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.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 20 February 2026
Published: unpublished

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. As of February 2026, Uno supports sequential (convex and nonconvex) quadratic programming, interior-point (barrier) methods, and sequential linear programming.

Uno breaks down optimization algorithms into reusable modular components such as step computation, constraint reformulation, globalization techniques, and acceptance criteria. This allows classical and hybrid methods to be configured and compared within a single framework. For full mathematical details of the algorithms implemented in Uno, see (Vanaret & Leyffer, 2024).

The core C++ code of Uno is organized into modular, object-oriented components that separate the mathematical logic of the algorithms from implementation details such as memory management, data structures, and computational routines. Uno provides interfaces to Julia, Python, C, Fortran, and AMPL, enabling use 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, machine learning, and scientific modeling (Nocedal & Wright, 2006). It also plays a key role in mixed-integer nonlinear optimization (Lee & Leyffer, 2011).

Typical nonlinear solvers share common building blocks such as step computation, constraint handling, and globalization strategies. These components can be found across major paradigms such as sequential quadratic programming, interior-point methods, and augmented Lagrangian methods, but these are usually developed as separate solver families. Algorithmic ideas are typically tested at the level of complete solvers, rather than individual components.

These observations motivate the development of a unified and composable framework in which algorithmic components are made explicit.

State of the field

Popular solvers such as IPOPT (Wächter & Biegler, 2006), KNITRO (Byrd et al., 2006), and SNOPT (Gill et al., 2005) are robust and efficient, but they are typically monolithic: parameter tuning is exposed while internal components remain rigid. These solvers are primarily designed

for end users rather than algorithmic experimentation. Testing new algorithmic variants usually requires intrusive modification or reimplementation.

Unification framework

Uno enables experimentation at the component level by providing a composable optimization framework in which algorithms are constructed from modular components that reflect mathematical concepts such as step computation, constraint reformulation, and globalization techniques.

Within this framework, strategies are organized into a coherent hierarchy, illustrated in the wheel of strategies (Figure 1): the outer ring represents high-level layers, the middle ring represents algorithmic ingredients that are combined automatically, and the inner ring lists possible strategies for each ingredient. The strategies currently implemented in Uno are highlighted in green.

Figure 1: Unification framework: wheel of strategies.

Software design

In Uno's object-oriented architecture, the ingredients of Figure 1 are abstract classes whose interfaces should be implemented by subclasses (the strategies). Uno's simplified UML diagram is shown in Figure 2. Inheritance is represented as dashed lines with white arrows, while composition is represented as solid lines with black diamonds. Abstract classes are written in italic, while subclasses are written in bold. This modular architecture offers a clear separation between the mathematical logic of the optimization algorithm (the reformulation of the problem, the definition of the subproblem, and the globalization techniques) and the computational aspects (evaluating the model's functions, and solving the subproblems).

Uno implements a generic Lagrange-Newton method in which the abstract classes interact with

one another and exchange data, while being agnostic of the underlying strategies. Strategies are picked by the user at runtime via options. Uno also implements presets, that is particular combinations of strategies (and sets of hyperparameters) that correspond to state-of-the-art solvers. Uno currently implements an `ipopt` preset that mimics 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 that mimics the filterSQP solver (Fletcher & Leyffer, 1998) (a *trust-region restoration filter SQP method with exact Hessian and no inertia correction*). While, in theory, all combinations of strategies can be generated, some are not supported yet (e.g., interior-point method with a trust-region constraint).

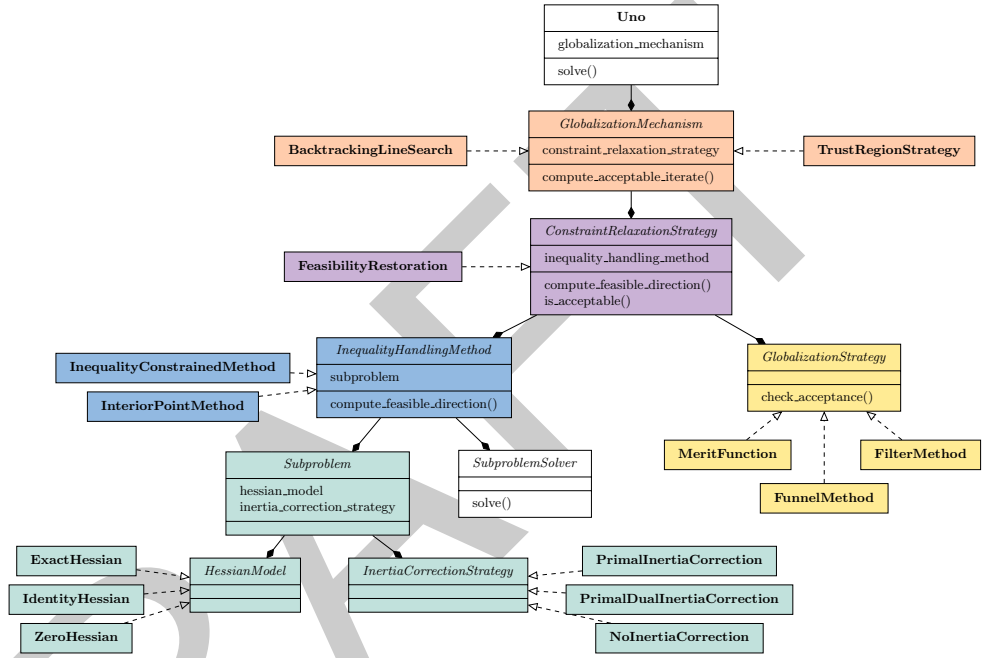

Figure 2: Uno's UML diagram.

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., 1990) 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.

A MATLAB interface is also under development, further expanding Uno's accessibility.

Research impact statement

Uno's `ipopt` and `filtersqp` presets currently 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 (Bongartz et al., 1995). An up-to-date performance profile is maintained on Uno's GitHub `README` page.

Uno's ongoing developments were presented at several international conferences over the years (ISMP 2018, SIAM OP 2021, ICCOPT 2022, ISMP 2024, and ICCOPT 2025), and were the subject of invited talks at Zuse-Institut Berlin (2020), Argonne National Laboratory (2022), and KU Leuven (2025). This resulted in scientific cooperations with the MECO team at KU Leuven (Kiessling et al., 2024, 2025) and the HiGHS team in Edinburgh.

Uno is currently used as a nonlinear optimization solver in:

- the `JuMP.jl` ecosystem,
- DNLP, an extension of CVXPY to general nonlinear programming,
- `Vecchia.jl`, a package for Gaussian processes approximation,
- FelooPy, a user-friendly tool for coding, modeling, and solving decision problems,
- IMPL © /IMPL-DATA © by Industrial Algorithms Limited, a modeling and solving platform used in the process industries especially suited for economic, efficiency and emissions optimization and estimation.

Ongoing discussions and community interest indicate potential future integrations in CasADi, Pyomo, pyOptSparse, Minotaur, and the NEOS Server.

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

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

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

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. (1990). AMPL: A mathematical programming language. *Management Science*, *36*(5), 519–554. 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

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

Kiessling, D., Vanaret, C., Astudillo, A., Decré, W., & Swevers, J. (2024). An almost feasible sequential linear programming algorithm. *2024 European Control Conference (ECC)*, 2328–2335. https://doi.org/10.23919/ECC64448.2024.10590864

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

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

Vanaret, C., & Leyffer, S. (2024). *Implementing a unified solver for nonlinearly constrained optimization*.

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