supply; negative means demand; zero is a transfer node. ARC fields are ID, from, to, lower bound, upper bound and unit cost.

The example sends 10 units through Source->Transfer->Sink at unit cost 5, total 50, instead of direct cost 8 per unit. All balances and arc bounds are checked.

Finite capacities are required. Negative costs, parallel arcs, self loops and lower bounds are supported. UNITS INTEGER requires exactly integral quantity fields; CONTINUOUS is the default. General integer declarations and arbitrary equations are not accepted in this dedicated record format.

Multiple commodities, fixed charges and nonlinear arc costs are outside this service. Use LP/MILP or NLP with the full formulation when required. Limits support up to 100000 nodes and one million input arcs. A numerical bound/gap can establish optimality within checks; a limited run may return a feasible flow without that conclusion.

Minimum-cost flow: reading balances and arcs

NODE source 8 gives that node net supply 8. NODE sink -8 gives net demand 8. NODE hub 0 requires flow conservation. Positive supply means outgoing minus incoming flow equals the given value. Total balances must sum to zero.

ARC route source hub 0 5 2 names a directed arc from source to hub, lower flow 0, upper flow 5 and cost 2 per unit sent. The first numeric field is a lower bound, not a distance; the second is a capacity. Flow 4 on that arc costs 8.

UNITS INTEGER requires supported integral flows. UNITS CONTINUOUS allows fractional amounts. Flow units must match node balances and arc bounds. Costs are in a consistent per-flow-unit scale.

Omitted arcs do not exist. To permit reverse movement, supply the appropriate reverse directed arc explicitly. This is a network allocation, not a vehicle-route sequence. Standard command: optimizer.exe --mincost-flow --file network.txt.

Flow worked file: every line explained

Send ten units from Source to Sink, optionally through Transfer. Collect each node balance and each directed arc lower/upper flow limit and per-unit cost.

Input line	Meaning in simple English
TYPE MIN_COST_FLOW	Selects the MIN_COST_FLOW input grammar. Keep this consistent with the service flag in your command.
UNITS INTEGER	INTEGER requires whole-number shipment/flow quantities with integral data; CONTINUOUS permits fractional quantities. Neither setting declares money amounts integer.
NODE Source 10	Node Source has net outgoing-minus-incoming balance 10. Positive is supply, negative is demand, zero is a transfer node.
NODE Transfer 0	Node Transfer has net outgoing-minus-incoming balance 0. Positive is supply, negative is demand, zero is a transfer node.
NODE Sink -10	Node Sink has net outgoing-minus-incoming balance -10. Positive is supply, negative is demand, zero is a transfer node.
ARC direct Source Sink 0 10 8	Arc direct goes from Source to Sink: minimum flow 0, maximum flow 10, cost 8 per flow unit.
ARC first Source Transfer 0 10 2	Arc first goes from Source to Transfer: minimum flow 0, maximum flow 10, cost 2 per flow unit.
ARC second Transfer Sink 0 10 3	Arc second goes from Transfer to Sink: minimum flow 0, maximum flow 10, cost 3 per flow unit.
EOF	Ends this standard-format example. It is a file terminator, not an object, item or decision.

Save and run The complete file is models/mincost_flow.txt.

```
optimizer.exe --mincost-flow --file models/mincost_flow.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

23. Resource placement and rebalancing

Resource placement: model format and field meanings

Resource placement assigns every object to exactly one eligible bin, respecting several capacity types at once. The objective is total placement cost plus supported migration charges.

```
TYPE RESOURCE_PLACEMENT
RESOURCE cpu
RESOURCE memory
BIN east 8 16
BIN west 8 16
OBJECT app1 4 6
OBJECT app2 4 8
COST app1 east 2
COST app1 west 5
ALLOW app2 east west
INITIAL app1 west
MIGRATION app1 1
GROUP replicas app1 app2
```

```
optimizer.exe --placement --file models/placement.txt --seconds 5
```

RESOURCE declaration order determines the numeric columns. BIN east 8 16 means east has CPU capacity 8 and memory capacity 16. OBJECT app1 4 6 means app1 consumes CPU 4 and memory 6 in whichever bin receives it. Resources need not share a unit, but each column must consistently use its own unit.

COST app1 east 2 charges 2 for that placement. Missing COST is zero unless DEFAULT_COST changes the object's default; it is not forbidden. ALLOW app2 east west restricts app2 to those bins. Without ALLOW, all bins are eligible.

Resource placement: migration, groups and options

INITIAL app1 west records the existing bin. It does not force that assignment or promise the initial arrangement is feasible. MIGRATION app1 1 charges an additional 1 when app1 moves away from its INITIAL bin. Without INITIAL, the migration field has no effect.

GROUP replicas app1 app2 requires these objects to occupy different bins: at most one group member may occupy each bin. Each object may belong to one such group. This is useful for replica separation; it is not a general statement about arbitrary pairwise conflicts.

In the preceding example, app1 in east costs 2+1=3 including migration, while app1 in west costs 5. app2 has no placement charge. A feasible lower-cost plan puts app1 in east and app2 in west, objective 3.

Option	Meaning and default
--seconds S	Time limit, default 5; 0 disables this clock limit.
--seed N	Random seed, default 42.
--tol T or --ctol T	Absolute resource-capacity tolerance, default 1e-9.
--placement-starts N	Construction-attempt limit, default 1000.
--placement-moves N	Improvement-work limit, default 2000000.
--placement-neighbors N	Repair-size control, default 8; allowed 1 through 12.
--progress	Show verified improvements.

Shared multi-seed controls are supported; --seconds remains the total budget. --extend is unsupported for this service. Objects are not split between bins. There are no lower fill limits. Returned assignments are checked, but limited heuristic search does not generally prove global

optimality.

Placement worked file: every line explained

Place two applications into two bins subject to CPU, memory and replica separation. Collect demands, capacities, costs, current assignments and migration charges.

Input line	Meaning in simple English
TYPE RESOURCE_PLACEMENT	Selects the RESOURCE_PLACEMENT input grammar. Keep this consistent with the service flag in your command.
RESOURCE cpu	Declares resource column cpu. Declaration order defines the positions of all capacity and demand values.
RESOURCE memory	Declares resource column memory. Declaration order defines the positions of all capacity and demand values.
BIN east 8 16	Bin east has resource capacities 8, 16, in RESOURCE declaration order: CPU then memory in this example.
BIN west 8 16	Bin west has resource capacities 8, 16, in RESOURCE declaration order: CPU then memory in this example.
OBJECT app1 4 6	Object app1 consumes resource amounts 4, 6, in the same resource order. Place the entire object in exactly one bin.
OBJECT app2 4 8	Object app2 consumes resource amounts 4, 8, in the same resource order. Place the entire object in exactly one bin.
COST app1 east 2	Placing object app1 in bin east costs 2, before any migration charge. Missing costs default to zero unless overridden.
COST app1 west 5	Placing object app1 in bin west costs 5, before any migration charge. Missing costs default to zero unless overridden.
ALLOW app2 east west	app2 may use only these listed bins: east, west. This is eligibility, not a cost.
INITIAL app1 west	Object app1 currently occupies bin west. This is a starting allocation, not a fixed assignment.
MIGRATION app1 1	Moving app1 away from its INITIAL bin adds cost 1. Without INITIAL this penalty has no effect.
GROUP replicas app1 app2	Group replicas contains app1, app2. At most one member can occupy any one bin.

Save and run The complete file is models/placement.txt.

```
optimizer.exe --placement --file models/placement.txt --seconds 10
```

Construct your own problem Keep the record keywords; replace identifiers and data with your real requirements. Use consistent units, match every referenced name, and preserve quantities and hard limits. Check the returned allocation against those original requirements. Do not treat a best feasible result as proven optimal unless its status explicitly establishes that.

24. Import existing LP, MPS and NL files

Choose the appropriate entry point

An optimization model has two layers: the modeling language describes how to build a problem, and the generated model contains the variables, objective and constraints to solve. AMPL and GAMS build their own generated models. Optimizer reads the supported generated representation and uses its existing solving methods.

Your input	How to submit it	Required additional software
Native equations or optimization text	optimizer.exe--fileproblem.txt	None
CPLEX-style affine LP file	optimizer.exe--input-formatlp--fileproblem.lp	None
Ordinary MPS linear/integer file	optimizer.exe--mps--fileproblem.mps	None
Supported ASCII or binary NL	optimizer.exe--nl--fileproblem.nl	None
AMPL model and data files	Python solve_ampl_file, or use Optimizer as an AMPL solver	Installed AMPL runtime and its applicable license
GAMS source with one executed solve	Python solve_gams_file	Installed GAMS runtime with CONVERT and its applicable license
Instantiated Pyomo model	Python solve_pyomo	Pyomo Python package
Existing affine	Retained optimizer_models.py	The corresponding modeling package
Gurobi/PuLP/Pyomo model	converters	

Reading a file does not guarantee that every construct in that format is supported. The importers check their documented mathematical subset and reject unsupported constructs. They do not replace nonlinear terms with linear approximations, relax integer domains, or silently ignore unsupported constraints.

Standard command-line file selection

Use an explicit import selector and one input file:

```
optimizer.exe --input-format lp --file problem.lp --seconds 20
optimizer.exe --mps --file problem.mps --seconds 20
optimizer.exe --nl --file problem.nl --seconds 20
```

--input-format accepts lp, mps, nl, and auto. Equivalent explicit aliases are --lp-import, --mps, and --nl. The historical --lp-filePATH and --mps-filePATH remain accepted, and --nl-filePATH is added.

The standard --MODE--filePATH form also remains available for native dedicated services. Do not combine a dedicated-service selector with an import-format selector. For example, use --assignment--filejobs.txt for an assignment service model, or import an LP representation of a general optimization problem; those are separate entry points.

File extensions .lp, .mps, and .nl are detected case-insensitively:

```
optimizer.exe --file problem.lp --seconds 20
optimizer.exe --file problem.mps --seconds 20
optimizer.exe --file problem.nl --seconds 20
```

A sole existing filename also retains its historical shorthand behavior. For other extensions, auto falls back to the existing native file handling. To import an LP file named problem.txt, specify --input-formatlp. Native equation files should normally use .txt; an explicit .lp extension now selects the LP importer.

--linear/--lp selects the native linear solver; it does not, by itself, select an external LP file

format. `--input-format`lp explicitly selects that format. An actual .lp file is also recognized by its extension.

Use one import file. Supplying both `--lp-file`PATH and `--file`PATH, for example, is rejected. Quote Windows paths containing spaces.

New import options

Option	Meaning
<code>--input-format</code> FORMAT (auto, lp, mps or nl)	Select the external file format; use with one <code>--file</code> PATH
<code>--lp-import</code>	Select LP import with <code>--file</code> PATH
<code>--mps</code>	Select MPS import with <code>--file</code> PATH
<code>--nl</code>	Select NL import with <code>--file</code> PATH
<code>--lp-file</code> PATH	Historical LP selector and path together
<code>--mps-file</code> PATH	Historical MPS selector and path together
<code>--nl-file</code> PATH	NL selector and path together
<code>--objective-number</code> N	Select one NL objective by zero-based index; required when an NL file has multiple objectives
<code>--sol-file</code> PATH	Write an AMPL solution file for NL solving
<code>--convert-only</code>	Print a native-text translation without solving
<code>--import-help</code>	Display the import help

Other solving options keep their existing meanings. Typical examples are `--seconds`, `--tol`, `--extend`, `--seed`, LP/MILP limits and gap settings. `--global-opt` is available for supported nonlinear models with the finite explicit bounds and supported operations required by that mode. It is not a guarantee of exact global optimality. Multi-seed NL solving can write a .sol file; it contains the final retained candidate, not the last trial's candidate.

`--convert-only` cannot be combined with solving options or `--sol-file`. Its output is a native equation/optimization file, including a comment-form copyright line. It proves neither feasibility nor optimality.

LP example: a production decision

File imports/interop_production.lp:

```
Maximize
  profit: 3 x + 2 y
Subject To
  capacity: x + y <= 4
  limit: x <= y + 2
  range: 1 <= x + y <= 4
Bounds
  0 <= x <= 4
  0 <= y <= 4
End
```

x and y are production quantities. Maximize says to make the total profit as large as possible. One unit of x earns 3 units of profit and one unit of y earns 2. capacity limits total production to 4. limit permits at most two more units of x than y. range requires total production between 1 and 4. Bounds restricts each quantity to the stated interval. The labels profit, capacity, limit and range identify expressions; they are not variables.

Run:

```
optimizer.exe --input-format lp --file imports\interop_production.lp --seconds 20
```

The example's optimum is x=3, y=1, with objective 11.

LP support includes affine sums, objective constants, numeric or affine right-hand sides, `<=`, `>=`, `=`, and constant ranged constraints such as `1<=x+y<=4` or the reversed `4>=x+y>=1`. Bounds can be one-sided, two-sided, fixed, or free. General/Integer selects integer variables and Binary selects zero-or-one variables. Ordinary undeclared LP variables default to nonnegative.

Place section headers on separate lines. Backslash line comments and labeled/multiline expressions are supported. For continued constraints, use a label on each new row. A complete RHS on a line ends that constraint; split RHS continuations are not a general-purpose grammar. Variable names use the supported identifier subset, including letters, digits, underscore and permitted dots; quoted/bracketed names are not accepted.

Quadratic LP sections, SOS, indicator constraints, strict inequalities and semi-continuous/semi-integer domains are not supported by this LP importer. A nonlinear model should be exported as supported NL instead of being disguised as an LP file.

MPS example: the same production decision

File imports/interop_production.mps:

```
NAME PRODUCTION
OBJSENSE MAX
ROWS
  N PROFIT
  L CAPACITY
  L LIMIT
COLUMNS
  x PROFIT 3 CAPACITY 1
  x LIMIT 1
  y PROFIT 2 CAPACITY 1
  y LIMIT -1
RHS
  rhs CAPACITY 4 LIMIT 2
BOUNDS
  UP b x 4
  UP b y 4
ENDATA
```

NAME gives a model name. OBJSENSEMAX selects maximization. ROWS lists the objective and constraints: N identifies the objective row, L means less than or equal, G means greater than or equal, and E means equality.

COLUMNS records coefficients by variable. The line xPROFIT3CAPACITY1 gives x a coefficient of 3 in the objective and 1 in the capacity constraint. yLIMIT-1 makes the second constraint $x - y \leq 2$. RHS supplies constraint right-hand sides. rhs and b are vector/set names, not variables. UPbx4 supplies the upper bound 4 for x; the ordinary lower bound defaults to zero. ENDATA ends the model.

Run:

```
optimizer.exe --mps --file imports\interop_production.mps --seconds 20
```

The optimum is again $x=3$, $y=1$, objective 11.

The supported subset includes free fields and ordinary fixed-field column/vector records, with continuation of omitted column/vector names. It supports OBJSENSE, OBJNAME, INTORG/INTEND, one RHS set, one RANGES set and one BOUNDS set. Supported bounds are LO, UP, FX, FR, MI, PL, BV, LI, and UI. Fortran D exponent notation is accepted.

INTORG columns retain the engine's documented default [0,1]. Explicit LI removes that implicit upper bound, while UI, UP, PL, and FR override it as documented. To model a general nonnegative integer explicitly, provide the appropriate LI/UI bounds rather than relying on a marker default. Explicit lower bounds are preserved even if a later upper bound makes the model infeasible. Duplicate RHS entries are rejected, including duplicates whose first value is zero.

Multiple vector sets, referenced additional free/objective rows, SOS, quadratic/nonlinear extensions and semi-domains are unsupported. Compressed files are not read directly. This is not a claim of complete compatibility with every solver's MPS dialect or every unusual fixed-field naming convention.

NL: generated algebraic models

NL carries already-instantiated variables, objective expressions and algebraic constraints. Optimizer accepts the supported ASCII and binary NL files generated by modeling tools. Use an exported file, rather than writing NL records by hand.

Variable bounds, equalities, inequalities, ranged constraints, objective direction and integer domains are retained. NL models without an objective run as feasibility searches with objective zero; this import behavior is separate from native multiple-root equation mode. Constant NL models are also accepted.

Supported nonlinear operations include addition, subtraction, multiplication, division, powers, negation, abs, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, natural log, log10, sqrt, atan2, and list min, max, and sums. Availability in the NL importer is distinct from the narrower operator/domain requirements of `--global-opt`.

Unsupported operations include external/imported functions, strings, logical/conditional operators, floor/ceil, remainder/rounding operators, complementarity, specialized network records, SOS semantics, and unconverted piecewise-linear operations. Unsupported codes produce an error. Reformulate them explicitly or use the modeling runtime's supported algebraic reformulations before export.

Current NL import retains primal starting values for nonlinear initialization; every candidate is rechecked against the original model. Dual starting values are validated/read but are not used. A primal hint is not checkpoint continuation, basis transfer or a proof, and not every backend consumes it identically. Ordinary auxiliary suffix annotations are validated and not used, with an import notice. SOS suffixes are rejected because they can change the feasible set. Do not rely on suffix-only branching priorities, basis states or specialized solver behavior.

A `.col` file beside an NL file supplies original variable names in NL column order. If present, its names/count must be valid and unique. Without it, names default to `v0`, `v1`, etc. Original names are available in the structured Python result; the CLI reports the name mapping when aliases are required. Aliases safely allow names such as `quantity['A']` that are not native equation identifiers.

Import support is not a promise that a very large nonlinear model will solve quickly. Check the selected solver service's intended scope and validate smaller instances before committing a long run.

ASCII coefficient literals are retained. Binary coefficients carry the precision provided by the exporting runtime; extended refinement cannot recover digits that the exporting runtime already rounded. AMPL/GAMS numeric processing and Python modeling values can likewise limit source-model precision.

Writing and reading AMPL solution files

```
optimizer.exe --nl --file model.nl --sol-file model.sol --seconds 30
```

The output path must not overwrite the NL input or its `.col/.row` name sidecars. A previous solution at the requested path is removed before solving, after path checks, so rejected input does not leave a stale solution there.

The solution contains the final retained candidate in NL column order. It does not contain invented dual values, basis information or sensitivity information. Extended result strings are retained in the solution file; a receiving modeling runtime may store them at its own precision.

SOL result code	Meaning in this release
0	Native linear backend reports optimality within its numerical tolerances
200	Native linear backend reports infeasibility within numerical tolerances
300	Native linear backend reports unboundedness within numerical tolerances
400	Uncertified/search-limited result; the message and primal-value count say whether a verified feasible candidate exists
500	Solver/input failure after a readable model was accepted

A nonlinear candidate is not labeled globally optimal merely because it is feasible or passes a first-order test. A heuristic failure to find a feasible candidate is not labeled infeasible. Input parsing/path errors can stop before a new solution file is written.

Standalone CLI exits retain their existing meaning: 0 for a returned feasible result/success, 1 for input/build errors, and 2 for no feasible candidate. In -AMPL driver mode, successfully reporting a non-error solver status through .sol returns process exit 0 so AMPL can read that status. A model/solver error remains a process error.

Inspect a conversion before solving

```
optimizer.exe --input-format lp --file imports\interop_production.lp --convert-only >
converted_model.txt
optimizer.exe --linear --file converted_model.txt --seconds 20
```

The first command writes the supported mathematical translation to a native text file; do not also pass --seconds, --extend or other solve settings to --convert-only. Check objective direction, variable domains, bounds and constraint senses in the translated file. Conversion does not establish feasibility or optimality. Keep the original export alongside the translation for reproducibility.

A second import example: nonlinear curve fitting

The companion imports/ampl_curve.nl and name sidecars are generated from this AMPL model:

```
var x >= 0, <= 4;
minimize fit: (x - 2)^2;
subject to minimum_level: x >= 1;
```

```
optimizer.exe --nl --file imports\ampl_curve.nl --seconds 5
```

x is a fitted parameter allowed between 0 and 4. The objective minimizes its squared difference from 2. The extra condition requires x to be at least 1. x=2 is feasible and has objective zero. The generated file already contains the instantiated model: an AMPL installation is not required merely to read this retained NL export. It is required to compile changed AMPL source. Keep the .col sidecar with the export to retain original names.

25. AMPL and GAMS modeling-language workflows

Use Optimizer directly from AMPL

Keep a compiled optimizer.exe available and select it as the AMPL solver:

```
model curve.mod;
option solver 'C:/Patodia/optimizer.exe';
option optimizer_options 'seconds=20 tol=1e-9 seed=42';
option optimizer_auxfiles 'rc';
solve;
display x, solve_result_num, solve_message;
```

AMPL handles model/data expansion, indexing and its own presolve. It invokes Optimizer with a generated stub and -AMPL. Optimizer reads STUB.nl, solves the supported representation and writes STUB.sol. Both default binary NL and explicit ASCII NL are accepted.

The standalone driver form is:

```
optimizer.exe model_stub -AMPL seconds=20 tol=1e-9
```

The stub can also end in .nl. Driver options use key=value, separated by spaces. Supported keys include seconds, tol, ctol, gtol, ftol, seed, starts, iterations, lp-pivots, mip-nodes, mip-gap, mip-abs-gap, global-gap, global-abs-gap, global-nodes, global-pivots, and objective-number. Underscores are accepted in place of key hyphens. Boolean extend and global-opt accept 0 or 1. Unknown or empty options are rejected. AMPL's optimizer_options environment setting is read before explicit driver arguments.

For a manual ASCII export:

```
model curve.mod;
option auxfiles 'rc';
write gcurve;
```

Then run optimizer.exe--nl--filecurve.nl--sol-filecurve.sol. Back in the same AMPL model context, solutioncurve.sol; loads the returned solution. With a path containing spaces, quote the complete write argument, including its leading g, for example write'gC:/Models/MyCurve';.

Do not expect files written by one presolved model to match a differently instantiated model when loading a solution. Keep the generation/model state consistent.

Build an indexed AMPL model and data file

imports/demand.mod:

```
set PRODUCTS;
param profit {PRODUCTS};
param capacity;
var quantity {PRODUCTS} >= 0;
maximize revenue: sum {p in PRODUCTS} profit[p] * quantity[p];
subject to limit: sum {p in PRODUCTS} quantity[p] <= capacity;
```

imports/demand.dat:

```
set PRODUCTS := A B;
param profit := A 3 B 2;
param capacity := 4;
```

Statement	Meaning in simple English
set PRODUCTS	A named list of products; the data file supplies A and B.

Statement	Meaning in simple English
param profit {PRODUCTS}	A fixed profit value for each product.
param capacity	Fixed total production capacity.
var quantity {PRODUCTS} >= 0	One nonnegative unknown quantity for each product.
maximize revenue: sum ...	Add each product's profit times quantity, and choose the highest total.
subject to limit: sum ... <= capacity	Combined production may not exceed the supplied capacity.

With these data, produce 4 units of A and 0 of B for an objective of 12. Use AMPL to instantiate indexed products and data, then let Optimizer solve the generated model. AMPL retains its own language, data handling and licensing requirements.

GAMS: export and solve an instantiated model

imports/curve.gms:

```
Variables x, z;
Equations objective, safety;
x.lo = 0; x.up = 4;
objective.. z =e= sqr(x - 2);
safety.. x =g= 1;
Model curve /all/;
Solve curve using nlp minimizing z;
```

Statement	Meaning in simple English
Variables x, z	x is the chosen parameter; z represents the objective value.
Equations objective, safety	Name two mathematical relationships.
x.lo=0; x.up=4	Inclusive lower and upper limits for x.
objective.. z =e= sqr(x-2)	The objective value equals the squared difference from 2.
safety.. x =g= 1	x must be at least 1.
Model curve /all/	Include all declared equations in this model.
Solve ... minimizing z	Instantiate a nonlinear model whose objective is to minimize z.

This example has $x=2$ and objective zero. Call `solve_gams_file` as shown in the Python chapter. GAMS and its CONVERT facility must be installed and usable. The bridge handles one executed solve; it does not register a native GAMS solver link. It exports and solves the model, returning a result dictionary to Python. It does not feed solution values or marginal values back into GAMS .l/.m fields or execute arbitrary post-solve GAMS statements with those results. If model generation eliminates a variable, the bridge does not invent a value for it.

Choose the right workflow

Use the AMPL solver-driver workflow when you need AMPL to load a .sol result into the current model session. Use the Python AMPL/GAMS methods for an application that needs a returned dictionary. Export a persistent NL file when you need to rerun precisely the same instantiated mathematical model without repeating compilation. Renaming .mod or .gms to .txt does not translate a modeling language.

26. Python modeling interfaces and imported files

Python interface

Keep optimizer_ctypes.py, optimizer_model_io.py, and the DLL accessible. Example:

```
from optimizer_ctypes import OptimizerDLL

solver = OptimizerDLL(r'C:\Patodia\optimizer.dll')
print(solver.model_import_formats)
result = solver.solve_file('problem.lp', seconds=20)

if result['feasible']:
    print(result['best']['objective'])
    print(result['values'])
else:
    print(result['message'])
```

The model_import_formats capability tuple reports the new DLL's documented lp, mps, nl-ascii, and nl-binary subsets. It is empty for older compatible DLLs without the added capability query. Calling a new feature requires rebuilding the DLL; replacing only the Python files cannot add a native parser to an old DLL.

Explicit file methods are solve_lp_file, solve_mps_file, and solve_nl_file. solve_file(path, input_format='lp') selects an explicit format; omitting that keyword uses extension/native-type detection.

```
result = solver.solve_nl_file(
    'model.nl', sol_file='model.sol', seconds=30,
    extend=True, tol='1e-24', objective_number=0)
```

Use decimal strings for very small tolerances and inspect objective_text/values_text when the printed extended digits matter. Numeric convenience values are Python numbers, and nonintegral values normally become Python float.

The returned dictionary retains the existing result fields: exit_code, feasible, best, candidates, values, best_bound, gaps, status, and string forms where available. import_names maps internal aliases to original names. A successful process exit alone is not an optimality certificate; check the applicable result status and bound/gap fields.

AMPL model and data from Python

File imports/demand.mod defines indexed product quantities and a profit objective. imports/demand.dat supplies two products, profit coefficients 3 and 2, and capacity 4.

```
result = solver.solve_ampl_file(
    'imports/demand.mod',
    data_files=['imports/demand.dat'],
    ampl_executable=r'C:\AMPL\ampl.exe',
    compile_seconds=60, seconds=20)
print(result['values'])
```

The expected example objective is 12: produce 4 units of product A. The dictionary uses AMPL's original generated names, such as quantity['A'].

The bridge reads a model and supplied data through installed AMPL, disables presolve for that export, writes ASCII NL and name sidecars in a temporary directory, and calls the DLL. Relative source includes are evaluated from the model's directory. data_files must be a sequence, even for one file.

This method solves one exported model. It is not an implementation of arbitrary .run solver-

control scripts with result substitution, and does not reload a solution into a continuing AMPL session. Use the direct AMPL driver for normal solve/solution-loading behavior. Model files are processed by the actual runtime; use model/data inputs you intend that runtime to execute. Paths containing quote characters are rejected by this bridge.

GAMS from Python

File imports/curve.gms declares x , an objective variable z , the equality $z=(x-2)^2$, and $x \geq 1$, with $0 \leq x \leq 4$. It contains one NLP solve minimizing z .

```
result = solver.solve_gams_file(
    'imports/curve.gms',
    gams_executable=r'C:\GAMS\gams.exe',
    compile_seconds=60, seconds=20)
print(result['best']['objective'], result['values'])
```

The expected optimum for the example is objective 0 with $x=2$.

The bridge asks GAMS CONVERT to export one model as ASCII NL with names, validates that exactly one solve used CONVERT, then calls Optimizer. Multiple executed solves, an overriding solver that prevents export, missing sidecars, runtime errors or an unsupported generated model cause explicit errors.

This is an **export-and-solve bridge**, not a registered GAMS solver link. It does not inject Optimizer values into GAMS .l/.m attributes or rerun post-solve GAMS calculations with those values. The returned variables are those retained by GAMS's NL export. GAMS may eliminate its objective variable; in that case the returned objective represents its value. Do not interpret missing eliminated variables as zero.

For multiple scenarios, call the bridge separately for each single-solve model/data instance, or export each generated model explicitly. A Python loop can compare the verified returned candidates.

Pyomo nonlinear or mixed-integer models

Install the optional modeling package:

```
python -m pip install pyomo
```

```
import pyomo.environ as pyo
from optimizer_ctypes import OptimizerDLL

m = pyo.ConcreteModel()
m.x = pyo.Var(bounds=(0, 4))
m.goal = pyo.Objective(expr=(m.x-2)**2)
m.minimum_level = pyo.Constraint(expr=m.x >= 1)

solver = OptimizerDLL(r'C:\Patodia\optimizer.dll')
result = solver.solve_pyomo(m, load_solution=True, seconds=20)
if result['feasible']:
    print(pyo.value(m.x), result['best']['objective'])
```

This exports an already-instantiated algebraic model to NL. It does not execute a Python source file to discover a model. The Pyomo writer and Optimizer importer check the supported domains and expressions. Multiple active Pyomo objectives require your application to select/deactivate objectives before export.

`load_solution=False` is the default and leaves the model values unchanged. With `True`, a verified returned candidate is assigned to the original Pyomo variables. Solver duals are not loaded. Only variables retained by the writer are returned/updated; unused or eliminated variables are not assigned fabricated values.

`compile_seconds` applies to external AMPL/GAMS compilation, separately from the solver's seconds. Returned `model_compile_seconds` records compilation/export time, while the usual solver/API timing fields retain their existing definitions. Temporary exports are cleaned up; use `solve_nl_file` on a retained export if you need a persistent .sol file or independently reproducible generated input.

Complete file-import program

Save this program beside source/ and imports/, or use the companion languages/model_imports.py. It checks feasibility before reading a candidate.

```
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).resolve().parents[1] / "source"))
from optimizer_ctypes import OptimizerDLL

engine = OptimizerDLL(sys.argv[1])
root = Path(__file__).resolve().parents[1]
for name in ("interop_production.lp", "interop_production.mps", "ampl_curve.nl"):
    result = engine.solve_file(root / "imports" / name, seconds=5)
    if not result["feasible"]:
        print(name, result["message"])
        continue
    point = result["best"]
    print(name, "objective:", point["objective_text"])
    for variable, value in point["values_text"].items():
        print(" ", variable, "=", value)
```

```
python languages\model_imports.py C:\Patodia\optimizer.dll
```

The first two models have objective 11; the NL example has objective zero. Adapt the file list to your exported models. Do not assume a feasible nonlinear result proves its global optimum.

Existing affine Gurobi, PuLP and Pyomo models

The retained source/optimizer_models.py builders/converters support their documented affine subset. They require the corresponding modeling package to be installed. Use those converters for linear coefficients, domains and constraints. Use solve_pyomo's NL export for supported nonlinear Pyomo models instead. The adapter is not a complete interpreter for every Python modeling library, callback, custom function or solver-specific construct.

Compilation time versus solving time

An AMPL/GAMS call has a model-generation stage and a solve stage. compile_seconds limits external generation; seconds limits the solver run. The total application call can take longer than seconds. Keep model_compile_seconds, elapsed_seconds and the original data when comparing repeat runs. A different modeling-runtime presolve configuration can change the exported variables even when the underlying business model is the same.

27. Multiple seeds and reproducible experiments

Native multi-seed search: usage and scope

Native multi-seed tries several seeds, retains the best verified candidate and uses remaining time on the winning seed. For minimization it keeps the lowest objective; for maximization the highest. Root mode accumulates distinct verified roots instead of ranking by cost.

```
optimizer.exe --crew --file model.txt --multseeds --seed_numbers 5 --seed-seconds 7 --seconds 60
--seed 38
```

--multiple-seeds is canonical; --multiple_seeds and --multseeds are aliases. --seed-count and --seed_numbers set the count (default 5, range 1-64). --seed-seconds/--seed_seconds set the positive trial ceiling (default 7). Supplying either valued option also enables the strategy.

Total time --seconds is the total allowance, not time per seed plus another full final run. Each finite trial is capped by $\min(\text{trial ceiling}, 0.7 * \text{total} / \text{count})$. With 60 seconds, five seven-second trials normally leave about 25 seconds; with default 20 seconds, five trials are capped near 2.8 seconds each. Preparation and checks may modestly overrun.

Continuation and protection Remaining time is used to improve the selected search. A better verified result from an earlier trial is retained if a later phase returns a worse candidate or none. Clock-limited searches can follow different trajectories even with the same seed.

Supported heuristic modes include roots, NLP/MINLP, heuristic MILP, TSP, CVRP, cutting and operational services. Pure LP and deterministic assignment/transport/flow reject the option. MILP with --no-mip-heuristics also rejects it. Small exact cases may stop after one trial.

More seeds trade improvement time for diversity; they do not guarantee a better result than one strong seed.

Multi-seed commands and Python calls

Crew

```
optimizer.exe --crew --file crew.txt --multiple-seeds --seed-count 5 --seed-seconds 7 --seconds 60
--seed 38 --progress
```

Nonlinear optimization

```
optimizer.exe --file nonlinear.txt --multseeds --seed_numbers 5 --seconds 30 --seed 38
```

City tour

```
optimizer.exe --tsp --file cities.tsp --multseeds --seed_numbers 5 --seconds 30 --seed 38
```

Equation roots

```
optimizer.exe --file roots.txt --multseeds --seed_numbers 5 --seconds 20 --max-solutions 100
```

The first seed is the supplied --seed, default 42; additional distinct seeds are generated reproducibly from it. Results include the selected seed and trial count. With --progress, verified feasible-result improvements are shown without internal counters or continuation details.

Python

```
r = engine.solve_crew_file("crew.txt", multiple_seeds=True,
    seed_count=5, seed_seconds=7, seconds=60, seed=38)
print(r["selected_seed"], r["seed_trials"])
```

For specialized Python wrappers that do not accept these new keywords, call `run_capture` with the actual CLI list. Do not assume `solve_tsp()`, `solve_cvrp()`, `solve_cutting_stock()` or operational convenience methods forward every global option.

--seconds 0 still gives finite initial trials, followed by a no-clock final phase subject to work caps. Same seeds with different clock allowances can differ. Keep the best complete verified result, not an average or median of objectives from several runs.

28. DLL integration: C, C++, Python, Java and C#

DLL contract: arguments, results and ownership

The supplied `optimizer_dll.h` exposes C-ABI version 1. No C++ object, Python object or STL container crosses the boundary. Windows calls use `__cdecl`. The library is synchronous and captures solver stdout/stderr.

Function	Contract
<code>optimizer_run(argc,argv,&result)</code>	Arguments exclude executable name; <code>argc=0</code> requests help
<code>optimizer_stdout(result)</code>	Borrowed NUL-terminated output string
<code>optimizer_stderr(result)</code>	Borrowed NUL-terminated diagnostic/error string
<code>optimizer_exit_code(result)</code>	Solver status, distinct from API status
<code>optimizer_elapsed_seconds(result)</code>	Elapsed call time as double
<code>optimizer_free(result)</code>	Free exactly once through this library
<code>optimizer_abi_version()</code>	Returns interface version 1
<code>optimizer_has_binary128()</code>	Nonzero when this build supports binary128
<code>optimizer_has_model_import(FO RMAAT)</code>	Optional added capability query; lp, mps, nl, nl-ascii or nl-binary. Unknown formats return zero.

`optimizer_run` returns 0 when the call completed, -1 for invalid arguments, -2 for allocation failure and -3 for an unexpected API failure. On API failure the result pointer is null. A completed call can still contain solver exit 1 or another unsuccessful solver status.

Arguments must be valid NUL-terminated strings for the call duration. Do not embed NUL characters. Copy output before `optimizer_free`; borrowed pointers become invalid afterward. `optimizer_free(NULL)` is safe. Free a result using the same library that created it.

Current interface scope Pass model text using `--eq` or files using `--file` and service-specific switches. The general command interface does not expose teacher/room object methods, progress callbacks or a cancellation callback. Resource placement additionally supports a direct array-based API, described later in this guide. These are not prerequisites for loops and branching in the calling language.

Calls through the current legacy `optimizer_run` interface are serialized. For independent legacy jobs, use separate processes. The additive typed core interface instead supports independent contexts, progress and cooperative cancellation; see Chapter 36. It does not expose every CLI service, typed binary128 or work-checkpoint options. Do not depend on manipulating another application's stdout while a legacy DLL call is running.

Call the DLL from C

The complete `languages/c_call.c` includes the supplied header and checks both status layers. Its core is:

```
const char *args[] = {"--eq", "x^2=4", "--seconds", "5"};
OptimizerResult *result = NULL;
int api = optimizer_run(4, args, &result);
if (api != 0 || result == NULL) {
    fprintf(stderr, "API failure: %d\n", api);
    return 1;
}
fputs(optimizer_stdout(result), stdout);
fputs(optimizer_stderr(result), stderr);
int code = optimizer_exit_code(result);
optimizer_free(result);
return code;
```

No executable name is included. No quotation marks are embedded around $x^2=4$ because this is an argument array, not shell syntax.

Windows: build and run From the companion root, after building the DLL/import library:

```
gcc -std=c11 languages/c_call.c -Isource -L. -loptimizer -o c_call.exe
c_call.exe
```

Keep optimizer.dll and its matching runtime dependencies available to the caller. liboptimizer.dll.a is the MinGW import library for linking; it is not the runtime DLL itself.

Linux

```
gcc -std=c11 languages/c_call.c -Isource -L. -loptimizer -Wl,-rpath,'$ORIGIN' -o c_call
./c_call
```

The Linux command uses a locally built liboptimizer.so. It does not load the Windows binary. Change relative paths to match your actual layout.

For several calls, allocate/free a separate result for each. Never use the host's free() on an OptimizerResult; it is an opaque library-owned object.

Call the DLL from C++ with automatic cleanup

Use the C header even from C++. A smart pointer with optimizer_free as deleter ensures cleanup on normal completion and exceptions.

```
using Result = std::unique_ptr<OptimizerResult,
                             decltype(&optimizer_free)>;
std::vector<std::string> args = {
    "--eq", "maximize x*y; x+y=10; bound x=[0,10]; bound y=[0,10]",
    "--seconds", "5"
};
std::vector<const char*> argv;
for (const auto& s : args) argv.push_back(s.c_str());
OptimizerResult* raw = nullptr;
int api = optimizer_run((int)argv.size(), argv.data(), &raw);
Result result(raw, &optimizer_free);
if (api != 0 || !result) throw std::runtime_error("API failure");
std::string output = optimizer_stdout(result.get());
int code = optimizer_exit_code(result.get());
```

The strings remain alive during the call. Do not mutate/reallocate args while argv points into them. The complete languages/cpp_call.cpp checks embedded NULs and catches errors.

Windows

```
g++ -std=c++17 languages/cpp_call.cpp -Isource -L. -loptimizer -o cpp_call.exe
cpp_call.exe
```

Linux

```
g++ -std=c++17 languages/cpp_call.cpp -Isource -L. -loptimizer -Wl,-rpath,'$ORIGIN' -o cpp_call
```

The example maximizes $x*y$ with $x+y=10$, returning a best objective near 25. Extract an application's structured values using its adapter or preserve the captured report. The ABI itself returns text and metadata, not a C++ vector of decision values.

Python DLL quick start and result dictionary

Use source/optimizer_ctypes.py with matching optimizer.dll. It needs no pybind11, NumPy or pandas for ordinary calls. The class is OptimizerDLL. Put the adapter beside your script or add source/ to sys.path. The supplied runnable callers do this themselves.

```
from optimizer_ctypes import OptimizerDLL
engine = OptimizerDLL("optimizer.dll")
```

```
r = engine.solve(["x*y=8", "x+y=6"], seconds=5)
if r["feasible"]:
    for point in r["candidates"]:
        print(point["values"])
else:
    print(r["message"])
```

solve accepts a model string or iterable of statement strings. solve_file accepts a path. solve_linear/solve_linear_file force affine handling. solve_crew/solve_crew_file select supplied patterns.

Result fields exit_code, stdout, stderr and elapsed_seconds preserve call information. feasible indicates an extracted candidate. success combines successful exit with a candidate. candidates contains solution dictionaries; best is the first or None. values is the best candidate's variable mapping.

Candidate fields include values, values_text, objective, objective_text, constraint_violation and its text, and first_order_converged where applicable. Mode-specific routes/patterns/assignments/plan records may also be present. best_bound, gaps, status and optimality fields should be interpreted according to the service.

Printing versus structured use run(args) prints the captured output and returns an exit code. run_capture(args) returns the structured dictionary. solve() returns data; a script must explicitly print or otherwise use it. Calling a DLL does not automatically display its report.

CLI-like invocation is also available:

```
python source/optimizer_ctypes.py optimizer.dll --eq "x^2=4"
```

Own text parsers must accept the public labels Precision:, Variables: and Model type:. No .pyd compiled for one particular Python minor version is involved. Use the updated adapter with the rebuilt DLL; it also accepts older report labels. backend contains a public model label for new reports. seed_continuation is None when that optional detail is omitted.

Python loops, branching and extended digits

Python controls loops and if statements outside the native model. Each call may contain a newly generated model.

```
for total in (6,8,10):
    model = f"maximize x*y; x+y={total}; bound x=[0,{total}]; bound y=[0,{total}]"
    r = engine.solve(model, seconds=2)
    if r["feasible"]:
        print(total, r["best"]["objective"], r["values"])
    else:
        print("No accepted result", r["message"])
```

The exact maxima are 9, 16 and 25. The complete example is languages/python_sweep.py. You can obtain totals and coefficients from CSV, SQL or application objects before generating the model.

Retain high precision

```
from decimal import Decimal
r = engine.solve("x^2=2; bound x=[0,2]",
                 extend=True, tol="1e-24", seconds=5)
if r["feasible"]:
    x = Decimal(r["best"]["values_text"]["x"])
    print(x)
```

The ordinary values field converts many decimal outputs to Python float, losing binary128-level digits. values_text preserves printed text. Decimal preserves those digits when constructed from the string; set an adequate Decimal context for subsequent arithmetic.

Option mapping General valued keywords use underscores: max_solutions, integer_neighbors, seed_count and so on. bounds={"x":(0,5)} supplies per-variable CLI bounds; global_bounds=(0,5) supplies a global override; integer=["x","y"] declares integers. extra_args is a list

of additional argument strings, not a shell command.

Use `raise_on_error=False` to receive input-error data rather than `ValueError`. Always test feasible before dereferencing best.

Python: operational services and plans

The adapter provides named methods for each supported operational/network service. Text methods take records as a string; file methods append `_file` and take a path.

Text method	Service
<code>solve_assignment</code>	Worker-job matching
<code>solve_transportation</code>	Balanced shipment allocation
<code>solve_mincost_flow</code>	Bounded network flow
<code>solve_schedule</code>	Job/machine scheduling
<code>solve_vrptw</code>	Window-constrained delivery
<code>solve_roster</code>	Employee shifts
<code>solve_timetable</code>	School periods
<code>solve_bin_packing</code>	Homogeneous 1D bins
<code>solve_packing2d</code>	Rectangular sheets
<code>solve_cutting_multi</code>	Multiple stock types

```
r = engine.solve_roster_file("models/rostering.txt", seconds=5)
if r["feasible"]:
    for record in r["best"].get("plan", []):
        print(record)
```

Service plan records have kind/id fields and service-specific fields, many preserved as strings. Convert quantities using the documented units; do not assume every field is a Python numeric scalar. TSP supplies tour, CVRP supplies routes/load/cost data, homogeneous cutting supplies patterns/production, assignment supplies assignments and flow supplies flows.

File-oriented methods `solve_tsp`, `solve_cvrp` and `solve_cutting_stock` take their instance path. Their keyword sets differ from generic solve: for example pool is the TSP keyword, not `tsp_pool`. See the city-tour option reference and the companion examples.

For global settings not accepted by a convenience method, use `run_capture` with the actual supported CLI arguments. Convenience methods intentionally reject unknown options rather than silently ignoring them.

Python resource placement without an input file

Use the matching adapter and DLL for the optional direct placement API:

```
from optimizer_ctypes import OptimizerDLL
engine = OptimizerDLL("optimizer.dll")
result = engine.place(
    demands=[[4, 6], [4, 8]],
    capacities=[[8, 16], [8, 16]],
    costs=[[2, 5], [0, 0]],
    initial=[1, -1],
    migration=[1, 0],
    groups=[0, 0],
    seconds=5, seed=42
)
print(result["assignment"])
print(result["objective"])
```

Rows in demands are objects; rows in capacities are bins. Resource columns use the same order. costs has one row per object and one column per bin. The returned assignment uses zero-based bin indices: [0,1] means object 0 goes to bin 0 and object 1 to bin 1.

initial uses -1 when unspecified; groups uses -1 for no group. Equal nonnegative group IDs require different bins. Optional allowed lists restrict bins per object. sparse_costs can replace dense costs with object/bin/cost triples; missing costs use zero or the object's default_cost. Do

not supply dense and sparse costs together.

Python loops and if statements can build these lists from your own data. Inputs are copied for the call; this is not a zero-copy promise. Read `objective`, `constraint_violation`, `elapsed_seconds` and `finite_search_complete`. Completion of a finite search is not an exact-arithmetic optimality proof. Older DLLs may lack these optional placement symbols; use the matching distribution.

Python linear model builder

`LinearModel` from `optimizer_models.py` provides a convenient affine modeling interface. Variables may have original human-readable names; export safely aliases them and solve remaps values.

```
from optimizer_models import LinearModel
m = LinearModel("production")
m.variable("Product A", lower=0, upper=8, kind="integer")
m.variable("Product B", lower=0, upper=8, kind="integer")
m.set_objective({"Product A":5,"Product B":4}, maximize=True)
m.constraint({"Product A":2,"Product B":1}, "<=", 8)
m.constraint({"Product A":1,"Product B":2}, "<=", 8)
r = m.solve(engine, seconds=5)
print(r["values"])
```

The expected values are Product A=3 and Product B=2, objective 23. The complete caller is `languages/python_builder.py`.

Builder methods `variable(name,lower=None,upper=None,kind='continuous')` creates a decision. Supported kinds are continuous, integer and binary. Bounds default free, matching native LP semantics. Pass `lower=0` for conventional nonnegative quantities.

`constraint(terms,sense,rhs,name=None)` takes a name-to-coefficient mapping and `=`, `<=` or `>=`. `set_objective(terms,maximize=False,constant=0)` sets one affine objective. `export()` returns native text and alias-to-original mapping. `write(path)` writes the model. `solve(engine,...)` calls native linear solving and remaps candidate values.

Coefficients may be finite strings or `Decimal` to preserve intended literals. This is an affine builder, not a nonlinear expression-overloading package. Use explicit supported text for variable products or functions.

For a database-driven application, construct dictionaries from query results, validate names and units, then call the builder. You need not expose generated text files to the end user.

Java: call the C ABI using JNA

Use Java Native Access (JNA) to map the plain C functions. Add a compatible `jna.jar` to your classpath; Java and the DLL must have matching architecture. The supplied `languages/OptimizerExample.java` contains the complete caller.

```
public interface Engine extends Library {
    int optimizer_run(int argc, String[] argv,
        PointerByReference result);
    Pointer optimizer_stdout(Pointer result);
    Pointer optimizer_stderr(Pointer result);
    int optimizer_exit_code(Pointer result);
    void optimizer_free(Pointer result);
}
```

Load with `Native.load(fullPath,Engine.class,options)`. Set `Library.OPTION_STRING_ENCODING` to UTF-8. Use `Library`, not a Windows `stdcall` mapping, because this API uses `cdecl`.

Windows example Place your JNA jar as `jna.jar` at companion root:

```
javac -cp jna.jar languages/OptimizerExample.java
java -cp "jna.jar;languages" OptimizerExample "C:\optimizer\optimizer.dll"
```

On Linux, use `'.'` in the classpath and a native `liboptimizer.so` path. Ensure its dependent shared

libraries can be resolved.

Read `PointerByReference.getValue()`, check the API return, then read output with `getString(0,"UTF-8")`. Free the result in finally after extracting all values. Java garbage collection does not free an opaque native result automatically.

The method receives separate argument strings without an executable name. You can use Java loops and collections to create the --eq text or service records. No JNI bridge code is needed for this JNA example, but JNA is an external calling-library dependency.

Reference: the official JNA Getting Started, Library API and Functional Description documentation describe these mappings and native loading.

C# / .NET: explicit UTF-8 interop

The companion languages/OptimizerExample.cs declares P/Invoke functions with `CallingConvention.Cdecl`, allocates UTF-8 argument strings and a pointer array, and frees both host allocations and the library-owned result correctly.

```
[DllImport("optimizer.dll", CallingConvention=CallingConvention.Cdecl)]
static extern int optimizer_run(int argc, IntPtr argv,
                              out IntPtr result);
[DllImport("optimizer.dll", CallingConvention=CallingConvention.Cdecl)]
static extern void optimizer_free(IntPtr result);
```

Use an x64 .NET process with the x64 DLL. Place `optimizer.dll` and its runtime dependencies where the application loader can find them. The complete sample's library name targets Windows.

Build with a suitable .NET SDK

```
dotnet new console -o DotNetExample
copy languages\OptimizerExample.cs DotNetExample\Program.cs
dotnet build DotNetExample -p:PlatformTarget=x64
```

Copy the DLL and dependencies beside the generated executable. Run that application according to your SDK's output layout. Do not assume the DLL search directory is the source directory.

Ownership rules `Marshal.AllocHGlobal` allocations are freed with `Marshal.FreeHGlobal`. The `OptimizerResult` is freed only by `optimizer_free`. Borrowed `stdout/stderr` pointers must be read before freeing the result. The sample decodes UTF-8 explicitly rather than relying on an unspecified ANSI encoding.

For another language, follow the same contract: `cdecl`, integer/double types, an argument pointer array, opaque result pointer, two status layers and library-owned cleanup. Never map the returned object as an application struct containing guessed fields.

Direct placement API for C, C++, Java and C

The public header supplies `OptimizerPlacementModel`, `OptimizerPlacementOptions` and `optimizer_place` in addition to `optimizer_run`. The existing command interface remains available for every service; use it from the complete C, C++, Java/JNA and C# companion callers.

For direct placement, fill `struct_size` with `sizeof` the matching structure. Supply `object_count`, `bin_count` and `resource_count`. Flatten demands in object-major order and capacities in bin-major order. Each resource value is a double. Provide the corresponding array-length fields exactly as specified in `optimizer_dll.h`.

Optional dense costs are object-major with `object_count*bin_count` entries. Alternatively supply sparse object/bin/value arrays. Eligibility uses the documented offsets and bin-index arrays; an empty eligibility row is invalid. initial and groups use -1 for unspecified values. All indices are zero-based.

`optimizer_place` borrows input arrays for the duration of the call. It returns an owned `Optimizer-`

Result. Inspect API status separately from `optimizer_exit_code`. Copy `optimizer_placement_bins` before `optimizer_free`; its pointer is borrowed. Use `optimizer_placement_count` to check that a placement exists before interpreting its objective and violation.

C and C++ use the supplied header directly. Java/JNA and C# require matching native structure layout, pointer widths, integer widths and calling convention; do not guess that layout from a textual example. For most applications, the documented `optimizer_run` interface is simpler, and the supplied Python place wrapper already handles the placement structures.

Independent direct placement calls may own separate results. General `optimizer_run` calls remain serialized. The public ABI does not provide live callbacks or cancellation. Keep input buffers alive throughout the synchronous call and free each result exactly once with the creating DLL.

Import files through the same C ABI

No separate DLL export is required to solve LP/MPS/NL. Pass the same options as the executable, without the executable name:

```
const char *args[] = {"--nl", "--file", "model.nl", "--seconds", "20"};
OptimizerResult *result = NULL;
int api_status = optimizer_run(5, args, &result);
if (api_status == 0 && result != NULL) {
    fputs(optimizer_stdout(result), stdout);
    fputs(optimizer_stderr(result), stderr);
    int solver_status = optimizer_exit_code(result);
    optimizer_free(result);
    /* Handle solver_status in the calling application. */
}
```

The same argument array can be supplied from C++, Python ctypes, Java JNA and C# P/Invoke. AMPL/GAMS source generation remains an external-runtime workflow; the DLL does not compile those languages itself. Feature queries help applications distinguish a current DLL from an older compatible build. The public header is the authoritative declaration of entry points and types.

29. Using the PHP front end

Solve from the browser

The current optimizer.php page offers 18 services, a model editor and two example levels for every service. Input can contain at most 100,000 Unicode characters. Select the service before loading an example; review the record explanations to adapt its numbers and names to your application.

1. Select a service, such as General equations, including conditions or Rectangular packing / cutting.
2. Type a model, use Open File, or choose Load simple example / Load complex example.
3. Choose a finite time limit and the applicable precision/tolerance.
4. Open Advanced options only for controls you need, then click Solve.
5. Read candidate values, constraint violation and objective/bound/gap status. Use Save results for the displayed report.

Save to File saves the model text, not all browser settings. Restore previous input undoes the most recent example/file replacement. Loading an example, opening a file or changing service resets service-specific advanced settings and bound overrides. Manually editing a model retains current advanced settings: use Reset advanced options when the new model needs different settings. Keep physical bounds in the model where possible.

Advanced settings and defaults

Setting	Default	End-user meaning
Total time	20 seconds, subject to the server cap	All seed trials and improvement share this allowance.
Standard / extended tolerance	1e-9 / 1e-24	Required root or constraint accuracy where the service supports it. Custom values are preserved when precision changes.
Random seed	42	Select a repeatable random sequence; clock-limited runs can still differ.
Multi-seed search	Off; 5 seeds when enabled	Survey several trials and retain the best verified result.
Trial ceiling	7 seconds	Reduced as needed to fit the total allowance.
Heuristic global boxes	256	Extra nonlinear region exploration; no objective-bound certificate by itself.
Numerical global optimization	Off	Optional bounded-model gap search, requiring an objective and finite bounds for every variable.
Relative global gap	1e-4	0.01% target when numerical global mode is enabled.
Global work limits	10,000 nodes; 1,000,000 linear iterations	Additional limits besides time.
Maximum root solutions	100 on the page	The executable's default is 500 if no value is supplied.

Controls appear only for compatible services. For example, deterministic assignment and flow do not offer multi-seed trials; routing does not use residual tolerance as a quality target. The command-line option reference explains every control. Blank settings normally leave the solver's default in effect.

New command controls and a PHP front end

This guide update does not automatically add web controls for every new CLI option. Verify the exact binary path, complete argv, input model and precision that PHP launches. Work continuation needs an explicit positive work limit and managed checkpoint identity/path; it is not enabled by an ordinary timed benchmark call. Add frontend support deliberately rather than assuming a checkbox or session value already maps to the new interface. Chapter 34 provides command examples and exit-2 continuation handling. Keep user jobs isolated and preserve their exact compatible settings.

Files supported by this front end

This page submits native model text or supported service-instance text using --MODE --file PATH (equations use --file without a mode). It opens valid UTF-8 text and rejects binary data. The new CLI/DLL LP/MPS/NL import support does not imply that the PHP page compiles AMPL/GAMS or selects exchange-format importers. Use the executable or the Python file-import methods for LP/MPS/NL and modeling-language workflows documented in the import chapters.

Install the page

Use PHP 8.0 or later. Place optimizer.php beside the matching executable, or set OPTIMIZER_PATH on the server. Make PHP sessions, temporary-file creation and process execution available to the service account. Set request timeouts long enough for the allowed solve plus cleanup.

```
OPTIMIZER_PATH=C:\Patodia\optimizer.exe  
OPTIMIZER_TEMP_DIR=C:\Patodia\optimizer_web_temp  
OPTIMIZER_MAX_SECONDS=120
```

The server time cap defaults to 120 seconds and is limited to 1-600. The browser form requires a finite allowance of at least 0.01 seconds. On Linux use a native Linux executable, not the Windows .exe. The page's deployment package contains its own full setup instructions and tests.

30. Troubleshooting and model checks

Troubleshooting and final acceptance checklist

Unknown option: use the help for the selected mode. --starts is not --service-starts or --tsp-starts. The legacy precision spelling is accepted, but new commands should use --extend.

Unknown variable in --bound: remove bounds belonging to another model or correct the variable name. Bounds are case-sensitive and must refer to detected unknowns. Constant parameters are not unknown variables.

No output from Python: solve() returns data. Print its fields or use run() to display a report. Check feasible and message before reading best.

DLL will not load: confirm full path, architecture and matching runtime dependencies. Python can use OPTIMIZER_DLL_DIR or the dll_directory constructor argument. A version-specific .pyd is not required.

No feasible result: check domains, contradictory hard rules, total demand, eligibility and timing units before increasing search. A limited failure is usually weaker than a proof of infeasibility.

Tighter tolerance loses candidates: the acceptance test became stricter. Increase time or use supported extended precision where appropriate, but choose tolerances justified by data and conditioning.

Different seeds/runtimes differ: heuristic results are not always deterministic under clock limits. Retain the best verified candidate and compare equivalent budgets.

Before using a plan Check complete coverage, original hard constraints, domains, units and objective. Read the stopping/optimality status. Preserve input, options, result and name mapping. Use the companion tests as installation smoke checks, not as proof for every new problem.

Constant errors: a missing initializer identifies its name; duplicate definitions must be removed; circular dependencies must be broken. Use constant a=3, b=2; for multiple initializers. Constants cannot depend on unknown decision variables. Check function domains if a constant has no finite real value.

Imported-model troubleshooting

Symptom	What to check
LP file treated as native equations AMPL/GAMS language keywords rejected	Use --input-format lp --file PATH, especially if its extension is .txt. Use the runtime bridge; --file does not compile .mod or .gms source.
Unsupported imported operation	Read the named limitation. Reformulate/export supported mathematics; do not remove a hard constraint merely to silence the error.
NL variables named v0, v1, ... Multiple NL objectives	Keep the matching .col sidecar with the export to preserve names. Select --objective-number N, with a zero-based index, or export one active objective.
High-precision results lose digits in Python	Read values_text / objective_text and construct Decimal from the text.
API status zero but solve failed	Inspect the separate exit_code, feasible, message and applicable optimality status.
New Python method fails on an old DLL	Rebuild/deploy the executable and DLL supporting imports; replacing the adapter alone does not add a parser.
AMPL cannot load the result	Keep model/presolve state consistent with the exported NL; inspect the solution message and variable count.
GAMS results not present in .l/.m	The bridge returns Python data; it is not a registered GAMS solver link.

Keep an auditable run

Save the original input/data, exact options, executable/DLL release, seed, returned candidate, feasibility, bound/gap and stopping reason. For imports also retain the generated NL/name sidecars when feasible. Do not compare elapsed time from one workflow with model-plus-call time from another without labeling both measurements.

31. Complete command-line option reference

The original service tables are retained below. Additional fully scoped option tables appear in Chapters 34-35 and the consolidated new-option index in Chapter 38. Multiobjective mode-record fields are described in Chapter 32; they are not CLI flags. Defaults are mode-specific, and legacy TSP tuning flags select a different policy from automatic TSP.

Option reference: input, objectives and shared settings

Options consume the stated number of following arguments. A quoted expression is one argument. Defaults depend on service; explicitly supplied values replace them. Do not combine incompatible input modes.

Option	Meaning, default and permitted use
--file PATH	Read input; - reads stdin where supported. With settings, prefer explicit --file. TSP/CVRP/cutting can recognize file type.
--eq TEXT; -e TEXT	Append mathematical input; -e aliases --eq for mathematical models. Operational/network text uses --eq (one input, not arbitrary repeated -e).
--minimize TEXT; --maximize TEXT	Set one general-model objective expression. Only one objective. Dedicated service objectives are selected by their own records.
--int NAMES; --integer NAMES	Declare comma-separated unrestricted integer names in mathematical input. Binary/signed forms belong in model statements.
--bound NAME LOW HIGH	Override one mathematical variable interval. Three values; inclusive finite endpoints. Intersects declaration domain.
--bounds LOW HIGH	Override all mathematical variable bounds. Two values; per-name --bound has precedence. Use cautiously.
--seconds S	Nonnegative approximate wall-time cap; 0 disables clock. Usually 20; TSP/CVRP/homogeneous cutting 30; multi-seed total default 20.
--tol T	Positive final root/constraint tolerance. Usually 1e-9; supported extended mathematical/network modes 1e-24. Not a TSP/CVRP option.
--ctol T	Alias for constraint feasibility --tol in general optimization/LP/network/crew. Accepted by condition-bearing equations; not by operational record services.
--extend	Binary128 in supported builds/modes. TSP/CVRP accept without quad refinement; operational services reject.
--extended; --extented	Legacy aliases for --extend, including the older spelling. Prefer --extend in new commands.
--seed N	Nonnegative 64-bit pseudorandom seed, usually 42. Root/NLP/crew/routing/operational/cutting; single-seed native MILP uses --mip-seed.
--progress	Show discoveries and verified feasible-result updates on stderr, without internal counters. DLL output is available after the call; this is not a callback.

Quote constraints with < or > in shell commands. --bounds and --bound take numeric CLI values, not the file's square-bracket syntax. File endpoints may use declared constant expressions.

In default equation mode with >= or <= conditions, --ctol is an alias for the root tolerance --tol. Explicit optimization modes retain their documented constraint-tolerance semantics.

Option reference: equation/root search

These settings apply to equation-only input. Do not send nonlinear objective controls to root mode. Automatic boxes expand only for variables without explicit fixed bounds.

Option	Meaning, default and permitted use
--starts N	Positive starting-point budget [12000]. When automatic bounds use several stages, starts must be at least stages.

Option	Meaning, default and permitted use
--iterations N	Positive maximum local iterations per starting point [300]. More iterations are useful when local convergence is the limiting factor.
--max-solutions N	Positive retained root limit [500]. A limit is not a requested guaranteed count.
--merge T	Positive relative real-coordinate deduplication tolerance [1e-6]. Different integer assignments remain distinct.
--radius R	Positive first automatic radius [10]. Does not widen explicit bounds.
--stages N	Positive number of automatic expansion stages [4]. Explicitly bounded coordinates stay fixed.
--growth G	Automatic radius multiplier, greater than 1 [10]. Default nominal radii are 10,100,1000,10000.
--integer-neighbors N	Nonnegative extra neighboring integer assignments [8]. Zero uses only the nearest integer assignment per relevant start.
--single-stage	Disable coarse discovery for a comparison/reference run. Despite its name, this does not replace --stages for range expansion.
--no-deflation	Disable avoidance of already discovered solution vectors. Original equations are always used for final checks.
--no-inference	Disable inferred domain/range and seed hints. Explicit user bounds/declarations remain hard.
--no-slices	Disable zero/small-integer coordinate-slice exploration.
--solver-help	Show equation grammar, functions, domains and root options; no model needed.
--solver-self-test	Run embedded equation regression checks; a diagnostic command rather than a model setting.

--tol supplies final residual acceptance, default 1e-9 or 1e-24 under --extend. --ctol is also accepted for root models containing inclusive conditions. --integer-nodes, --integer-radius and --integer-enumeration belong to nonlinear optimization, not this root option parser.

Option reference: general nonlinear and MINLP search

These controls apply to general objectives or nonlinear feasibility. Native LP/MILP rejects root/nonlinear search controls when forced. Some controls are shared with equations but have different defaults.

Option	Meaning, default and permitted use
--sqp	Force the nonlinear/mixed-integer heuristic service, including for an affine comparison. Cannot combine with --linear or --crew.
--starts N	Positive restart cap [128]. Stops can occur earlier due to time or other work limits.
--iterations N	Positive local iteration cap per start [300].
--gtol T	Positive scaled first-order/stationarity tolerance [1e-6 double; 1e-18 extended]. Not a global-optimality test.
--ftol T	Positive relative objective stagnation tolerance [1e-12 double; 1e-28 extended]. Separate from feasibility.
--max-solutions N	Positive number of distinct feasible candidates retained/displayed [1]. Not a count of proven optima.
--merge T	Positive relative candidate deduplication tolerance [1e-6].
--integer-neighbors N	Nonnegative integer neighbor repair allowance [8]. Integer final validation remains mandatory.
--integer-enumeration N	Small-box enumeration cap [256], nonnegative; zero disables this enumeration. Larger boxes use heuristic search.
--integer-nodes N	Extra affine integer seed-search cap per stage [200000]; zero disables that search.
--integer-radius N	Free-integer seed L1 radius [16], range 0..1000000. Not a global integer bound.
--radius R; --stages N; --growth G	Automatic search boxes [10;4;10]. Radius positive, stages positive, growth >1; explicit bounds stay fixed.
--global-search	Enable ordinary bounded-region exploration. Default: automatically enabled where applicable to continuous nonlinear models; no certified bound.
--global-boxes N	Exploration budget [256], integer 0..100000; supplying it enables exploration. Zero permits no box subdivisions.
--no-global-search	Disable extra region exploration; preserve the same objective and hard constraints.
--global-opt	Enable numerical global optimization; objective and explicit finite bounds required. Off by default.
--global-gap R	Nonnegative relative gap target [1e-4]; also enables numerical global mode.
--global-abs-gap A	Nonnegative absolute gap target [0], in objective units; also enables numerical global mode.

Option	Meaning, default and permitted use
--global-nodes N	Positive numerical global work limit [10000]; also enables numerical global mode.
--global-pivots N	Positive numerical global linear-work limit [1000000]; also enables numerical global mode.
--global-help	Explain numerical global support, domain requirements and examples.
--no-inference	Disable sampling/domain hints. Does not remove supplied hard domains.
--single-stage; --no-deflation; --no-slices	Root comparison flags; accepted but ignored in general SQP optimization. Do not use them as NLP tuning promises.

--tol/--ctol sets original feasibility; --seconds sets total approximate time. Use a realistic tolerance and bounds before adding exploration. Extended accuracy does not establish global or mixed-integer optimality.

Option reference: native LP/MILP

Linear settings apply to native affine models and supported linear LP/MPS/NL imports. They do not select a valid linear representation for a nonlinear expression.

Option	Meaning, default and permitted use
--linear; --lp; --milp	Force affine LP/MILP; reject nonlinear terms. Integer declarations, not the alias spelling, determine domains.
--lp-pivots N	Nonnegative total numerical pivot cap [1000000]. Zero permits no pivots; may leave unresolved status.
--mip-nodes N	Nonnegative integer-search node cap [100000]. Zero permits no nodes; not a request for a proof.
--mip-seed N	Nonnegative 64-bit heuristic seed [42]. Use this in ordinary native linear runs instead of root/NLP --seed.
--mip-heuristic-moves N	Nonnegative feasible-candidate repair move allowance per call [20000].
--mip-dive-nodes N	Restricted candidate-search node allowance per call [32], range 0..10000. Zero disables that work.
--no-mip-heuristics	Disable automatic primal heuristic aids. Native multi-seed rejects this combination.
--mip-gap R	Nonnegative relative objective gap target [0]. Denominator $\max(1, \text{abs}(\text{incumbent}))$; requires a valid bound.
--mip-abs-gap A	Nonnegative absolute objective gap target [0], in objective units.
--mip-int-tol T	Integer recognition tolerance [1e-7 double; 1e-24 extended], strictly between 0 and 0.5. Rounded candidates are verified.
--mip-clique-cuts	Opt-in eligible root packing/partition tightening. No guaranteed bound or incumbent improvement.
--mip-clique-rounds N	Root rounds [2], range 0..16. Supplying this also enables the optional cuts.
--mip-clique-limit N	New cuts per round [32], range 0..128. Supplying this also enables optional cuts.
--lp-help; --lp-self-test	Display linear help or run its diagnostic regression checks; --extend can be used for applicable tests.

--seconds, --tol/--ctol, --bound/--bounds, --extend and --progress are supported. Native variables default free. Root/NLP --starts, gradients and range controls do not tune the native LP service.

Option reference: supplied binary pattern selection

Use these only with --crew. Scope is binary exact-one coverage minimization/feasibility, not an arbitrary binary LP.

Option	Meaning, default and permitted use
--crew	Select supplied-pattern service. Explicit; default mathematical dispatch is unchanged.
--crew-starts N	Initial feasible-seed target [50], range 1..100000. Search can finish at a clock/work limit first.
--crew-pool N	Retained distinct elite schedules [8], range 1..64.
--crew-moves N	Nonnegative total move-attempt cap [2000000]. Zero permits no such improvement moves.
--crew-neighborhood N	Maximum selected patterns released for repair [24], range 2..64. Larger repair can cost more time.

Option	Meaning, default and permitted use
--crew-block-nodes N	Nonnegative work cap for each local repair [20000]. Zero disables that repair.
--crew-connected	Enable optional connected/adaptive repair, default off. Compare against the original search on your data.
--crew-help	Display accepted model scope, settings and an example.
--crew-self-test	Run supplied-pattern diagnostic checks; not a production-model flag.

Shared settings: --seconds [20], --seed [42], --tol/--ctol [1e-9 or extended 1e-24], --extend, --bound/--bounds and --progress. Do not combine with --linear, --sqp or their dedicated work controls.

For diversification, add --multseeds --seed_numbers 5 --seconds 60. Returned coverage and fixed domains remain mandatory. A feasible selection plus an independently evaluated bound can quantify a gap; a limited heuristic run does not certify integer optimality.

Option reference: city tours / TSP

Use a symmetric TSPLIB file. TSP defaults to a 30-second allowance; --extend is accepted without binary128 refinement. Generic residual/gradient tolerances are rejected.

Option	Meaning, default and permitted use
--tsp --file PATH	Select dedicated city tour input; TYPE TSP. --file can auto-detect supported files.
--tsp-pool N	Rank construction tours [0=off], range 0..1000 and at most city count. Extra starts share the same time.
--tsp-elite N	Top-ranked starts to improve [1]. Positive, <=pool when enabled; without pool must be 1.
--tsp-depth N	Compound depth: 0 disables or 2..16. Default 8 with pool, 0 without; applicable to tours >200 cities.
--tsp-starts N	Without-pool construction cap [8], range 1..1000. Pool replaces that survey.
--tsp-neighbors K	Candidate breadth [64], range 2..256. Tours <=200 use all neighbors.
--tsp-iterations N	Nonnegative explicit iteration cap. Small default 2000; large defaults time-driven, or 500 with seconds=0.
--tsp-stall N	Positive nonimproving-iteration cap. Small default 400; large time-driven, or 64 with seconds=0.
--tsp-patch N	Fragment improvement size [0=off]; otherwise 3..10, applicable to large tours. Not a global proof.
--tsp-passes N	Positive local sweeps per iteration [100].
--tsp-3opt K	Breadth [24], range 0..32; zero disables. Applicable to tours >200 cities.
--tsp-exact N	Exact-small-instance threshold [16], range 0..20; zero disables.
--tsp-target C	Nonnegative feasible cost target; omitted means no target. Meeting it does not establish optimality.
--tsp-help; --tsp-self-test	Display tour options or run tour diagnostic checks.

Python solve_tsp(path,...) keywords: seconds, seed, pool, elite, depth, starts, neighbors, iterations, stall, patch, passes, threeopt, exact, target, progress, extend, raise_on_error. Use run_capture for native global multi-seed arguments.

Option reference: capacity routing / CVRP

Use one-depot homogeneous-capacity VRPLIB input. --seconds defaults to 30. --extend is accepted but is not quad route refinement. General nonlinear settings are not routing controls.

Option	Meaning, default and permitted use
--cvrp --file PATH	Select native capacity-routing file; --file also auto-detects supported CVRP.
--vehicles K	Positive maximum available fleet, <=1000000 before effective customer-count cap. Overrides file VEHICLES. Omitted uses file or one per customer.
--cvrp-starts N	Construction/local-search start allowance [50], range 1..10000.
--cvrp-neighbors K	Candidate breadth [40], range 2..256. <=300 customers use all neighbors.
--cvrp-iterations N	Nonnegative explicit improvement cap; normally time-driven. With seconds=0 the finite default is 2000.
--cvrp-ruin N	Customers released per attempt [0=adaptive], range 0..100000. More is not necessarily better.
--cvrp-target C	Nonnegative feasible cost threshold; omitted disables target. No proof from reaching it.

Option	Meaning, default and permitted use
--cvrp-help	Display capacity/fleet semantics, formats, options and limitations.
--cvrp-self-test	Run capacity, coverage and route diagnostic checks.

Shared --seed [42], --seconds and --progress are supported. Native multi-seed is supported. No --cvrp-exact user option is supported in this release.

Python `solve_cvrp(path,...)` keywords: `seconds`, `seed`, `vehicles`, `starts`, `neighbors`, `iterations`, `ruin`, `target`, `progress`, `extend` and `raise_on_error`. For multi-seed use `run_capture` with the actual CLI argument list.

A lower route cost is not useful if the fleet or capacity model is wrong. Vehicle counts in file-names are never inferred. Time windows, service times and heterogeneous fleets belong in VRPTW, not arbitrary added CVRP sections.

Option reference: homogeneous 1D cutting

Use --cutting-stock with one stock length. Default --seconds is 30; --seed is 42. Mathematical --tol and arbitrary operational settings are not cut controls.

Option	Meaning, default and permitted use
--cutting-stock --file PATH	Select homogeneous .cstock input; TYPE: CUTTING_STOCK auto-detection also works through --file.
--cut-starts N	Initial diversified packing starts [50], range 1..10000.
--cut-iterations N	Additional pattern-search cap [1000], range 0..100000. Zero limits that work.
--cut-patterns N	Retained pattern cap [6000], range 1..20000 and at least item-type count.
--cut-nodes N	Packing/completion/integer-plan work cap [100000], range 0..10000000. Zero disables those improvements.
--cut-enumerate N	Supported completion pattern cap per node [20000], range 0..100000. Zero disables completion.
--cut-pricing-cells N	Normalized size work allowance [2000000], range 1..5000000. Larger values may require more resources.
--cut-help; --cut-self-test	Display homogeneous cutting formats/options or run diagnostic checks.

--extend supports extended numerical work; original lengths and counts are verified in scaled exact units. --progress is optional. Native multi-seed may be used; choose a total budget rather than assuming every seed receives a full --seconds allowance.

Python `solve_cutting_stock(path,...)` keywords: `seconds`, `seed`, `starts`, `iterations`, `patterns`, `nodes`, `enumerate`, `pricing_cells`, `progress`, `extend` and `raise_on_error`. Global multi-seed can be passed via `run_capture`.

File MAX_STOCK is a hard limit, not a CLI work cap. DEMAND_MODE EXACT and AT_LEAST have different production semantics. For prices/multiple lengths use the multistock service and its operational controls.

Option reference: operational, network and import services

Dedicated formats reject unsupported options. These selectors and controls do not accept arbitrary general-model declarations in place of their required records.

Option	Meaning, default and permitted use
--schedule; --vrptw	Select TYPE SCHEDULING or TYPE VRPTW; use --file or one --eq text input.
--roster; --timetable	Select TYPE ROSTERING or TYPE TIMETABLE; mandatory complete assignments.
--bin-packing; --packing2d	Select TYPE BIN_PACKING or TYPE PACKING2D.
--cutting-stock-multi	Select TYPE CUTTING_STOCK_MULTI; costs, lengths and stock inventories are record fields.
--service-starts N	Operational restart cap [32], range 1..10000.
--service-iterations N	Operational nonnegative move cap [200000], maximum 100000000. Clock limit remains separate.

Option	Meaning, default and permitted use
--tol T (operational)	Positive and $\leq 1e-3$ [$1e-9$]. Absolute VRPTW feasibility; does not loosen fixed exact hard quantities in other services.
--assignment	Select rectangular minimum-cost worker/job matching.
--transport; --transportation	Select balanced shipment allocation; second spelling is an alias.
--mincost-flow	Select finite-capacity network flow, balance outgoing minus incoming.
--network-iterations N	Dedicated network nonnegative work cap [1000000]. Zero permits no such optimization iterations.
--input-format FORMAT	Select auto, lp, mps or nl; use with one --file PATH. auto is the default.
--lp-import; --mps; --nl	Select one LP, MPS or NL importer with --file PATH.
--lp-file PATH; --mps-file PATH; --nl-file PATH	Format selector and path together; do not add a second --file.
--objective-number N	Zero-based NL objective index; needed for multiple objectives.
--sol-file PATH	Write an AMPL-format solution for NL; cannot overwrite input/name sidecars.
--convert-only	With LP/MPS/NL input, print native conversion without solving. Cannot combine with solve settings.

Operational defaults: seconds=20, seed=42, progress off; --extend/--ctol rejected. Network defaults: seconds=20, tol=1e-9 (extended 1e-24); --extend and --ctol supported; seed/multi-seed rejected. LP/MPS and affine NL use applicable linear controls. Nonlinear NL uses applicable non-linear controls. LP/MPS default domains differ from native free variables; preserve the imported bounds.

Option reference: multi-seed, help and diagnostics

Help requires no input. Use the help matching your service. Diagnostics are optional installation checks, not model options or evidence that a new production model is optimal.

Option	Meaning, default and permitted use
--multiple-seeds; --multiple_seeds; --multseeds	Enable native seed trials for supported heuristic modes. Off by default.
--seed-count N; --seed_numbers N	Number of seeds [5], range 1..64. Supplying it also enables trials.
--seed-seconds S; --seed_seconds S	Positive finite trial-time ceiling [7]. Supplying it also enables trials. Total --seconds default 20; trials share $\leq 70\%$.
--multi-seed-help	Show scope, total-budget semantics, aliases and a command example.
--help; -h	Main mathematical optimization help; no arguments also displays help.
--solver-help; --solver-self-test	Equation language/help and equation diagnostic checks.
--lp-help; --lp-self-test	Native LP/MILP usage and checks.
--crew-help; --crew-self-test	Supplied exact-cover patterns, controls and checks.
--tsp-help; --tsp-self-test	Symmetric tours, supported metrics and checks.
--cvrp-help; --cvrp-self-test	Capacity-routing formats, fleet controls and checks.
--cut-help; --cut-self-test	Homogeneous cutting conventions, settings and checks.
--services-help; --services-self-test	Scheduling, VRPTW, staff, school, bins, 2D and multistock records/checks.
--network-help; --network-self-test	Assignment, transport and network-flow records/checks.
--import-help	Supported LP/MPS/NL subsets and conversion conventions.
--self-test	Run general optimizer diagnostic checks. --extend is supported for applicable mathematical tests.

For multi-seed, --seed is the first/master seed [42]; --seconds is TOTAL time [20], 0 disables overall clock. Default single-seed behavior remains unchanged without new flags. Constant equation systems run once; deterministic LP/network services reject seed trials.

For a model-specific question, begin with its complete worked example and accepted settings rather than combining switches from unrelated help screens. Save the input and final accepted plan for reproducibility.

--placement-help displays resource-placement records and its accepted options. Use --placement --file PATH for a saved placement model.

Resource-placement option reference

Option	Default and meaning
--placement --file PATH	Select TYPE RESOURCE_PLACEMENT input; --resource-placement is an alias.
--placement-starts N	Construction attempts [1000], positive integer.
--placement-moves N	Improvement allowance [2000000], nonnegative integer.
--placement-neighbors N	Repair neighborhood [8], integer 1..12.
--placement-help	Explain placement records and examples.

Placement supports --seconds [5], --seed [42], --tol/--ctol [1e-9], --progress and native multi-seed search. Its tolerance checks absolute capacity feasibility; it rejects --extend. These controls do not change resource capacities, eligibility, group separation or fixed assignments.

AMPL solver-driver options

These use key=value syntax inside optimizer_options or after STUB -AMPL, rather than the ordinary --flag syntax. Supported numeric keys: seconds, tol, ctol, gtol, ftol, seed, starts, iterations, lp-pivots, mip-nodes, mip-gap, mip-abs-gap, global-gap, global-abs-gap, global-nodes, global-pivots and objective-number. Underscores may replace hyphens. Boolean extend and global-opt take 0 or 1. Other CLI flags are not automatically available through this restricted driver option string.

Defaults are service-specific

Do not send every flag to every model. Native general equations without an objective remain root problems even with $\leq$ or $\geq$ conditions. Native affine objectives may select the linear service automatically; --sqp forces ordinary nonlinear handling. Imported no-objective NL models use feasibility handling. Dedicated services have their own input records and options. Numerical global mode has a narrower function set than ordinary calculator/equation mode.

32. Weighted blending and Pareto sampling

Choose the right multiple-objective method

Several business goals do not automatically mean several independent solves. Choose a method whose meaning matches your decision:

Method	Decision meaning	Supported scope
One traditional objective	Optimize one scalar expression	Existing mathematical and service-specific modes
Weighted blend	Trade goals against one another using explicit positive weights and scales	Native mathematical LP/MILP/NLP/MINLP; up to 64 named objectives
Two-objective Pareto sampling	Return alternative trade-offs for a human/application to choose	Exactly two named mathematical objectives
Priority / lexicographic	Protect the completed higher-priority objective before optimizing the next	Affine LP/MILP only; Chapter 33

These features use the native mathematical grammar. They do not add multiple objectives to TSP, routing, cutting, scheduling or resource-placement record formats. Imported LP/MPS/NL objective vectors are not imported into this controller. Export or construct the supported native mathematical model instead. Do not mix priority fields with blend/Pareto fields, or mix named objectives with a traditional minimize/maximize statement.

A named objective is not an ordinary equation. `objectivecost...minimize...` declares a goal; `cost=...` declares an equation involving a variable called `cost`.

Weighted blending: complete example

Save this as `blend.txt`:

```
multiobjective blend normalization raw
bound x=[0,10]
objective cost weight 1 scale 10 minimize (x-2)^2
objective benefit weight 1 scale 10 maximize x
```

```
optimizer.exe --file blend.txt --seconds 10 --seed 38 --result-json blend.json
```

The controller minimizes the signed aggregate:

```
sum_i weight_i * (sense_i * objective_i - offset_i) / scale_i
sense_i = +1 for minimize, -1 for maximize
```

With raw normalization, offsets are zero. In this example the aggregate is $((x-2)^2 - x)/10$, whose mathematical minimum occurs at $x=2.5$. The individual objective values are $\text{cost}=0.25$ and $\text{benefit}=2.5$; the aggregate is -0.225 . Numerical search may return nearby values within its convergence/feasibility checks. The aggregate is not measured in the original units of either objective.

Weights and scales default to 1 and must be finite and strictly positive. A negative weight is not the way to maximize a goal: use `maximize`. Objective names must be unique identifiers. Declaration order defines objective indices in blend/Pareto output. Use at most 64 objectives.

Scaling matters. A cost measured in rupees can dominate a small dimensionless score unless you choose meaningful scales. Equal weights do not mean equal influence if the numerical ranges differ. Explicit scales retain your intended units and are generally easier to audit than estimated scales.

Automatic normalization

```
multiobjective blend normalization auto
bound x=[0,10]
objective cost weight 1 minimize (x-2)^2
objective benefit weight 1 maximize x
```

Automatic normalization estimates anchor ranges and offsets. Those estimates depend on the anchor solves and may be incomplete under tight budgets or heuristic nonlinear searches. They are not certified global ranges. Do not combine normalizationauto with explicit objective scales. Inspect exported objective metadata and compare the resulting decision against the original units.

A globally solved positive-weight aggregate can support a numerical efficiency conclusion under its assumptions. A local stationary point or a feasible blended result alone does not prove Pareto efficiency or global optimality.

Pareto sampling: complete example

Save as pareto.txt:

```
multiobjective pareto points 5 range 0 10 dominance 1e-7
bound x=[0,10]
objective cost minimize x
objective benefit maximize x
```

```
optimizer.exe --file pareto.txt --seconds 15 --result-json pareto.json --pareto-csv pareto.csv
```

The first objective is the primary scalar objective. The second supplies epsilon thresholds. A minimized second objective receives an upper threshold; a maximized second objective receives a lower threshold. The example samples inexpensive versus high-benefit choices: increasing x raises both cost and benefit. The range is in the second objective's ORIGINAL units, not normalized or negated units.

points is the number of requested threshold-grid positions, not a promise of exactly that many distinct returned points. Anchors, tie solves, infeasible thresholds, merging and dominance filtering can change archive size. For a maximized second objective, the controller may visit thresholds strongest-first; do not assume ascending exported epsilon order.

Without an explicit range, anchor solves estimate a range. Failure to obtain usable anchors can leave `epsilon_range_unavailable`. An explicit valid range permits threshold attempts even when anchors do not supply a usable range. The controller has bounded retries and may reuse later verified points to resolve an earlier threshold. A failed local solve does not establish infeasibility.

Mode-record controls

These are fields inside the multiobjective statement, not command-line flags.

Field	Default and allowed values	Meaning
normalization raw auto	raw; blend only	Raw/explicit scaling or estimated anchor normalization
points N	11; 2..1000; Pareto only	Requested epsilon-grid size
range LOW HIGH	Estimated when absent; Pareto only	Finite original-unit second-objective range; LOW <= HIGH
slice S	No explicit per-solve ceiling; positive finite seconds	Pareto scalar-solve ceiling within the shared total
dominance T	1e-7; positive finite; Pareto only	Filtering tolerance between returned objective vectors; not a constraint relaxation
polish F	0.25; finite 0..0.8	Fraction reserved for nonlinear scalar polishing
moves N	64; 0..10000	Bounded direct single-integer edits in polishing
pairs N	16; 0..1000	Bounded paired-integer edits in polishing
pool N	8; 0..32	Verified cross-pass starting-point pool size

Field	Default and allowed values	Meaning
native on off	on	Enable eligible native nonlinear bounding/search layer

Example:

```
multiobjective pareto points 7 range 0 10 slice 2 polish 0.25 moves 64 pairs 16 pool 8 native on
```

Keep the two named objectives and bounds from the preceding example. Every mode-record key must have a value; do not repeat a key. Blend does not accept the Pareto-only keys.

Shared budgets, refinement and qualification

All scalar passes share the overall --seconds, node and LP-pivot budgets. A slice does not add time. With finite time, the controller allocates remaining time across actual outstanding passes. A completed pass releases unused time. An explicit slice or an early-completed search can make the entire schedule finish before the overall allowance.

Nonlinear scalar polishing reserves the configured fraction of its allocation. Pareto additionally reserves 0.4 times that fraction of the finite total for final archive polishing: the default 0.25 therefore reserves 10% for that final stage. These are portions of the total, not extra seconds. Zero polishing disables the associated refinement/neighborhood/final-polish stages; zero pool disables cross-pass pool reuse. The linear proof path is not improved simply by increasing these nonlinear controls.

Continuous polishing fixes integer choices. Single and paired integer changes are rechecked and may need continuous restoration. Archive polishing protects achieved objective values while seeking a dominating improvement. An original-feasible seed may violate a new epsilon row: it remains only a repair seed until the current threshold is satisfied. Dominance tolerance never relaxes the original epsilon feasibility test.

Read the qualifications separately:

Output concept	Interpretation
sample_nondominated	Not dominated among the retained returned samples under the filtering rule
numerical_pareto_optimal	Supported scalar/tie solves established the required numerical optimality qualification
sampling_complete / schedule completion	Scheduled threshold work completed; not a complete Pareto front
all_thresholds_feasible	Verified feasible representatives exist for all scheduled thresholds
all_thresholds_optimal	All required threshold solves carry their numerical optimality qualification
front_complete	Remains false; finite sampling does not certify the entire front
feasible_from_archive	A later checked archive point satisfies that threshold; not proof its scalar optimum was solved

Unresolved thresholds and failed anchors remain meaningful diagnostics. Do not hide them just because the final archive is nonempty. A threshold proven infeasible is still not a feasible sampled threshold.

JSON and CSV exports

--result-jsonPATH applies to blend/Pareto CLI results; it is not a universal export switch for every solver mode. --pareto-csvPATH additionally requires Pareto mode. Single-objective, lexicographic and work/checkpoint CLI runs reject these exports. Typed core JSON is a separate interface that does support its own single-objective/hierarchy results.

Exports include objective metadata, feasible values, scalar-pass statuses and samples. Pareto CSV has objective columns and var_ coordinate columns. Preserve the original model and objective order alongside the exports. Missing bounds are unavailable, not zero. A scalar pass bound belongs to its specified objective and feasible region; never apply it to a different trade-off.

When reading extended JSON numeric values, preserve decimal text:

```
import json
from decimal import Decimal
from pathlib import Path
result = json.loads(Path('pareto.json').read_text(), parse_float=Decimal)
```

CLI and typed-core schemas are not interchangeable. Check each schema/version before consuming it. Use the complete Pareto archive rather than parsing only Candidate1 as if it summarized every objective.

33. Priority and lexicographic linear objectives

Express priorities instead of a weighted compromise

Use distinct priorities when a higher-priority result must be protected before optimizing the next goal. All constraints and every objective must be affine. Nonlinear lexicographic optimization is not supported by this path.

Save as priority.txt:

```
int basic,premium
bound basic=[0,100]
bound premium=[0,100]
demand: basic+premium>=20
objective cost priority 20 minimize 5*basic+5*premium
objective preference priority 10 maximize premium
```

```
optimizer.exe --file priority.txt --lp-engine native --seconds 20
```

The primary minimum cost is 100. With that cost protected, premium is maximized at 20 and basic is 0. It is not a weighted score; a large benefit cannot arbitrarily compensate for a worse protected cost.

Syntax:

```
objective NAME priority N [abstol A] [reltol R] minimize EXPRESSION
objective NAME priority N [abstol A] [reltol R] maximize EXPRESSION
```

Up to 64 unique objective names and distinct priorities are allowed. Priority is a nonnegative signed-32-bit integer; higher values run first. Equal-priority blending is not added by this syntax. Do not mix it with a traditional scalar objective or a multiobjectiveblend/pareto record. Imported objective hierarchies and specialized service hierarchies are not provided.

Degradation tolerances and locks

For a completed pass value v :

```
d = max(abstol, reltol * abs(v))
minimize: later f(x) <= v + d
maximize: later f(x) >= v - d
```

Both degradation tolerances default to zero and must be finite/nonnegative. The absolute tolerance has objective units; relative tolerance is dimensionless. It is the maximum, not the sum. Objective constants participate in v and its relative tolerance. Ordinary constraint tolerance also applies to the generated lock rows.

For example, allowing `abstol5` on `cost=100` permits later cost up to 105. That is an intentional hierarchical compromise, not strict cost optimality at the final point. A tolerance on the last objective has no later pass to affect. Distinguish primary pass value from its final returned value after allowed degradation.

Limits and hierarchy completion

Time, pivots and integer nodes are shared across all passes. A lower-priority pass starts only after its predecessor reports numerical optimal. A feasible time-limited or gap_limit predecessor does not authorize advancing, even if its incumbent is useful. An interruption retains the last verified feasible point. Do not sum pass gaps into one hierarchy gap.

Read Hierarchycomplete and per-objective status/value/bound. An unstarted objective may have a final value evaluated at the returned point without having been optimized. Candidate1:objective=... reports the highest-priority objective at the final point, not a combined hierarchy score. Exit 0 means a verified feasible candidate exists; it does not by itself establish a completed hierarchy.

Supported entry points include native CLI, legacy DLL arguments and typed-core native text/file solves. Standard and supported extended precision are available. Do not combine with work/checkpoint mode, multi-seed scalar selection, --sqp, --crew, nonlinear global controls, --iis or --feas-relax. Diagnose original feasibility separately. Typed result JSON exposes objective pass/final values and hierarchy completion; the blend/Pareto CLI export flags do not apply.

34. Deterministic bounded work, checkpoints and continuation

Scope and present recommendation

This is an opt-in search scheduler for ONE traditional scalar NLP/MINLP objective, available in the current CLI and legacy optimizer_run argument interface. It saves search progress between invocations. It is not ordinary timed search with a save flag: it uses a different bounded schedule and may produce worse objective quality.

Known quality limitation: the user reported Specialized heat exchange at 166509.57334840705 after 32+128 work units, versus 154997.33486584452 from a 30-second ordinary timed command. The latter agrees with the listed reference to numerical precision. The commands used work seed 38 and the timed command's default seed, so this is not a controlled same-seed comparison. It nevertheless demonstrates that more work units do not guarantee reference-level quality. Earlier seeded preservation tests did not establish cold-start discovery quality. Prefer the successful timed mode for that case while the work scheduler is investigated. This documentation update does not claim that regression has been repaired.

Every work-mode option

Option	Default / range	Meaning
--work-limit N	Required positive count; 1..1000000	ADDITIONAL completed units in this invocation, not a cumulative target
--work-iterations N	24; 1..10000	Local/restoration iteration cap used by work mode
--work-pivots N	2000; 1..10000000	Per-internal-LP pivot cap; not a global accumulated pivot target
--work-qp-passes N	32; 1..100000	Active-set QP pass cap
--checkpoint PATH	No output unless checkpoint/resume supplied	Publish resulting state to this file
--resume PATH	None	Load compatible state; defaults to updating the same file
--incumbent PATH	None; fresh run only	Full name/value coordinate file, verified against the original model
--seconds S	Ordinary default 20; 0 disables deadline	Optional cooperative wall-time safety ceiling

Supplying checkpoint, resume, incumbent or a phase-cap flag enables work mode and therefore requires a valid --work-limit. --checkpoint alone is not a request to save the ordinary timed solver. Without a checkpoint or resume path, a work-mode solve can run but does not publish reusable state.

One unit is a completed bounded attempt. It is neither one iteration nor one CPU-second. Model size, restoration difficulty and arithmetic mode affect its cost. The cumulative cursor is limited to 1000000000 units. The unit schedule mixes continuous relaxation/rounding, simple integer assignments, same-assignment refinement, coordinated one/pair/group changes and restarts. A bounded pool and assignment-visit information guide later attempts.

Increasing ordinary --starts or --iterations is not a substitute for the dedicated work controls. Work mode overrides the local iteration setting with --work-iterations and uses its own schedule. Legacy integer-enumeration/global-box knobs do not turn it into the old timed algorithm. Keep irrelevant controls out of reproducibility experiments.

A small complete example

Save as work_demo.txt:

```
minimize (x-3)^2+20*z
int z
bound z=[0,1]
bound x=[0,10]
x>=2*z
```

The known best decision is $z=0$, $x=3$, objective 0. A heuristic solve is still accepted through the original checks, not by assuming those coordinates in advance.

```
optimizer.exe --file work_demo.txt --work-limit 8 --seconds 0 --seed 38 --tol 1e-9 --checkpoint
work_demo.owc
```

Later:

```
optimizer.exe --file work_demo.txt --work-limit 16 --seconds 0 --seed 38 --tol 1e-9 --resume
work_demo.owc
```

The cumulative completed count becomes 24. --seconds0 removes the clock ceiling; it does not increase the requested unit count. If eight units finish quickly, the first invocation can return quickly. More search is an explicit additional-work request.

General heat-exchanger example

These commands assume the downloaded file exists in minlplib_data; use your actual path. The model is not generated by this manual.

```
optimizer.exe --nl-file "minlplib_data\heatexch_gen1.nl" --work-limit 16 --seconds 0 --seed 38
--tol 1e-9 --checkpoint general_38.owc
optimizer.exe --nl-file "minlplib_data\heatexch_gen1.nl" --work-limit 16 --seconds 0 --seed 38
--tol 1e-9 --resume general_38.owc
```

Run each command separately. On the tested Linux build, seed 38 returned 221479.5961427013 at 16 units and 221478.60175301062 at 32 total. The full 32-unit state matched a 16+16 continuation byte for byte. These are measured observations, not Windows objective guarantees. Seed 41 had no feasible point at 32 units but reached 209799.59104137763 after another 32. This illustrates useful continuation of an unsuccessful search, not a guaranteed improvement on every model.

Add time and fork a saved search

For a cooperative 15-second ceiling while requesting up to 32 additional units:

```
optimizer.exe --file work_demo.txt --work-limit 32 --seconds 15 --seed 38 --tol 1e-9 --resume
work_demo.owc
```

The invocation stops when the requested completed work or the cooperative time ceiling is reached. A partial unit does not advance the committed cursor/RNG/guidance; it is replayed later. Any better candidate already verified during that partial attempt is retained independently. A partial point may make the retained best differ favorably from an uninterrupted trajectory, so full byte identity is not promised for arbitrary wall-time interruptions.

To preserve the input checkpoint and write a branch:

```
optimizer.exe --file work_demo.txt --work-limit 8 --seconds 0 --seed 38 --tol 1e-9 --resume
work_demo.owc --checkpoint work_demo_branch.owc
```

Forking does not allow a different seed or changed algorithm settings. Start a fresh checkpoint for a different search configuration.

What is saved and checked

The snapshot stores the cumulative cursor, random-generator state, committed warm coordinates, best verified coordinates, promising assignment pool and visit counts. The best objective is recomputed from coordinates; a claimed scalar objective is not trusted. Original equations/inequalities, finite search domain, declared hard bounds, exact integrality and requested tolerance are checked again on load. Derived sampling bounds do not replace original inequality tolerances.

Compatibility includes the exact parsed model representation, seed, precision, feasibility/stationarity/stagnation settings, phase caps, configured ranges, relevant algorithm settings and compiler/platform/compilation timestamp. Keep the SAME executable for a checkpoint's lifetime. Rebuilding can invalidate it even when source looks unchanged. Moving an unchanged file need not change its model content, but do not edit its algebra/name mapping or sidecars. Cross-platform floating-point equivalence is not guaranteed.

Additional work count, wall-time ceiling and output paths can change on resume. Algorithm caps, tolerance and precision cannot. For extended mode, add `--extend` and the same explicit tolerance to both commands; use a different checkpoint from the normal run.

Import a previous feasible incumbent

Save every variable as a name/value pair. For the small example:

```
x 3
z 0
```

Save as `work_seed.txt` and run:

```
optimizer.exe --file work_demo.txt --work-limit 8 --seconds 0 --seed 38 --tol 1e-9 --incumbent
work_seed.txt --checkpoint seeded_demo.owc
```

Every name must be known and supplied exactly once; coordinates must be finite and satisfy the original model, hard bounds and integer domains. Objective-only files are insufficient. Use variable names printed by the model/import path, not guessed business labels. The current source package provides `tests/extract_incumbent.py` to extract a full printed coordinate vector:

```
python tests\extract_incumbent.py previous.stdout.txt previous.incumbent.txt
```

This helper does not certify the point. The fresh work run verifies it. Do not repeat `--incumbent` on resume: the snapshot already contains its best point. Imported/typed partial starts are different interfaces; the checkpoint seed file requires the complete coordinate vector.

Stopping, exit codes and file safety

Inspect Completed work units, Work units this invocation, Uncommitted unit will be replayed, Stopped and Checkpoints saved. Exit 0 returns a verified feasible candidate; 2 means no feasible candidate at the requested tolerance and is not an infeasibility proof; 1 is an error. An exit-2 run may still have saved useful search state. Do not chain continuation behind shell `&&` if that prevents resuming exit 2. A failed save is an error, not a confirmed checkpoint.

One coherent snapshot is published at graceful invocation end, including a cooperative stop. Unit commits are otherwise in memory. Choose smaller calls for more frequent recovery points. A force kill, crash or power failure can lose unpublished work. Publication uses a flushed temporary file, atomic replacement and read-back verification; POSIX power-loss durability is not promised. A checksum detects accidental corruption, not malicious modification.

Canonical `.running` directory leases guard input/output checkpoints during the call; `.writing` guards the temporary publication. Concurrent writers are rejected. A crash may leave a stale directory: remove it only after ensuring no process uses that checkpoint. Keep a separate backup before maintenance. Model input and incumbent files must not also be checkpoint

output paths. A path spelled - is a literal checkpoint filename, not stdout; only supported model-input --file- is stdin.

Incompatible modes and quality troubleshooting

Work mode rejects root-only models, named multiple objectives (even one named objective), explicit linear/crew/global modes, multi-seed orchestration, dedicated operational services, --sol-file, CLI JSON/CSV result exports and multiple retained candidates. Typed-core options do not expose work continuation. An affine scalar expression can be searched heuristically through work mode; that does not confer the LP/MILP proof path.

Symptom	Interpretation and next step
Immediate return, worklimitreached	Requested units completed quickly; increase additional work only if objective quality benefits
Many continued units, poor/stagnant objective	Search may be trapped or phase caps inappropriate; compare with same-seed timed cold starts; more units are not a universal cure
Zero units completed under short extended runs	A unit can exceed the ceiling and be replayed; try no time ceiling or a larger one, or fresh state with smaller phase caps
Settings mismatch	Use the original binary/model/settings; otherwise extract and reverify coordinates into a fresh run
No candidate but valid checkpoint	Continue if useful; lack of a point is not proof of infeasibility
Better candidate exists in another run	Import its complete verified coordinates into a fresh compatible work run; objective scalar alone cannot be used

For the Specialized regression, use ordinary timed search until a tested correction improves cold-start work quality. This guide documents the current implementation and limitations; it does not silently change algorithms.

35. Backend selection, diagnostics and additional numerical controls

Select the affine LP/MILP engine

```
optimizer.exe --file production.txt --lp-engine native --seconds 20
optimizer.exe --file production.mps --lp-engine auto --seconds 20
optimizer.exe --file production.mps --lp-engine highs --seconds 20
```

native explicitly selects our affine engine. highs requires a build linked to your existing prebuilt HiGHS library; it fails if unavailable. auto uses enabled backend selection and can fall back to native for supported preflight/range cases; it is not a promise to retry every external error. OPTIMIZER_LP_ENGINE can override auto with native or highs. Prefer an explicit CLI engine for controlled comparisons, and inspect the printed backend. Do not confuse --lp-enginenative with the separate bounded-native nonlinear layer.

Native-specific cuts, reliability branching, basis caches and incumbent neighborhoods do not become identical HiGHS settings simply because their CLI values parse. Force native to evaluate those implementation-specific controls. HiGHS is an external optional dependency, not source built by this manual. Some older built-in help still prints historical single-file build examples; Chapter 1's modular commands supersede them.

HiGHS discovery and extended affine refinement

In an enabled build, affine --extend with engine auto/highs discovers a double point, then checks/refines using the ORIGINAL binary128 model and requested tolerance. Native engine selection bypasses discovery. Standard double HiGHS requests require constraint tolerance at least 1e-10; use native or supported binary128 refinement for tighter requests. Auto can select the native path for a too-tight double tolerance.

```
optimizer.exe --file production.mps --lp-engine highs --extend --tol 1e-20 --seconds 120
optimizer.exe --file production.mps --lp-engine native --extend --tol 1e-20 --seconds 120
```

The discovered point is a seed, not a high-precision certificate. LP refinement solves the full native model; basis transfer is not provided. MILP refinement may fix discovered integers and solve their continuous subproblem. A resulting feasible_refined point has no global mixed-integer bound from that fixed assignment. Only appropriate full-model work can establish a new numerical global conclusion. HiGHS discovery bounds are not reused as binary128 certificates, and the final requested tolerance is not silently loosened.

Discovery/refinement share the total budget and cancellation. Discovery may receive up to 60% of remaining finite scalar time; unlimited overall runs still cap discovery itself at 30 seconds. Preserve extended decimal values through the legacy output or decimal-aware JSON reader. Typed-core ABI v1 remains double-based and cannot carry a binary128 witness merely by returning many printed digits.

Native MILP memory, branching, cuts and neighborhoods

The following supplement Chapter 31. MiB settings are estimates for specified frontier/cache structures, not process RSS caps or evidence of leak-free operation.

Option	Default / range	Purpose
--mip-memory-mb N	256; 1..1048576 MiB	Estimated frontier/cache budget; limited results retain valid known information

Option	Default / range	Purpose
--mip-cache-mb N	64; 0..1048576 MiB	Disposable compatible LP basis cache; zero disables
--mip-strong-candidates N	4; 0..32	Reliability probe candidate count at eligible early nodes; zero disables probing
--mip-strong-pivots N	64; nonnegative parser integer	Per-probe LP pivot ceiling; zero permits no probe pivots
--mip-reliability N	3; 0..100	Pseudocost observations before estimates are trusted
--mip-pump-passes N	24; 0..1000	Auxiliary primal LP portfolio pass cap; zero disables
--mip-lns-nodes N	128; 0..10000	Per-neighborhood node cap; zero disables neighborhood work
--mip-lns-seconds S	5; positive finite seconds	Per-neighborhood ceiling within the existing total
--no-mip-cover-cuts	Off: cuts enabled	Disable eligible integer-row rounding/binary cover cuts
--no-mip-conflicts	Off: conflicts enabled	Disable exact-activity binary decision nogoods

--no-mip-heuristics disables the automatic primal repair/diving/neighborhood aids. A larger neighborhood budget can consume time that otherwise reaches branch-and-bound; compare valid incumbents AND valid bounds. Auxiliary/fixed-neighborhood LP bounds are not global bounds. Disposable-cache eviction should not be interpreted as infeasibility.

Existing --mip-clique-cuts, --mip-clique-rounds and --mip-clique-limit remain separate, opt-in clique controls. Cover/conflict disabling is not the same as disabling every possible valid cut. --lp-pivots, --mip-nodes, global time and requested gaps remain important aggregate limits.

```
optimizer.exe --file production.mps --lp-engine native --mip-memory-mb 512 --mip-cache-mb 128
--mip-strong-candidates 4 --mip-strong-pivots 64 --mip-reliability 3 --seconds 30
```

Sparse nonlinear controls

--nlp-sparse forces the sparse active-set/L-BFGS path; it is selected automatically for eligible dimensions above 128. --nlp-memoryN sets L-BFGS history pairs, default 12, range 1..100. These are nonlinear search controls, not native affine LP settings.

```
optimizer.exe --sqp --file nonlinear.txt --nlp-sparse --nlp-memory 12 --seconds 30 --seed 38
```

Sparse algebra can reduce storage on suitable large models; it does not make a dense formulation sparse, fix poor scaling, guarantee a better objective or establish a global optimum. Increase history only after measuring memory and convergence. In work mode these relevant settings must remain compatible on resume.

Explain linear infeasibility with IIS diagnostics

Name rows so explanations refer to meaningful requirements. Save as infeasible.txt:

```
minimize x
bound x=[0,10]
required: x>=8
capacity: x<=3
```

```
optimizer.exe --linear --file infeasible.txt --lp-engine native --iis --seconds 10
```

--iis applies a numerical deletion filter to conflicts, including explicit bounds with declared variable domains as background. Only completed supported checks justify an irreducibility label. It is not a minimum-cardinality conflict, an exact-arithmetic certificate, or a diagnosis of unmodelled business rules. Under a limit, an incomplete diagnostic is not an irreducible result. It is a single-objective affine diagnostic; do not combine it with hierarchy, blend/Pareto or work mode.

Explore a linear feasibility relaxation

```
optimizer.exe --linear --file infeasible.txt --lp-engine native --feas-relax --seconds 10
```

--feas-relax minimizes total L1 row violation while retaining original variable bounds/domains. It helps identify which requirements need reconsideration. The returned relaxed point is a diagnostic, not a feasible solution to the original model. Violations in different units are not automatically economically comparable; reformulate/scalewise interpret the diagnostic. Diagnostics and the original solve share time. Never feed an infeasible relaxed point into an operational plan as if all original constraints were satisfied.

Bounded native nonlinear search and qualification

Eligible explicitly bounded, standard-double AUTO NLP/MINLP models can use expression propagation, binary probing, reduced local search, guarded LP relaxations and spatial subdivision before fallback. Current eligibility includes at most 64 original variables and 1500 expression nodes. Supported expression/domain combinations include arithmetic, eligible powers, exp/log, sin/cos and selected piecewise functions. Division must exclude zero; finite valid domains matter. Unsupported graphs, automatic bounds, explicit --sqp and binary128 do not acquire a new native nonlinear certificate.

This layer is separate from Chapter 8's explicit numerical-global mode and Chapter 34's work scheduler. Its numerical optimal status needs a guarded floating-point objective bound, not only successful polishing. Current gap allowance includes the configured absolute/relative target and a constraint-tolerance-sized numerical safeguard. It is not exact-arithmetic certification. The layer has conservative internal work limits; its limited search does not enumerate all arbitrary MINLP possibilities. Inspect native-search diagnostics and original feasibility.

36. Typed C core API, progress and cancellation

Two interfaces with different contracts

Interface	Use it for	Important limitations
Legacy optimizer_dll.h / optimizer_run	CLI-compatible modes, specialized services, extended printed values and work/checkpoint arguments	Synchronous captured output; current general calls serialize; inspect API status and solver exit separately
Typed optimizer_core.h ABI v1	Independent mathematical contexts, typed double values, structured JSON, progress and cooperative cancellation	No general operational record interface, no typed binary128 and no typed checkpoint options
Direct placement API	Array-based resource placement	Separate model/options/result contract; follow placement header

Use matching headers and binary exports. An old DLL with the same filename may lack the additive core functions. Match architecture and runtime dependencies. None of these interfaces turns a model text parser into arbitrary Python/C++ code evaluation.

Context ownership and safe concurrency

Create one context per independent concurrent solve. Only one solve may run on a context at a time. API return values are OPT_CORE_OK=0, OPT_CORE_ERROR=1, OPT_CORE_BUSY=2; these are API statuses, not the legacy CLI solver exit table. After a completed call, inspect feasibility, solver status and available bound in result JSON.

Initialize public option/info structures using the provided default initializer where available, struct_size=sizeof(struct) and abi_version=1. Do not copy hand-guessed layouts across languages. Structures and ABI versions are checked; additive accessors do not enlarge the existing options layout.

Solve calls are synchronous. A progress callback executes on the solving thread. Keep it short. Only cancellation is reentrant on that same context: do not solve, query, reconfigure or destroy it from inside its callback. Query results while idle. The caller must prevent destroy/read/change races; a busy status is not a substitute for lifetime discipline. Borrowed strings remain owned by the context and can be invalidated by its next solve or destruction. Copy them if needed longer.

Minimal C example

```
#include <stdio.h>
#include "optimizer_core.h"

int main(void) {
    optimizer_core_context *ctx = optimizer_core_create();
    if (!ctx) return 1;
    optimizer_core_options_v1 options;
    optimizer_core_default_options_v1(&options);
    options.seconds = 5;
    options.backend = OPT_CORE_AUTO;
    if (optimizer_core_set_options_v1(ctx, &options) != OPT_CORE_OK) {
        optimizer_core_destroy(ctx);
        return 1;
    }
}
```

```

}
int rc = optimizer_core_solve_text_v1(ctx,
    "minimize (x-3)^2;bound x=[0,10]");
if (rc == OPT_CORE_OK)
    puts(optimizer_core_result_json_v1(ctx));
else
    fprintf(stderr, "API failed: %d\n", rc);
return optimizer_core_destroy(ctx) == OPT_CORE_OK ? 0 : 1;
}

```

Compile/link your caller against the matching optimizer library and its headers; the optimizer's build does not compile external solver source. Do not interpret a successful API call alone as proof of feasibility/optimality.

Typed option fields and defaults

Defaults below come from the current core initializer, not the similarly named root CLI defaults.

Field	Default / range	Meaning
seconds	20; finite >=0	Cooperative time; zero disables deadline
constraint_tolerance	1e-9; positive finite	Original-model acceptance
gradient_tolerance	1e-6; positive finite	Local nonlinear stationarity
relative_gap / absolute_gap	0 / 0; nonnegative finite	Applicable objective-bound stopping targets
seed	42; uint64	Search seed
node_limit / pivot_limit	100000 / 1000000; 0..2147483647	Aggregate work ceilings in supported paths
frontier_memory_mb / basis_cache_mb	256 / 64	Estimated native MILP storage budgets; minimum frontier 1, cache 0 allowed; max 1048576
backend	AUTO=0, LINEAR=1, NLP=2	AUTO dispatch; force affine; force local nonlinear
starts / iterations	128 / 300; 1..1000000	Local search restart/iteration allowances
lbfgs_memory	12; 1..100	Nonlinear history pairs
sparse_nlp	0	Nonzero requests sparse nonlinear path
strong_candidates / strong_pivots	4 / 64	Native reliability probe limits; candidates 0..32, pivots <=2147483647

Select the affine engine separately with `optimizer_core_set_linear_engine_v1`: AUTO=0, NATIVE=1, HIGHS=2. Do not confuse engine selection with backend selection. The Python adapter exposes its additive engine helper when supplied; consult its matching version. Configure extra MILP portfolio settings with `optimizer_core_default_milp_options_v1` and `optimizer_core_set_milp_options_v1`: enabled=1, neighborhood_nodes=128 (0..10000), portfolio_passes=24 (0..1000), neighborhood_seconds=5 (positive finite). Those fields are not appended to the original core options structure.

Python core example

Place the matching `optimizer_core.py` next to the application or on its import path:

```

from pathlib import Path
from optimizer_core import Core

library = Path('optimizer.dll').resolve() # Linux: liboptimizer.so
with Core(str(library)) as solver:
    solver.configure(seconds=5, backend=0,
        constraint_tolerance=1e-9, seed=38)
    result = solver.solve('minimize (x-3)^2;bound x=[0,10]')
    print(result['status'], result['feasible'])
    if result['feasible']:
        print(result['objective'], result['variables'])

```

On Windows, use appropriate DLL search directories before loading, for example `os.add_dll_directory` retained in a live scope for the DLL and MinGW runtimes. The legacy `optimizer_ctypes.py` adapter and typed `optimizer_core.py` adapter are different modules and return different result contracts.

Progress events and cooperative cancellation

Event	Code	Meaning
STARTED	1	Solve entered
INCUMBENT	2	A verified feasible incumbent event
BOUND	3	Bound/search-work event; check has_bound before using its value
FINISHED	4	Solve finished
OBJECTIVE_STARTED	5	A hierarchy objective pass starts
OBJECTIVE_FINISHED	6	That hierarchy pass ends

Event objective/bound values refer to the active pass. ABI-v1 progress does not include an objective-pass index; a caller can track pass-start events in priority order. Never use an absent bound as zero.

```
from optimizer_core import Core

with Core('C:/optimizer/optimizer.dll') as solver:
    solver.configure(seconds=10)
    def notify(event):
        if event['objective'] is not None:
            print('incumbent', event['objective'])
        if event['seconds'] >= 2:
            solver.cancel()
    solver.progress(notify)
    result = solver.solve('minimize (x-3)^2; bound x=[0,10]')
    print(result['status'], result['feasible'])
```

This is cancellation at a callback opportunity, not a guarantee that a callback arrives at exactly two seconds. For external application deadlines, a separate controlling thread can request cancel() while the solve is active, following the context lifetime rules. Do not forcibly terminate an arbitrary in-process C++ solving thread. A supervised subprocess provides stronger fault isolation when the embedding architecture requires it; retain a separately verified fallback decision. No zero-latency or memory-leak guarantee follows from bounded search controls.

The Python wrapper retains callback references. A callback exception requests cancellation and is surfaced by the wrapper after return; keep application logging and exception handling explicit.

Named starts, queries and result fields

optimizer_core_set_primal_start_v1 supplies named finite double coordinates as guidance. They are reverified, not accepted as proof or a basis. The Python helper is:

```
solver.primal_start({'x': 3.0})
```

Use it before solve, while idle. Partial named starts and backend-specific initialization differ from the strict complete-coordinate work-mode --incumbent file. A start does not continue the random/search cursor of a checkpoint.

Typed JSON includes status, error, feasible, objective, bound, seconds, maximum_violation, variables, nodes and pivots. Unavailable numerical fields are null. first_feasible_seconds and best_objective_seconds are available only where populated; they are not total application latency. first_order_convergence and kkt_measure are local nonlinear checks. linear_engine, milp_native, milp_diagnostics, storage estimates and native/polishing audits explain the used path.

For named objectives, objective_mode, objectives, hierarchy_complete and final/pass values qualify the result. For Pareto, consume the complete pareto_points archive and scalar_solves, with their separate statuses. front_complete remains false. Do not collapse a multiobjective result into a single scalar screen field.

Additive accessors

API group	Accessors / Python helpers	Purpose
Basic result	status/error/variable_count/variable_name/primal_value/objective/bound/result_json	Read supported scalar values and full structured result
Objective passes	objective_count/objective_name/objective_status/get_objective; objective_info(i)	Priority pass values, final values, tolerances and statuses
Multiobjective metadata	objective_mode/objective_weight_scale	Declaration-order blend/Pareto metadata
Pareto archive	pareto_count/get_pareto/pareto_primal/pareto_status/pareto_csv; pareto_info(i), pareto_csv()	Sample metadata and full coordinates
Coverage summary	get_pareto_summary; pareto_summary()	Threshold/schedule coverage separate from front completeness
Polishing audit	get_polish; polish_info()	Refinement work, not proof
MILP portfolio	set_milp_options/get_milp; matching adapter methods	Configure/audit native incumbent search
Native nonlinear	get_native; matching adapter method	Supported native-search work counters

Names in the table abbreviate the common `optimizer_core_..._v1` prefixes/suffixes; use the exact declarations in `optimizer_core.h`. Initialize each queried info structure with its correct size/version. Counts and strings are context-owned views, not arrays for the caller to free. Different objective modes populate different pass/metadata arrays; use the matching mode and result schema rather than assuming every accessor returns meaningful data everywhere.

37. New service controls, compatibility and reproducible evaluation

Automatic TSP versus manual tuning

Current default TSP search uses its automatic policy. Supplying a `--tsp-*` tuning flag, except `--tsp-target`, selects the legacy manual search. The default command should therefore be your first baseline:

```
optimizer.exe --tsp --file att48.tsp --seconds 10 --seed 38
```

`--tsp-pool50`, `--tsp-elite`, `depth`, `neighbor` and `iteration` settings remain supported advanced comparisons, not automatically recommended upgrades to the default algorithm. Read `--tsp-advanced-help`. `--tsp-target` requests stopping at an attained cost and does not certify that cost as optimal. Historical manual defaults in the option tables are conditional on that manual path; do not treat them as the current automatic policy's complete parameterization. The shorthand token `--tsp-` is not an option: use an actual listed flag.

Exact-mode option for small 1D bin packing

```
optimizer.exe --bin-packing --file models/bin_packing.txt --bin-exact --service-iterations 10000 --seconds 20
```

`--bin-exact` opts into supported Ryan-Foster branch-and-price for at most 256 copies and normalized capacity at most 2000000. Exact conflict-aware pricing is required for valid node bounds. Work/time limits retain incumbents and do not imply completed proof. Larger models retain the existing aggregated heuristic path. Read the actual status and bound; the word `exact` in a flag is not by itself evidence that the search finished. This option is restricted to `BIN_PACKING`; it is not a rectangular-packing or general cutting-stock switch. Its work budget uses `--service-iterations`, not the `MINLP --work-limit`.

Capability and incompatibility matrix

Feature	Scalar NLP/MINLP timed	Work/checkpoint mode	Blend/Pareto	Affine priority	Dedicated services
Cooperative seconds	Yes	Yes; 0 recommended for completed-unit repeatability	Shared total	Shared total	Service-specific
Explicit work-unit cursor and disk resume	No	Yes	No	No	No
CLI JSON/CSV flags in Chapter 32	No	Rejected	JSON; CSV Pareto only	Rejected	Not this export interface
Named objectives	Not a traditional scalar model	Rejected	Required	Required	Separate grammar
Native multi-seed wrapper	Supported heuristic paths	Rejected	Rejected	Rejected	Supported heuristic services only
Typed core double contexts	Supported mathematical models	No typed work options	Supported	Supported affine	No general service records

Feature	Scalar NLP/MINLP timed	Work/checkpoint mode	Blend/Pareto	Affine priority	Dedicated services
Binary128	Supported builds/paths	Direct selected arithmetic	Supported with qualification limits	Supported affine	Mode-specific; many operational modes reject

The table describes these interfaces, not a universal restriction on writing your own application-level JSON or running independent external jobs. A program can serialize its own decision record from a supported result; that is distinct from a built-in CLI export flag.

Choosing effort versus accuracy

Feasibility tolerance governs model acceptance; stationarity concerns local continuous optimality; objective quality concerns the achieved decision value; global gap needs a valid bound. More printed digits or a tighter residual cannot substitute for better exploration. Use explicit physical bounds and appropriate scaling before tightening tolerances.

Evaluate cold-start and seeded runs separately. An imported good solution demonstrates preservation/refinement, not ability to discover it from scratch. Compare identical models, seeds, precision, binaries and budgets. Timed experiments can still diverge under machine load. Compare deterministic work by completed units, not by assuming equal seconds per unit. When comparing timed and work schedulers, report BOTH runtime and objective because their search paths differ.

Benchmark evidence and the Specialized limitation

The supplied normal-precision MINLPlib PHP report dated 16 October 2026, 14:11:02 UTC records seven process-profile instances, budgets 5/15/30 seconds and seeds 38..42: 105 runs. It accepts solver-reported feasibility, not independent original-model certification. It reports 100 accepted runs: 30/35 at five seconds, 35/35 at 15 and 30 seconds. All five misses are Specialized heat exchange at five seconds.

Instance	Best reported objective	Listed primal reference	Best budget / seed
alkyl	-1.7649996476604	-1.76499965	5s / 38
process	-1161.336602411	-1161.336602	5s / 40
haverly	-400.00000001404	-400	30s / 40
pooling_haverly1pq	-400.00000000793	-400	5s / 38
pooling_haverly2pq	-600.00000000072	-600	5s / 38
heatexch_gen1	229590.92988541	154895.933	30s / 38
heatexch_spec1	154997.33486584	154997.3349	30s / 38

General's best is 48.22% above that reference; its 30-second median is 391713.90639932. Six other best values agree closely with their references at displayed precision. These are primal-reference differences, not certified global gaps. None of those facts proves quality on arbitrary MINLP families, and the profile is not seven exclusively mixed-integer models. The report does not supply a hardware/executable fingerprint or direct competitor comparison.

The separate user-reported 32+128-unit Specialized test in Chapter 34 is weaker than the successful timed result. Keep that regression visible when choosing a mode. Do not describe the resumable scheduler as an across-the-board quality replacement for the existing timed solver. Future source corrections require cold-start, same-seed testing on both heat-exchanger models and reserved unrelated families; this guide makes no claim that those corrections are already implemented.

Practical OEM acceptance checklist

Record the model/hash, binary/build, active backend, seed, precision, bounds, options and all outputs. Measure time to first verified point, objective over time/work, final original feasibility, useful bounds, total wall time, cancellation latency and peak memory. Include unsuccessful runs. A median objective among accepted runs must be accompanied by acceptance rate. Run long-lived/repeated and concurrent tests before making reliability or memory claims.

A useful feasible decision can be the right application output without an optimality proof. That is a product/use-case judgment. Required physical, contractual or operational constraints must still be represented and checked; approximation of the objective is not permission to violate a hard rule. A numerical solution can be sensitive to conditioning and uncertain inputs even with a small residual.

38. Updated option index, migration and troubleshooting

Added mathematical CLI options at a glance

Option / family	Detailed location	Scope
--result-json PATH;	Chapter 32	Blend/Pareto structured CLI export; CSV Pareto only
--pareto-csv PATH		
--multiobjective-help;	Chapters 32 and 38	Grammar help / regression diagnostics
--multiobjective-self-test		
--lex-help; --lex-self-test	Chapter 33	Affine objective hierarchy help / tests
--work-limit;	Chapter 34	Bounded scalar NLP/MINLP work scheduler
--work-iterations;		
--work-pivots;		
--work-qp-passes		
--checkpoint; --resume;	Chapter 34	Current strict continuation/full-coordinate seeding interface
--incumbent		
--lp-engine	Chapter 35	Affine engine selection; optional prebuilt external dependency
auto native highs		
--iis; --feas-relax	Chapter 35	Single-objective affine diagnostics
--mip-memory-mb;	Chapter 35	Estimated native frontier/cache storage controls
--mip-cache-mb		
--mip-strong-candidates;	Chapter 35	Native reliability branching/probe effort
--mip-strong-pivots;		
--mip-reliability		
--mip-pump-passes;	Chapter 35	Native primal portfolio/neighborhood effort
--mip-lns-nodes;		
--mip-lns-seconds		
--no-mip-cover-cuts;	Chapter 35	Disable eligible native cut/conflict aids
--no-mip-conflicts		
--nlp-sparse; --nlp-memory	Chapter 35	Sparse nonlinear active-set/L-BFGS controls
--milp-self-test	Chapters 35 and 38	Native MILP safety tests
--bin-exact	Chapter 37	Supported small BIN_PACKING exact-mode attempt
--tsp-advanced-help	Chapter 37	Manual TSP policy help; tuning flags select legacy path

Model fields such as polish, priority, normalization, native, points, slice, range and pool are not CLI flags. A bare --polish is not added by this documentation. Root-only options, native linear options and service options remain mode-specific.

Migration from the 8 October edition

1. Build the modular source with build.py or its current Windows wrappers. Do not compile only the thin optimizer.cpp entry point. Reuse optional prebuilt HiGHS rather than building it repeatedly.
2. Select the result contract: legacy CLI-compatible output versus typed core JSON/values. Keep API status separate from solver status and original feasibility.
3. Use native named objective records for blend, Pareto or affine priority workflows. Do not combine their controls with work/checkpoint mode.
4. Use the established timed path as your quality baseline. Enable work mode only deliberately, with the Specialized limitation understood.
5. Give --work-limit whenever a checkpoint/resume/phase-cap work flag is used. Keep the same executable and settings to resume. Use a fresh state for a changed precision or phase configuration.
6. Preserve full coordinates, not just an objective scalar, when moving a solution into a fresh work run. Reverify the original model before relying on it.
7. Preserve raw outputs and measure application-level latency. Search-time observations

are not operating-system deadlines.

Diagnostic commands and build validation

```
optimizer.exe --help
optimizer.exe --solver-help
optimizer.exe --lp-help
optimizer.exe --lex-help
optimizer.exe --multiobjective-help
optimizer.exe --global-help
optimizer.exe --tsp-help
optimizer.exe --tsp-advanced-help
optimizer.exe --cvrp-help
optimizer.exe --services-help
optimizer.exe --network-help
optimizer.exe --import-help
optimizer.exe --placement-help
optimizer.exe --multi-seed-help
```

--core-help and --native-nonlinear-help are not current CLI commands; use the public header/documented APIs and corresponding chapters. The typed core is not configured by forwarding arbitrary CLI flags.

After building, relevant regression commands include:

```
optimizer.exe --self-test
optimizer.exe --solver-self-test
optimizer.exe --lp-self-test
optimizer.exe --milp-self-test
optimizer.exe --lex-self-test
optimizer.exe --multiobjective-self-test
optimizer.exe --self-test --extend
python tests\dll_smoke_test.py --library optimizer.dll
```

Use matching local test files and DLL paths. Current Windows wrappers build the optimizer; they do not implicitly run all suites mentioned in historical release notes. build_windows.cmd-independent is not an accepted build.py flag in the current modular wrapper. Run available independent test scripts explicitly with their stated dependencies. The full work validation script is tests/work_validation.py in the current source package and was exercised on Linux; it includes lengthy cold/extended runs. Checkpoint correctness tests do not by themselves establish cold-start objective quality.

Reproducibility and documentation basis

This edition preserves the original model-format explanations and examples and adds the current source/help/header controls available in the supplied optimizer source candidate. Main sources are CLI parser/help, optimizer_core.h/.py, BUILD_OPTIMIZER.md, MULTIOBJECTIVE.md, LEXICOGRAPHIC_OBJECTIVES.md, NATIVE_NONLINEAR_SEARCH.md and WORK_CONTINUATION_UPDATE.md. Build-dependent capabilities must be confirmed against the executable actually deployed.

Older help/release documents may retain single-file build commands or descriptions predating named multiple objectives. This manual's modular build chapter and explicitly scoped objective chapters describe the current audited source. No solver source was changed as part of this guide revision. New documentation examples were smoke-checked on the available native Linux CLI/core where stated; Windows execution, external modelling runtimes and a real HiGHS-linked run are not claimed by the documentation update.

Final troubleshooting map

Message / symptom	Recommended action
Unknown new option	Confirm executable path/build; PHP may invoke a different binary than the terminal
CLI and PHP give different objectives	Compare complete argv, seeds, precision, starts/limits, imported model/sidecars, binary/DLL versions, starts and wall-time load
--seconds0 ends quickly	Independent starts, iterations, nodes, work units or schedule have completed; zero time does not disable them
Work objective is worse than timed	Different algorithms/caps; use same-seed cold-start comparison and the successful timed baseline, not blind larger work limits
Checkpoint settings mismatch	Restore matching model/binary/settings or start fresh from reverified full coordinates
A lock directory remains	Confirm no active process before removing stale state; protect backups
Pareto file has fewer rows than points	Duplicates, domination, unavailable/infeasible thresholds and anchor logic affect size; inspect all pass statuses
Higher-priority objective is feasible but hierarchy incomplete	Its pass may not have established numerical optimality; do not optimize a lower priority by assuming completion
CLI JSON flag rejected for scalar mode	Built-in CLI export is blend/Pareto-specific; use typed-core JSON for supported scalar mathematical models
Extended result loses digits in application	Preserve decimal strings or parse floating tokens with a decimal-capable reader; typed-core v1 is double-based
Memory option did not cap all process memory	Frontier/cache estimates are scoped controls, not a total process RSS limit
No candidate	Distinguish resource/numerical limit, unsupported model and established infeasibility; do not display an empty result as an optimum