The Journal of Open Source Software

Uno: a composable framework for nonlinearly constrained optimization

Charlie Vanaret [1¶] **and Alexis Montoison** [*2]

1 Applied Optimization Department, Zuse-Institut Berlin, Germany **2** Mathematics and Computer Science Division, Argonne National Laboratory, USA ¶ Corresponding author

DOI: 10.21105/joss.10229

Software

- Review
- Repository
- Archive

Editor: Vincent Knight

Reviewers:

- @anugrahjo
- @victoraalves

Submitted: 20 February 2026
Published: 14 July 2026

Summary

Uno is a composable software framework for nonlinearly constrained optimization written in modern C++. It unifies the workflows of Lagrange-Newton methods, i.e., gradient-based algorithms that iteratively solve the KKT optimality conditions using Newton's method. The central idea is to decompose these methods into reusable, interchangeable components (constraint reformulation, step computation, globalization techniques, and acceptance criteria) so that classical and hybrid algorithms can be assembled, compared, and tested within a single framework rather than reimplemented as separate solvers. As of July 2026, Uno supports sequential (convex and nonconvex) quadratic programming, interior-point (barrier) methods, sequential linear programming, and unconstrained optimization. For full mathematical details of the algorithms implemented in Uno, see Vanaret & Leyffer (2026).

Uno has interfaces to Julia, Python, C, Fortran, and AMPL, enabling interoperability across scientific computing environments. Precompiled artifacts are available on GitHub, and the solver can be accessed directly via `UnoSolver.jl` in Julia or `unopy` in Python.

Statement of need

Nonlinearly constrained optimization is central to engineering, optimal control (Lewis et al., 2012), machine learning (Sra et al., 2011), scientific modeling (Nocedal & Wright, 2006), and plays a key role in mixed-integer nonlinear optimization (Lee & Leyffer, 2011). The major solution paradigms (sequential quadratic programming, interior-point methods, and augmented Lagrangian methods) are usually developed and implemented as separate solver families. Yet they share the same building blocks: step computation, constraint handling, Hessian models, and globalization strategies. Because these building blocks are rigid inside monolithic codes, algorithmic ideas are typically tested at the level of *complete solvers* rather than *building blocks*: evaluating a new globalization strategy or Hessian approximation means reimplementing an entire solver or intrusively modifying an existing one.

Uno targets two audiences. For **algorithm developers and optimization researchers**, it makes the building blocks explicit and recombinable across paradigms, so that a single new strategy can be swapped into an otherwise state-of-the-art method and benchmarked in isolation. For **practitioners and end users**, it exposes those building blocks, pre-assembled into presets that reproduce established solvers, behind multiple language interfaces. Because these presets live in a single framework with a common interface, users can identify the most suitable method for a given class of problems by reconfiguring strategies within Uno, rather than installing and benchmarking several independent solvers.

*Now at Advanced Micro Devices (AMD).

The gap Uno fills is the absence of a framework in which *production-grade* algorithmic components can be composed across solver paradigms under a unified abstraction.

State of the field

Popular solvers such as IPOPT (Wächter & Biegler, 2006), KNITRO (Byrd et al., 2006), SNOPT (Gill et al., 2005), and WORHP (Büskens & Wassel, 2012) are robust and efficient but monolithic: parameter tuning is exposed while internal components remain rigid. They are designed for end users rather than algorithmic experimentation, and testing a new variant usually requires intrusive modification or reimplementation. A notable exception among production codes is MadNLP.jl (Shin et al., 2024), an actively developed Julia solver, though it remains focused on interior-point methods.

Frameworks that do emphasize modularity occupy a different niche. General-purpose libraries such as `scipy.optimize`, NLopt, pyOpt, and pyOptSparse let users select among complete solvers, but they do not expose the *internal* algorithmic components for recombination. Closest in spirit is modOpt (Joshy & Hwang, 2026), a Python environment that builds optimization algorithms from self-contained modules (line searches, Hessian approximations, merit functions) with an educational focus.

Uno is distinct on three counts. First, its components are production-grade algorithmic building blocks compiled in modern C++. Second, it expresses these paradigms as instances of one generic Lagrange-Newton method, so the same components are reused across all of them rather than confined to a single method. Third, its presets reproduce state-of-the-art solvers at competitive performance (see below), so a component swapped into Uno is measured against a credible baseline rather than a teaching reference.

Software design

Within Uno, strategies are organized into a coherent hierarchy, illustrated in the wheel of strategies (Figure 1). The outer ring lists the high-level *layers* of a Lagrange-Newton method (for example, constraint reformulation and globalization); the middle ring lists the algorithmic *ingredients* that Uno combines automatically to realize each layer; and the inner ring lists the concrete *strategies* available for each ingredient. Strategies currently implemented in Uno are highlighted in green.

In Uno's object-oriented architecture, the ingredients of Figure 1 are abstract classes whose interfaces are implemented by subclasses (the strategies). Uno's simplified UML diagram is shown in Figure 2: inheritance is represented as dashed lines with white arrows, composition as solid lines with black diamonds; abstract classes are written in italic and subclasses in bold. This architecture separates the mathematical logic of the algorithm (problem reformulation, subproblem definition, globalization) from the computational aspects (evaluating the model's functions, solving the subproblems).

Uno implements a generic Lagrange-Newton method in which the abstract classes interact and exchange data while remaining agnostic to the underlying strategies. Strategies are selected by the user at runtime via options. Uno also provides *presets*: particular combinations of strategies and hyperparameters that correspond to state-of-the-art solvers. Two presets are currently available: an `ipopt` preset mimicking the IPOPT solver (Wächter & Biegler, 2006) (a *line-search restoration filter interior-point method with exact Hessian and primal-dual inertia correction*), and a `filtersqp` preset mimicking the filterSQP solver (Fletcher & Leyffer, 1998) (a *trust-region restoration filter SQP method with exact Hessian and no inertia correction*). In principle all combinations of strategies can be generated, although some are not yet supported (e.g., an interior-point method with a trust-region globalization mechanism). When no preset is specified, Uno automatically dispatches to the `filtersqp` or `ipopt` preset based on the problem

size, sparing users the choice of the underlying method. Acting as an oracle, this selection mechanism can be readily extended to dispatch to new presets as they become available.

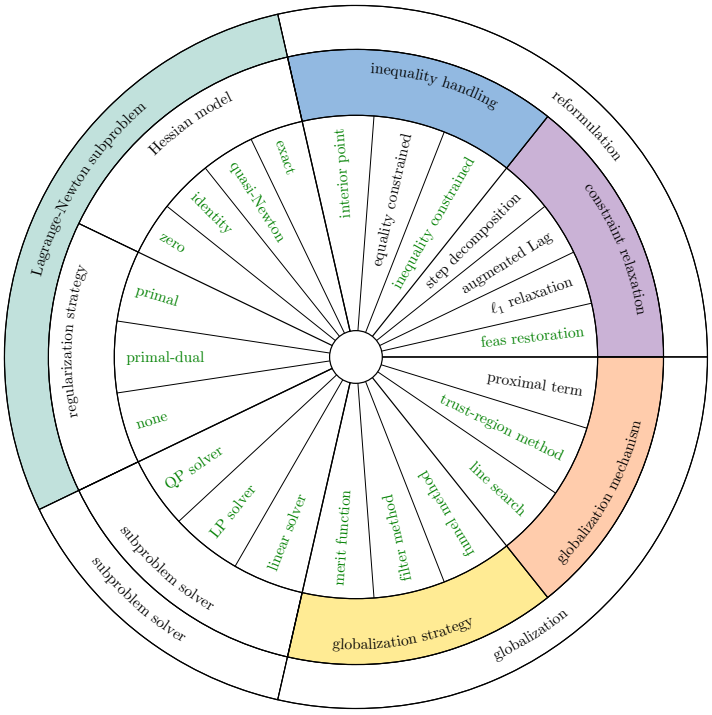

Figure 1: Unification framework: wheel of strategies.

These two presets perform on a par with the state-of-the-art solvers IPOPT (Uno is slightly less robust) and filterSQP (Uno is slightly more robust) in terms of function evaluations on a set of 429 small CUTE instances with fewer than 100 variables and constraints (Bongartz et al., 1995) translated to AMPL by Vanderbei. An up-to-date performance profile is maintained on Uno's documentation page. This benchmark shows that Uno's composability does not come at the cost of performance: assembling a method from modular components reproduces the behavior of the hand-written solvers it mimics.

Subproblem solvers are treated as interchangeable components that can be plugged in or swapped out without modifying the algorithmic logic, letting users match the solver to their problem structure, licensing constraints, or performance requirements. Uno currently interfaces several established LP, QP, and linear solvers: BQPD (Fletcher, 2000), HiGHS (Huangfu & Hall, 2018), MUMPS (Amestoy et al., 2000), MA27 (I. S. Duff & Reid, 1983), MA57 (Iain S. Duff, 2004), MA86 (J. Hogg & Scott, 2010), SSIDS (J. D. Hogg et al., 2016), and `Krylov.jl` (Montoison & Orban, 2023).

Interfaces

To make Uno accessible to a wide range of users, we provide multiple language interfaces.

The AMPL Solver Library (Gay, 1997) interface gives access to the AMPL (Fourer et al., 1989) modeling language for optimization. It is distributed as a binary that takes a compiled AMPL model (.nl file) as input, allowing users to solve problems without directly interacting with the C++ core.

The C interface provides direct access to Uno's core functionality while maximizing

interoperability with other programming languages and tools. The optimization process is expressed as the interaction between two independent opaque objects: a model describing the optimization problem and a solver holding the algorithmic configuration and execution state. The interface is therefore centered around two main structures:

- **Model**: represents an optimization problem and stores information about variables, bounds, constraints, the objective function, and derivative information. Users create a model and set its components (objective, constraints, derivatives, initial primal-dual point).
- **Solver**: represents the algorithm used to solve a given model. Users set solver options, attach callbacks, and access results such as primal and dual solutions, residuals, iteration counts, and performance metrics.

The Fortran interface provides access to Uno through the C API using `iso_c_binding`. It is split into two files: `uno_c.f90` for low-level C bindings, and `uno_fortran.f90` for Fortran-friendly wrappers that handle string arguments. The interface can be included directly in a source file or wrapped in a module for cleaner `use` statements. It is provided as source rather than a precompiled Fortran module (.mod) to maximize portability and interoperability across compilers and platforms.

The Julia interface is distributed as the registered package `UnoSolver.jl`. It integrates Uno into the Julia optimization ecosystem through:

- a thin wrapper around the full C API,
- an interface to `NLPModels.jl` for solving problems following the NLPModels API, such as `ADNLPModels.jl` or `ExaModels.jl`,
- an interface to `MathOptInterface.jl` for handling JuMP models.

Precompiled artifacts are downloaded automatically, making the package plug-and-play without requiring user compilation.

The Python interface is registered on PyPI as `unopy`, providing Uno bindings via precompiled wheels on most platforms.

MATLAB and Rust interfaces are also under development, further expanding Uno's accessibility.

Looking ahead, we plan to integrate Uno as a continuous relaxation solver within MINLP frameworks such as SCIP (Bestuzheva et al., 2023), which will require efficient reoptimization that reuses internal solver state (active sets, factorizations, preallocated workspace) across structurally similar solves.

Research impact statement

Uno has demonstrated value both as a research tool and as a dependency of established software. Its original authors used it to prototype a novel globalization strategy, the funnel method (Kiessling et al., 2025). It has since been used as the lower-level solver in derivative-free bilevel optimization (Cesaroni et al., 2026), been benchmarked on Euclidean problems with bound constraints (Baran et al., 2026), and been cited in (Cederberg et al., 2026; Desef, 2025; Gruss, 2024).

Uno is currently available as a nonlinear optimization solver in:

- Julia:
  - the `JuMP.jl` ecosystem (Dunning et al., 2017),
  - `control-toolbox`, a collection of Julia packages for mathematical control and its applications (Caillau et al., 2024),
- C++:
  - CasADi, an open-source tool for nonlinear optimization and algorithmic differentiation (Andersson et al., 2019),

- Python:
  - pyOptSparse, an object-oriented framework for formulating and solving nonlinear constrained optimization problems (Wu et al., 2020),
  - DNLP, an extension of CVXPY to general nonlinear programming (Cederberg et al., 2026),
- Fortran:
  - CUTEst, the Constrained and Unconstrained Testing Environment with safe threads for optimization software (Gould et al., 2015),
  - IMPL © /IMPL-DATA © by Industrial Algorithms Limited, a modeling and solving platform used in the process industries.

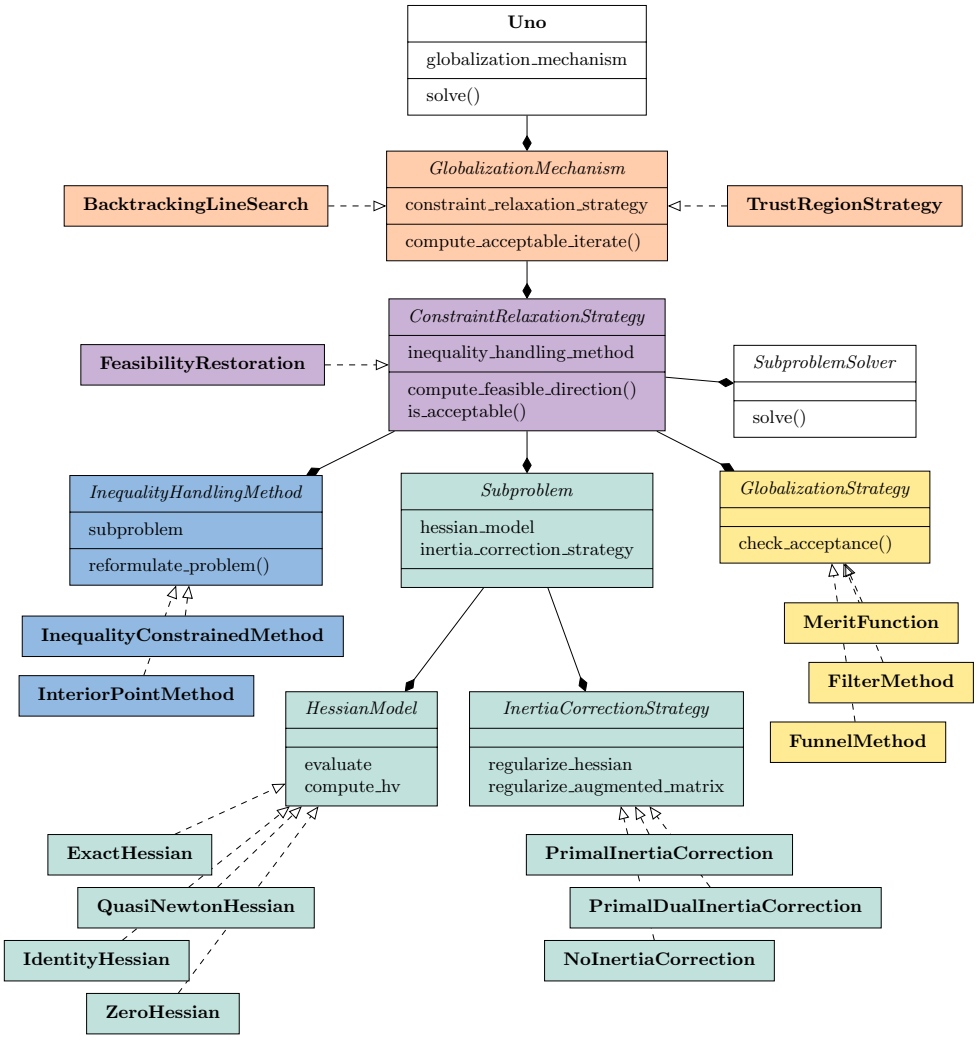

Figure 2: Uno's UML diagram.

Acknowledgments

This work was supported by the Applied Mathematics activity of the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research, under Contract No. DE-AC02-06CH11357. This work was also supported in part by NSF CSSI Grant No. 2104068.

AI usage disclosure

No generative AI was used in the creation of the software, interfaces, implementation of algorithms, or documentation. ChatGPT was used to check spelling, grammar, and clarity of the English text in this paper.

References

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., & Koster, J. (2000). MUMPS: A general purpose distributed memory sparse solver. *International Workshop on Applied Parallel Computing*, 121–130. https://doi.org/10.1007/3-540-70734-4_16

Andersson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., & Diehl, M. (2019). CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, *11*(1), 1–36. https://doi.org/10.1007/s12532-018-0139-4

Baran, M., Bergmann, R., & Przybysz, P. (2026). A Riemannian quasi-Newton algorithm for optimization with Euclidean bounds. *arXiv Preprint arXiv:2605.10573*. https://doi.org/10.48550/arXiv.2605.10573

Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., Van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., & others. (2023). Enabling research through the SCIP optimization suite 8.0. *ACM Transactions on Mathematical Software*, *49*(2), 1–21. https://doi.org/10.1145/3585516

Bongartz, I., Conn, A. R., Gould, N., & Toint, P. L. (1995). CUTE: Constrained and unconstrained testing environment. *ACM Transactions on Mathematical Software (TOMS)*, *21*(1), 123–160. https://doi.org/10.1145/200979.201043

Büskens, C., & Wassel, D. (2012). The ESA NLP solver WORHP. In *Modeling and optimization in space engineering* (pp. 85–110). Springer. https://doi.org/10.1007/978-1-4614-4469-5_4

Byrd, R. H., Nocedal, J., & Waltz, R. A. (2006). KNITRO: An integrated package for nonlinear optimization. In *Large-scale nonlinear optimization* (pp. 35–59). Springer. https://doi.org/10.1007/0-387-30065-1_4

Caillau, J.-B., Cots, O., Gergaud, J., Martinon, P., & Sed, S. (2024). *OptimalControl.jl: a Julia package to model and solve optimal control problems with ODE's*. https://doi.org/10.5281/zenodo.13336563

Cederberg, D., Zhang, W., Nobel, P., & Boyd, S. (2026). Disciplined nonlinear programming. *arXiv Preprint arXiv:2606.02896*. https://doi.org/10.48550/arXiv.2606.02896

Cesaroni, E., Liuzzi, G., & Lucidi, S. (2026). Derivative-free bilevel optimization with inexact lower-level solutions. *arXiv Preprint arXiv:2603.20759*. https://doi.org/10.48550/arXiv.2603.20759

Desef, B. (2025). Optimization techniques in quantum information. *arXiv Preprint arXiv:2512.15831*. https://doi.org/10.48550/arXiv.2512.15831

Duff, Iain S. (2004). MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software (TOMS)*, *30*(2), 118–144. https://doi.org/10.1145/992200.992202

Duff, I. S., & Reid, J. K. (1983). The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw.*, *9*(3), 302–325. https://doi.org/10.1145/356044.356047

Dunning, I., Huchette, J., & Lubin, M. (2017). JuMP: A modeling language for mathematical

optimization. *SIAM Review*, *59*(2), 295–320. https://doi.org/10.1137/15M1020575

Fletcher, R. (2000). Stable reduced Hessian updates for indefinite quadratic programming. *Mathematical Programming*, *87*(2), 251–264. https://doi.org/10.1007/s101070050113

Fletcher, R., & Leyffer, S. (1998). User manual for filterSQP. *Numerical Analysis Report NA/181, Department of Mathematics, University of Dundee, Dundee, Scotland*, *35*.

Fourer, R., Gay, D. M., & Kernighan, B. W. (1989). AMPL: A mathematical programming language. In S. W. Wallace (Ed.), *Algorithms and model formulations in mathematical programming* (pp. 150–151). Springer. https://doi.org/10.1007/978-3-642-83724-1_12

Gay, D. M. (1997). *Hooking your solver to AMPL*. Technical Report 93-10, AT&T Bell Laboratories, Murray Hill, NJ, 1993, revised.

Gill, P. E., Murray, W., & Saunders, M. A. (2005). SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, *47*(1), 99–131. https://doi.org/10.1137/S0036144504446096

Gould, N. I., Orban, D., & Toint, P. L. (2015). CUTEst: A constrained and unconstrained testing environment with safe threads for mathematical optimization. *Computational Optimization and Applications*, *60*(3), 545–557. https://doi.org/10.1007/s10589-014-9687-3

Gruss, L. (2024). *Estimation avancée à horizon glissant, état et paramètres, pour la conduite du coeur des réacteurs PWR* [PhD thesis, Ecole nationale supérieure Mines-Télécom Atlantique]. https://doi.org/10.70675/58a06f96z5f38z4254zad92z861e4b453806

Hogg, J. D., Ovtchinnikov, E., & Scott, J. A. (2016). A sparse symmetric indefinite direct solver for GPU architectures. *ACM Transactions on Mathematical Software (TOMS)*, *42*(1), 1–25. https://doi.org/10.1145/2756548

Hogg, J., & Scott, J. (2010). *An indefinite sparse direct solver for multicore machines*. Tech. Rep. TR-RAL-2010-011, Rutherford Appleton Laboratory, Chilton, Oxfordshire, UK.

Huangfu, Q., & Hall, J. J. (2018). Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, *10*(1), 119–142. https://doi.org/10.1007/s12532-017-0130-5

Joshy, A. J., & Hwang, J. T. (2026). modOpt: A modular development environment and library for optimization algorithms. *Advances in Engineering Software*, *213*, 104084. https://doi.org/10.1016/j.advengsoft.2025.104084

Kiessling, D., Leyffer, S., & Vanaret, C. (2025). A unified funnel restoration SQP algorithm. *Mathematical Programming*, 1–45. https://doi.org/10.1007/s10107-025-02284-3

Lee, J., & Leyffer, S. (2011). *Mixed integer nonlinear programming*. Springer Science & Business Media. https://doi.org/10.1007/978-1-4614-1927-3

Lewis, F. L., Vrabie, D., & Syrmos, V. L. (2012). *Optimal control*. John Wiley & Sons. https://doi.org/10.1002/9781118122631

Montoison, A., & Orban, D. (2023). Krylov.jl: A Julia basket of hand-picked Krylov methods. *The Journal of Open Source Software*, *8*(89), 5187–5187. https://doi.org/10.21105/joss.05187

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization*. Springer. https://doi.org/10.1007/978-0-387-40065-5

Shin, S., Anitescu, M., & Pacaud, F. (2024). Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods. *Electric Power Systems Research*, *236*, 110651. https://doi.org/10.1016/j.epsr.2024.110651

Sra, S., Nowozin, S., & Wright, S. J. (2011). *Optimization for machine learning*. MIT press. https://doi.org/10.7551/mitpress/8996.001.0001

Vanaret, C., & Leyffer, S. (2026). Implementing a unified solver for nonlinearly constrained optimization. *Mathematical Programming Computation*. https://doi.org/10.1007/s12532-026-00310-9

Wächter, A., & Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, *106*(1), 25–57. https://doi.org/10.1007/s10107-004-0559-y

Wu, E., Kenway, G., Mader, C. A., Jasa, J., & Martins, J. R. R. A. (2020). pyOptSparse: A Python framework for large-scale constrained nonlinear optimization of sparse systems. *Journal of Open Source Software*, *5*(54), 2564. https://doi.org/10.21105/joss.02564