Les Cahiers du GERAD ISSN: 0711–2440

NomadBBO: A direct search framework for constrained mixed-variable blackbox optimization in Python

E. Hallé-Hannan, C. Tribes

G–2026–73

October 2026

La collection *Les Cahiers du GERAD* est constituée des travaux de recherche menés par nos membres. La plupart de ces documents de travail a été soumis à des revues avec comité de révision. Lorsqu'un document est accepté et publié, le pdf original est retiré si c'est nécessaire et un lien vers l'article publié est ajouté.

Citation suggérée : E. Hallé-Hannan, C. Tribes (Octobre 2026). NomadBBO: A direct search framework for constrained mixed-variable blackbox optimization in Python, Rapport technique, Les Cahiers du GERAD G– 2026–73, GERAD, HEC Montréal, Canada.

Avant de citer ce rapport technique, veuillez visiter notre site Web (https://www.gerad.ca/fr/papers/G-2026-73) afin de mettre à jour vos données de référence, s'il a été publié dans une revue scientifique.

The series *Les Cahiers du GERAD* consists of working papers carried out by our members. Most of these pre-prints have been submitted to peer-reviewed journals. When accepted and published, if necessary, the original pdf is removed and a link to the published article is added.

Suggested citation: E. Hallé-Hannan, C. Tribes (October 2026). NomadBBO: A direct search framework for constrained mixed-variable blackbox optimization in Python, Technical report, Les Cahiers du GERAD G–2026–73, GERAD, HEC Montréal, Canada.

Before citing this technical report, please visit our website (https://www.gerad.ca/en/papers/G-2026-73) to update your reference data, if it has been published in a scientific journal.

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2026
– Bibliothèque et Archives Canada, 2026

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2026
– Library and Archives Canada, 2026

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

NomadBBO: A direct search framework for constrained mixed-variable blackbox optimization in Python

Edward Hallé-Hannan
Christophe Tribes

Département de mathématiques et de génie industriel, Polytechnique Montréal & GERAD, Montréal (Qc), Canada, H3T 1J4

edward.halle-hannan@polymtl.ca
christophe.tribes@polymtl.ca

October 2026
Les Cahiers du GERAD
G–2026–73
Copyright © 2026 Hallé-Hannan, Tribes

Les textes publiés dans la série des rapports de recherche *Les Cahiers du GERAD* n'engagent que la responsabilité de leurs auteurs. Les auteurs conservent leur droit d'auteur et leurs droits moraux sur leurs publications et les utilisateurs s'engagent à reconnaître et respecter les exigences légales associées à ces droits. Ainsi, les utilisateurs:

- Peuvent télécharger et imprimer une copie de toute publication du portail public aux fins d'étude ou de recherche privée;
- Ne peuvent pas distribuer le matériel ou l'utiliser pour une activité à but lucratif ou pour un gain commercial;
- Peuvent distribuer gratuitement l'URL identifiant la publication.

Si vous pensez que ce document enfreint le droit d'auteur, contactez-nous en fournissant des détails. Nous supprimerons immédiatement l'accès au travail et enquêterons sur votre demande.

The authors are exclusively responsible for the content of their research papers published in the series *Les Cahiers du GERAD*. Copyright and moral rights for the publications are retained by the authors and the users must commit themselves to recognize and abide the legal requirements associated with these rights. Thus, users:

- May download and print one copy of any publication from the public portal for the purpose of private study or research;
- May not further distribute the material or use it for any profit-making activity or commercial gain;
- May freely distribute the URL identifying the publication.

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Abstract : Mixed-variable blackbox optimization arises in simulation-based engineering and machine learning, where objective and constraint functions are expensive to evaluate and variables may be continuous, integer, binary, or categorical. This work presents NomadBBO, a user-friendly Python library for inequality-constrained mixed-variable blackbox optimization built around Nomad. Its core optimization framework is CatADS, a new direct-search method that extends Adaptive Direct Search (ADS) to mixed-variable problems. CatADS combines the mechanisms of ADS for quantitative variables with data-driven neighborhoods for categorical variables. These neighborhoods are automatically constructed using Gaussian process-based or empirical Wasserstein distances derived from available data. Inequality constraints are handled through the progressive barrier. The mixed-variable and surrogate-based mechanisms are implemented in Python and connected to the efficient C++ backend of Nomad via Cython. This integration enables Nomad to handle categorical variables and supports hybridization with other Python packages, including SMT 2.0 for Gaussian process modeling and Bayesian optimization. Numerical experiments compare NomadBBO with other solvers on constrained and unconstrained mixed-variable problems from the Cat-Suite benchmark collection. The results show that NomadBBO generally outperforms the compared solvers. The beta release is available at https://test.pypi.org/project/NomadBBO/.

Keywords : Optimization software; blackbox optimization; derivative-free optimization; constrained optimization; mixed-variable problems

Acknowledgements: This research is funded by a Natural Sciences and Engineering Research Council of Canada (NSERC) PhD Excellence Scholarship (PGS D) and a Fonds de Recherche du Québec (FRQNT) PhD Excellence Scholarship. The authors thank Dr. Paul Saves for his assistance with the integration of SMT into NomadBBO.

1 Introduction

Blackbox optimization problems arise in many simulation-based engineering and hyperparameter tuning applications. These problems often involve variables of different types, including continuous, integer, binary, and categorical variables, and may also be subject to constraints. Several Python packages, such as PyMOO, Optuna, and BoTorch, provide flexible and accessible tools for mixed-variable blackbox optimization. These packages offer flexible and user-friendly interfaces, but lack theoretical convergence guarantees. Other solvers such as Nomad provide convergence guarantees, but their core implementations rely on lower-level languages such as C++. This offers high computational performance, but many users prefer accessible and user-friendly software for prototyping and experimentation. Moreover, the latest and maintained version of Nomad does not directly support categorical variables.

This work presents NomadBBO, a Python package for constrained mixed-variable blackbox optimization that supports continuous, integer, binary, and categorical variables. NomadBBO is built on the well-known Nomad software. However, NomadBBO is not simply a Python interface to Nomad: it is a Python framework integrated with Nomad. Several optimization components are implemented directly in Python. These include mechanisms for handling categorical variables and integrations with external libraries such as SMT 2.0 for surrogate modeling. The Python software layer extends the native capabilities of Nomad and interacts with its C++ backend throughout the optimization via Cython. The package is easy to use, computationally efficient, and its optimization methods provide theoretical convergence guarantees. NomadBBO implements state-of-the-art methods from the derivative-free optimization literature, including direct-search methods and Bayesian optimization. Its modular design enables optimization methods and algorithmic components to be readily combined and hybridized.

NomadBBO is currently available as a beta Python package through TestPyPI. The package is under active development, with ongoing testing and improvements toward its first stable release. Users are encouraged to test the beta release and report any issues or unexpected behavior to edward.halle-hannan@polymtl.ca.

Notation

Vectors are in bold, and scalars are in normal font. Column vectors are denoted using parentheses, *e.g.*, $(a, b, c) = [a, b, c]^\top$. Superscripts are reserved for types of variables. Subscripts without parentheses index variables, *e.g.*, x_1^{cat} is the first categorical variable. Subscripts with parentheses are reserved for indexing an iteration k, *e.g.* $\boldsymbol{x}_{(k)}$, or for listing points, *e.g.* $\boldsymbol{x}_{(1)}, \boldsymbol{x}_{(2)}, \ldots, \boldsymbol{x}_{(s)}$. Sets are denoted by capital letters in either roman or calligraphic font, *e.g.*, A or $\mathcal{A}$. The set of known points is an exception and is denoted by $\mathbb{X}_{(k)}$ at iteration k.

1.1 Problem formulations: classes of problems considered

This work considers inequality-constrained mixed-variable blackbox optimization problems of the form

$$\min_{\boldsymbol{x} \in \Omega} f(\boldsymbol{x}), \tag{1}$$

where $f : \mathcal{X} \to \overline{\mathbb{R}}$, with $\overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty\}$, is the objective function, $\mathcal{X}$ is the domain of the objective and constraint functions, and $\boldsymbol{x} \in \mathcal{X}$ denotes a point. The feasible set is defined as

$$\Omega \coloneqq \{\boldsymbol{x} \in \mathcal{X} : g_j(\boldsymbol{x}) \leq 0,\ j \in J\}, \tag{2}$$

where $g_j : \mathcal{X} \to \overline{\mathbb{R}}$ is the j-th inequality constraint function and J is the index set of the constraints.

The objective and constraint functions are assumed to be *blackboxes*, defined as follows in [8]: "*any process that when provided an input, returns an output, but the inner workings of the process are not analytically available*". In aircraft design, complex simulations are used to find optimal designs.

Such simulations take input variables describing a design, including its geometry and material choices, and return outputs corresponding to objective and constraint values, such as fuel consumption [23]. Typically, the execution of these simulations are computationally expensive [2]. Moreover, since the underlying process is inaccessible, derivatives with respect to the continuous variables are unavailable.

The execution of a blackbox can crash or fail to return a valid value because of *hidden constraints* [14], *e.g.*, an unexpected division by zero. In that case, the point $\boldsymbol{x} \in \mathcal{X}$ is assigned the value $f(\boldsymbol{x}) = +\infty$. The inequality constraint functions in Problem (1) are assumed to be *relaxable* and *quantifiable*. They can be evaluated and return a finite value even when they are violated [21]. Consequently, points violating these constraints can still be considered during the optimization. Their constraint violations provide useful information that can be exploited to find feasible solutions.

The problems considered can have any combination of continuous (cont), integer (int), binary (bin) or categorical (cat) variables. Variables of the same type are grouped into a column vector, referred to as a *component*. Following the notation of [6], for each $t \in \{\text{cat}, \text{int}, \text{cont}\}$, the component of type t is given by

$$\boldsymbol{x}^t \coloneqq \left(x_1^t, x_2^t, \ldots, x_{n^t}^t\right) \in \mathcal{X}^t \coloneqq \mathcal{X}_1^t \times \mathcal{X}_2^t \times \ldots \times \mathcal{X}_{n^t}^t, \tag{3}$$

where $n^t \in \mathbb{N}$ denotes the number of variables of type t. For each $i \in I^t \coloneqq \{1, \ldots, n^t\}$, the variable x_i^t takes values in the set $\mathcal{X}_i^t$. The Cartesian product of the sets $\mathcal{X}_i^t$ defines the set of type t, denoted $\mathcal{X}^t$. A point is formed by concatenating the components of each type, and the domain is the Cartesian product of the corresponding sets, such that

$$\boldsymbol{x} \coloneqq \left(\boldsymbol{x}^{\text{cat}}, \boldsymbol{x}^{\text{bin}}, \boldsymbol{x}^{\text{int}}, \boldsymbol{x}^{\text{cont}}\right) \in \mathcal{X} \coloneqq \mathcal{X}^{\text{cat}} \times \mathcal{X}^{\text{bin}} \times \mathcal{X}^{\text{int}} \times \mathcal{X}^{\text{cont}}. \tag{4}$$

Categorical variables take qualitative values commonly refereed to as *categories* or *levels*. For each $i \in I^{\text{cat}}$, the variable x_i^{cat} takes values in the set $\mathcal{X}_i^{\text{cat}} \coloneqq \{c_1, c_2, \ldots, c_{\ell_i}\}$, where ℓ_i denotes the number of categories of the variable. A typical example is the material in an engineering design, which could take a category from the set {Steel, Aluminum, Titanium}. These variables are typically difficult to handle because the sets lack structure, such as natural ordering or standard metrics like the Euclidean distance [16]. The categorical set $\mathcal{X}^{\text{cat}}$ is assumed to be finite and its cardinality is $|\mathcal{X}^{\text{cat}}| = \prod_{i=1}^{n^{\text{cat}}} \ell_i$.

The continuous $\boldsymbol{x}^{\text{cont}} \in \mathcal{X}^{\text{cont}} \subseteq \mathbb{R}^{n^{\text{cont}}}$ and integer $\boldsymbol{x}^{\text{int}} \in \mathcal{X}^{\text{int}} \subseteq \mathbb{Z}^{n^{\text{int}}}$ components are composed of *quantitative* variables. Unlike categorical variables, they belong to ordered sets equipped with standard metrics, making them generally easier to optimize. For convenience, the integer and continuous components are grouped into the quantitative component $\boldsymbol{x}^{\text{qnt}} \coloneqq (\boldsymbol{x}^{\text{int}}, \boldsymbol{x}^{\text{cont}}) \in \mathcal{X}^{\text{qnt}} \coloneqq \mathcal{X}^{\text{int}} \times \mathcal{X}^{\text{cont}}$.

Binary variables take values in the set $\{0, 1\}$. The binary set $\mathcal{X}^{\text{bin}}$ is finite and its cardinality is $\left|\mathcal{X}^{\text{bin}}\right| = 2^{n^{\text{bin}}}$. Although they admit a natural ordering and distance, they are frequently used to model qualitative decisions, such as the presence or absence of a feature in an engineering design. It is not always clear whether they should be viewed as qualitative or quantitative variables: ultimately, this is a modeling choice. Binary variables are treated as a distinct variable type but they are assigned to either the categorical or quantitative component depending on their nature and the modeling choice. Consequently, throughout the rest of the work, a point is represented as

$$\boldsymbol{x} = \left(\boldsymbol{x}^{\text{cat}}, \boldsymbol{x}^{\text{qnt}}\right), \tag{5}$$

where each binary variable is assigned to the categorical or quantitative component.

1.2 Contributions of the work

This work makes two major contributions to derivative-free optimization. The first contribution is methodological: it introduces CatADS, a new direct-search method for mixed-variable blackbox optimization. The second, and primary contribution, is NomadBBO, a Python library that implements CatADS within an accessible, efficient, and modular software framework.

CatADS builds on two recent developments in direct-search optimization. The first is CatMADS, a method for mixed-variable blackbox optimization that extends the well-known Mesh Adaptive Direct Search (MADS) algorithm from quantitative variables to problems with categorical variables. It handles categorical variables with automatically constructed neighborhoods and establishes convergence guarantees in a mixed-variable setting. The second is Adaptive Direct Search (ADS), a recent generalization of MADS for continuous optimization that removes its mesh discretization. The methodological contributions of CatADS are:

1. adapting ADS to handle integer variables;
2. combining this quantitative mechanism with the neighborhood-based mechanism of CatMADS for handling categorical variables;
3. introducing a categorical polling strategy based on Wasserstein distances as a computationally cheaper alternative to the surrogate-based strategy used in [6, 7];
4. removing the MADS mesh from CatMADS by replacing MADS with ADS, facilitating hybridization with other optimization methods such as Bayesian optimization.

These methodological developments form the optimization core of NomadBBO. Although other derivative-free optimization libraries are openly available, they typically lack at least one desirable feature, such as support for mixed-variable or constrained problems, ease of use, computational efficiency, theoretical guarantees, or modularity. NomadBBO was developed to combine these features within a single Python library. Its main characteristics are:

1. **Accessibility**. An easy-to-use Python interface for defining optimization problems and configuring optimization algorithms.
2. **Mixed-variable support**. Support for continuous, integer, binary, and categorical variables in inequality-constrained blackbox optimization problems.
3. **Efficiency**. Derivative-free optimization methods built on the C++ software Nomad for high computational performance.
4. **Theoretical guarantees**. Implementations of optimization methods with established convergence guarantees whenever available.
5. **Modularity and hybridization**. A flexible software architecture that facilitates the integration of new optimization methods and the development of hybrid algorithms.

Numerical experiments on mixed-variable problems from Cat-Suite [18] show promising performance compared with established general-purpose solvers (see Section 4).

1.3 Related work

Several general-purpose software packages have been developed for derivative-free optimization. Table 1 compares five libraries. Optuna [1], BoTorch [24], and PyMOO [13] are Python libraries that support continuous, integer, binary, and categorical variables through accessible interfaces, whereas Nomad is a compiled direct-search solver with convergence guarantees. NomadBBO aims to bridge these two approaches by combining the accessibility and flexibility of a Python library with the algorithmic guarantees and performance of Nomad. The considered features are support for categorical variables, an accessible interface, integration of external optimizers, a compiled backend, support for inequality and equality constraints, and convergence guarantees.

Optuna is a Python framework with a unified interface to several derivative-free optimization algorithms, including Tree-structured Parzen Estimators (TPE), Gaussian process Bayesian optimization, CMA-ES, NSGA-II, and NSGA-III. In practice, it is primarily designed for hyperparameter optimization. Optuna is flexible and easy to use. However, its support for general constrained optimization is limited, and it does not offer convergence guarantees.

Table 1: Comparison of general-purpose derivative-free optimization software. A checkmark indicates full support, while ∼ indicates partial support.

Solver	Categorical variables	Accessible interface	External optimizers	Compiled backend	Inequality constraints	Equality constraints	Guarantees
Optuna	✓	✓	✓		∼		
BoTorch	✓	✓	✓	✓	∼		
PyMOO	✓	✓	✓	∼	✓	✓	
Nomad				✓	✓	✓	✓
NomadBBO	✓	✓	✓	✓	✓		✓

BoTorch is a Python library primarily designed for surrogate modeling and Bayesian optimization. It provides a broad selection of probabilistic surrogate models, particularly Gaussian processes. Acquisition functions and optimization routines can then be combined with these models to construct different Bayesian optimization strategies. BoTorch is built on PyTorch and GPyTorch for efficient computations. It has a modular architecture, but provides limited support for general constrained optimization and does not offer convergence guarantees.

PyMOO is a Python framework providing a large collection of derivative-free optimization algorithms. These include evolutionary algorithms such as genetic algorithms and differential evolution, as well as local methods such as Nelder–Mead and pattern search. The algorithms are themselves modular and can be customized through different optimization components and operators. Most of PyMOO is implemented in Python. Some computationally expensive operations are accelerated through compiled Cython modules. PyMOO is highly versatile and provides extensive support for constrained and mixed-variable optimization. However, it does not have convergence guarantees.

NomadBBO is built on the C++ software Nomad, which serves as a computationally efficient backend within a complete Python framework. The Python and C++ components are fully integrated through Cython, allowing solver components implemented in either language to interact seamlessly within a single optimization workflow. The framework allows users to define optimization problems, configure optimization algorithms, and coordinate the different parts of the solver. NomadBBO also extends Nomad with optimization capabilities that are not available in the original software, such as mixed-variable optimization with categorical variables. The current version supports inequality constraints but not equality constraints, which is a limitation compared with the other Python packages considered here. It provides modular integration with external libraries to develop hybrid optimization algorithms.

Nomad already provides a Python interface through PyNomad. It exposes only a subset of Nomad functionalities and does not support mixed-variable problems with categorical variables. Its problem definitions and solver configuration remain close to the native Nomad syntax. In contrast, NomadBBO extends Nomad through a complete Python framework with additional optimization capabilities. It also provides a simpler and more natural Python workflow for defining and solving problems.

The remainder of this work is organized as follows. Section 2 presents the CatADS optimization framework and its main algorithmic mechanisms. Section 3 describes the software architecture of NomadBBO and the interactions between its Python, Cython, and Nomad layers. Finally, Section 4 assesses the performance of NomadBBO through numerical comparisons with other solvers.

2 Optimization framework: CatADS with the progressive barrier

The optimization framework of NomadBBO is a direct-search algorithm. At each iteration, a direct-search algorithm generates trial points around a current solution to improve the objective values or the feasibility of the constraints. The mechanisms used to generate and accept trial points are described throughout this section.

NomadBBO implements CatADS, a mixed-variable extension of Adaptive Direct Search (ADS) [5]. ADS is a direct-search method for optimization problems with quantitative variables. It generates trial points along a set of directions around the current solution while ensuring that new points remain sufficiently far from previously evaluated points. CatADS extends ADS by combining its directional mechanism for quantitative variables with a neighborhood mechanism for handling categorical variables. The resulting direct-search framework handles continuous, integer, binary, and categorical variables. For simplicity, CatADS is first presented in Algorithm 1 for unconstrained problems. The constraints are added in Section 2.3.

Algorithm 1: The CatADS framework for unconstrained problems (based on [6]).

0. **Initialization**. Set $k = 0$, perform a Design of Experiment (DoE), and initialize the parameters
1. **Search** (optional).
2. **Poll**. Perform polling around the incumbent solution $\boldsymbol{x}_{(k)}$
3. **Extended poll** (optional). If Steps 1 & 2 are unsuccessful, perform quantitative polls around points with objective function values sufficiently close to $f(\boldsymbol{x}_{(k)})$
4. **Update**. Set $k \leftarrow k + 1$, update parameters according to the iteration outcome, and check the stopping criterion

The optimization starts with a Design of Experiments (DoE) that uses part of the evaluation budget to sample points from the domain. The sampled points provide initial information about the problem and determine the initial solution. They are also used to construct categorical neighborhoods. The number of points required for categorical variables depends on the strategy used to construct these neighborhoods.

The main algorithmic steps of CatADS are the same as those of CatMADS [6]. They consist of the *search*, *poll*, and *extended poll* steps. If one of these steps finds a solution with a strictly lower objective function value, this solution becomes the new incumbent and the iteration is said to be *successful*. Otherwise, the iteration is deemed *unsuccessful*. The search is an optional and flexible step that allows additional trial points to be evaluated before the poll with any strategy. The poll is the mechanism that provides the theoretical guarantees of CatADS. At iteration k, trial points are generated around the incumbent $\boldsymbol{x}_{(k)}$ through a *quantitative poll* and a *categorical poll*. The quantitative poll fixes the categorical variables and generates trial points along directions in the quantitative domain. The categorical poll fixes the quantitative variables and evaluates categorical components selected from their neighborhoods. If there are no categorical variables, the categorical poll is omitted and the poll reduces to the quantitative poll.

If both the search and poll steps fail to improve the objective, the optional *extended poll* may be performed at Step 3. It performs additional quantitative polls around points from the categorical poll that almost, but did not, improve the incumbent value $f(\boldsymbol{x}_{(k)})$. These promising points result from changes to the incumbent categorical component, and the extended poll adjusts their quantitative variables to potentially find a better point.

After each iteration, the direct-search parameters are updated according to the iteration outcome. The optimization terminates when one of the stopping criteria is satisfied.

ADS generalizes the convergence framework of MADS [5], and CatMADS extends it to mixed-variable problems with categorical variables [6]. The convergence properties of CatADS therefore follow directly from those of ADS and CatMADS [5, 6]. These properties involve several results on the behavior of sequences of unsuccessful incumbents. For the categorical variables, this requires the mechanism used to construct the neighborhoods to remain fixed after some iteration. With these variables fixed, the accumulation point is Clarke stationary with respect to the continuous variables when the objective function is locally Lipschitz continuous with respect to them. These properties provide convergence guarantees under mild assumptions and motivate natural stopping criteria for the algorithm.

A detailed theoretical analysis is outside the scope of this work. That said, most arguments follow directly from CatMADS [6] by replacing the analysis of the quantitative variables in MADS with the corresponding ADS results.

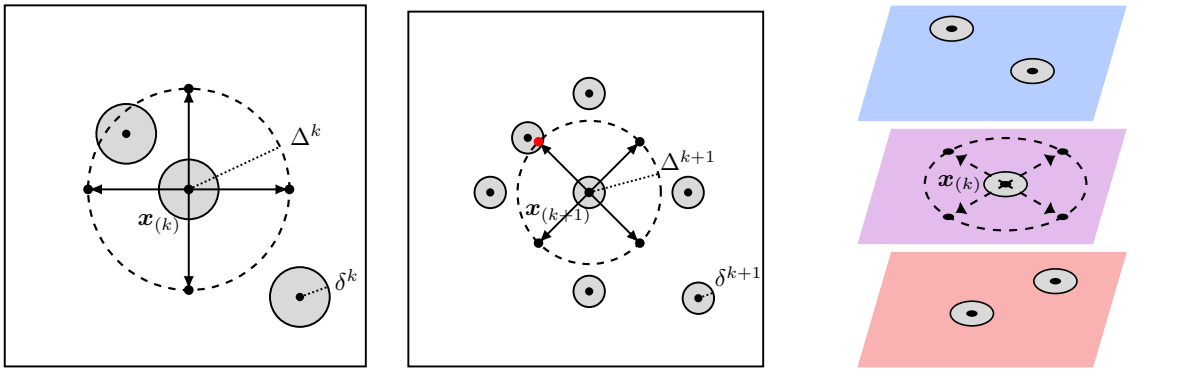
The next subsections detail the main components of CatADS. Section 2.1 first presents ADS in the continuous setting, and Section 2.2 extends its poll mechanism to quantitative variables. Section 2.3 then introduces the constraint-handling strategies used by NomadBBO. Section 2.4 presents the categorical distances used to construct neighborhoods, and Section 2.5 finally describes the categorical poll.

2.1 Overview of ADS

The quantitative poll of CatADS builds on ADS. Before detailing the quantitative poll, ADS is presented in a strictly continuous setting, with $\mathcal{X} = \mathcal{X}^{\text{cont}} \subset \mathbb{R}^{n^{\text{cont}}}$. ADS defines a *punctured space* to prevent new trial points from being evaluated too close to previously evaluated points. At iteration k, let $\mathbb{X}_{(k)}$ denote all points evaluated since the beginning of the optimization, including the DoE points. This set is referred to as the *set of known points* and it is non-empty. Around each known point, an exclusion region of radius $\delta_{(k)}$ is defined, where $\delta_{(k)} > 0$ is the *exclusion size parameter*. The punctured space contains all points outside these exclusion regions and is defined by

$$\mathring{\mathcal{X}}_{(k)} = \left\{ \boldsymbol{x} \in \mathcal{X} : \|\boldsymbol{x} - \boldsymbol{y}\| \geq \delta_{(k)} \text{ for all } \boldsymbol{y} \in \mathbb{X}_{(k)} \right\} \subset \mathbb{R}^{n^{\text{cont}}}. \tag{6}$$

Figure 1a illustrates a punctured space $\mathring{\mathcal{X}}_{(k)} \subset \mathbb{R}^2$ with three known points. The gray regions correspond to their exclusion regions. The remaining white region corresponds to the punctured space, where points can be evaluated at iteration k. The set of known points grows as new evaluations are performed. When the exclusion size is decreased, new points can be evaluated closer to the known points.



(a) Poll $P_{(k)}$ with $\delta_{(k)} \leq \Delta_{(k)}$. **(b) Poll $P_{(k+1)}$ after an unsuccessful iteration k.** **(c) Quantitative poll with the categorical component fixed.**

Figure 1: Illustration of the quantitative poll. Panels (a) and (b) show two consecutive ADS polls in a 2D continuous space. Red points correspond to poll points that are not evaluated. Panel (c) shows the quantitative poll in a mixed-variable domain with the categorical component fixed.

The *poll* is the main mechanism used by ADS to generate trial points. It generates these points around the incumbent using a set of normalized directions. Let $\mathbb{D}_{(k)}$ denote this set of directions at iteration k. The directions form a positive spanning set to ensure sufficient coverage around the incumbent [5]. At each iteration, a single random normalized vector is generated. A Householder transformation built from this random vector is then used to construct the positive spanning set of poll directions.

A second parameter $\Delta_{(k)} > 0$, called the *poll size parameter*, determines the distance between the incumbent and the trial points. The poll is defined by

$$P_{(k)} = \left\{ \boldsymbol{x}_{(k)} + \Delta_{(k)} \boldsymbol{d} : \boldsymbol{d} \in \mathbb{D}_{(k)} \right\}, \tag{7}$$

Since the directions are normalized, every point in the poll $P_{(k)}$ is at distance $\Delta_{(k)}$ from the incumbent $\boldsymbol{x}_{(k)}$. Only poll points that belong to the punctured space $\mathring{\mathcal{X}}_{(k)}$ are evaluated. Figure 1a illustrates this mechanism in $\mathbb{R}^2$ with four directions.

Figures 1a and 1b illustrates two consecutive polls. Iteration k is unsuccessful since no point in the poll improves the incumbent. Consequently, the incumbent remains unchanged, and the poll size and exclusion size are decreased, leading to a smaller poll and smaller exclusion regions at iteration $k+1$. This forces the algorithm to generate points closer to the incumbent at the next iteration. Consecutive unsuccessful iterations therefore make the poll increasingly local around the incumbent. The exclusion size satisfies $\delta_{(k)} \leq \Delta_{(k)}$ and must decrease faster than the poll size as both sizes decrease. In ADS, this is ensured by setting

$$\delta_{(k)} = \min \left\{ \Delta_{(k)}, \frac{(\Delta_{(k)})^2}{\Delta_{(0)}} \right\}. \tag{8}$$

This relative decrease of the exclusion size is required for the convergence guarantees of ADS. For more details, see [5].

2.2 Quantitative poll

The continuous formulation of ADS is now extended to the quantitative domain of CatADS. Consider first a quantitative domain with continuous and integer variables, but no categorical variables, such that $\mathcal{X} = \mathcal{X}^{\text{qnt}} = \mathcal{X}^{\text{int}} \times \mathcal{X}^{\text{cont}}$. To handle integer and continuous variables together, each variable is assigned its own exclusion and poll sizes. This approach comes from anisotropic MADS and the granular mesh version of MADS (G-MADS) [11, 12]. At iteration k, let

$$\boldsymbol{\delta}_{(k)} = \left(\delta_{(k),1}, \ldots, \delta_{(k),n^{\text{qnt}}}\right), \qquad \boldsymbol{\Delta}_{(k)} = \left(\Delta_{(k),1}, \ldots, \Delta_{(k),n^{\text{qnt}}}\right) \tag{9}$$

denote the *exclusion and poll size vectors*, respectively. The i-th quantitative variable x_i^{qnt} is associated with the exclusion size $\delta_{(k),i}$ and the poll size $\Delta_{(k),i}$.

The allowable poll sizes depend on the variable type. For integer variables, $\Delta_{(k),i} \in \{a \times 10^b : a \in \{1,2,5\},\, b \in \mathbb{N}\}$, with $\Delta_{(k),i} \geq 1$, for $i \in I^{\text{int}}$. The poll size of integer variables cannot be smaller than one since they are whole numbers. For continuous variables, $\Delta_{(k),i} \in \{a \times 10^b : a \in \{1,2,5\},\, b \in \mathbb{Z}\}$ for $i \in I^{\text{cont}}$.

The poll sizes are updated according to the iteration outcome. They are decreased after unsuccessful iterations and increased after successful ones. Their independent variable-wise management allows each poll size to adapt to the type and scale of its corresponding variable. Detailed update rules are given for anisotropic MADS and G-MADS in [11, 12].

For each quantitative variable, the exclusion size is defined from its poll size using the same rule as in Equation (8). This rule is applied component-wise to $\boldsymbol{\delta}_{(k)}$ and $\boldsymbol{\Delta}_{(k)}$, so that $\delta_{(k),i} \leq \Delta_{(k),i}$ for every $i \in I^{\text{qnt}}$. With these variable-wise exclusion sizes, the exclusion regions and punctured space are defined independently along each quantitative variable. For a known point $\boldsymbol{y} \in \mathbb{X}_{(k)}$, its exclusion region is defined by

$$\mathcal{R}_{(k)}(\boldsymbol{y}) = \left\{ \boldsymbol{x} \in \mathcal{X} : |x_i - y_i| < \delta_{(k),i} \quad \text{for all } i \in I^{\text{qnt}} \right\}. \tag{10}$$

The exclusion regions of all known points are removed from the quantitative domain. The punctured space is therefore

$$\mathring{\mathcal{X}}_{(k)} = \mathcal{X} \setminus \bigcup_{\boldsymbol{y} \in \mathbb{X}_{(k)}} \mathcal{R}_{(k)}(\boldsymbol{y}). \tag{11}$$

For the quantitative domain, Equation (7) is generalized by replacing the scalar poll size $\Delta_{(k)}$ with the diagonal matrix $\operatorname{diag}\left(\boldsymbol{\Delta}_{(k)}\right)$. A point in the quantitative poll is therefore written as $\boldsymbol{x}_{(k)} + \operatorname{diag}\left(\boldsymbol{\Delta}_{(k)}\right) \boldsymbol{d}$,

so that each component of the direction is scaled by the poll size of its corresponding quantitative variable.

Finally, categorical variables are added to recover the mixed-variable setting. The generalization of the quantitative poll to the mixed-variable setting is straightforward. Figure 1c illustrates the punctured space in a mixed-variable domain with one categorical variable $x^{\text{cat}} \in \{\text{Blue}, \text{Purple}, \text{Red}\}$. For the quantitative poll, the categorical variables are fixed at their incumbent values and the directions are constructed only from the quantitative variables. Figure 1c shows an example of a quantitative poll with the categorical variable fixed. The quantitative poll is effectively only modifying quantitative variables.

2.3 Handling inequality constraints with the progressive barrier

NomadBBO handles inequality constraints using the progressive barrier (PB) [4]. The PB allows infeasible trial points to be considered while progressively restricting their accepted constraint violation. For inequality constraints $g_j(\boldsymbol{x}) \leq 0$, $j \in J$, the infeasibility of an evaluated point is quantified by the *constraint aggregation function* $h : \mathcal{X} \to \overline{\mathbb{R}}$ defined as

$$h(\boldsymbol{x}) = \begin{cases} \sum_{j \in J} \left(\max\{0, g_j(\boldsymbol{x})\}\right)^2 & \text{if } \boldsymbol{x} \in \mathcal{X}, \\ \infty & \text{otherwise,} \end{cases} \tag{12}$$

By construction, $h(\boldsymbol{x}) \geq 0$, and a point is feasible if and only if $h(\boldsymbol{x}) = 0$.

The main idea of the PB is to work with two incumbent solutions, each with its own independent poll, as described above. The *feasible poll* is performed around the *feasible incumbent* $\boldsymbol{x}_{\text{FEA}} \in \Omega$, which is the best known feasible solution. The *infeasible poll* is performed around the *infeasible incumbent* $\boldsymbol{x}_{\text{INF}} \in \mathcal{X} \setminus \Omega$, which is the best known infeasible solution within an accepted level of constraint violation. This accepted infeasibility is controlled at iteration k by the *barrier* $h_{(k)}^{\max} \geq 0$, such that $h(\boldsymbol{x}_{\text{INF}}) \leq h_{(k)}^{\max}$. The barrier is progressively tightened throughout the optimization. The feasible and infeasible incumbents are formally defined by

$$\boldsymbol{x}_{\text{FEA}} \in \operatorname{argmin}\left\{f(\boldsymbol{x}) : \boldsymbol{x} \in \Omega \cap \mathbb{X}_{(k)}\right\} \quad \text{and} \quad \boldsymbol{x}_{\text{INF}} \in \operatorname{argmin}\left\{f(\boldsymbol{x}) : 0 < h(\boldsymbol{x}) \leq h_{(k)}^{\max},\ \boldsymbol{x} \in \mathbb{X}_{(k)}\right\}, \tag{13}$$

where $\mathbb{X}_{(k)}$ denotes the set of known points. The barrier rejects candidate points from the infeasible poll whose constraint violation exceeds $h_{(k)}^{\max}$. When both incumbents are available, the poll is the union of the feasible and infeasible polls,

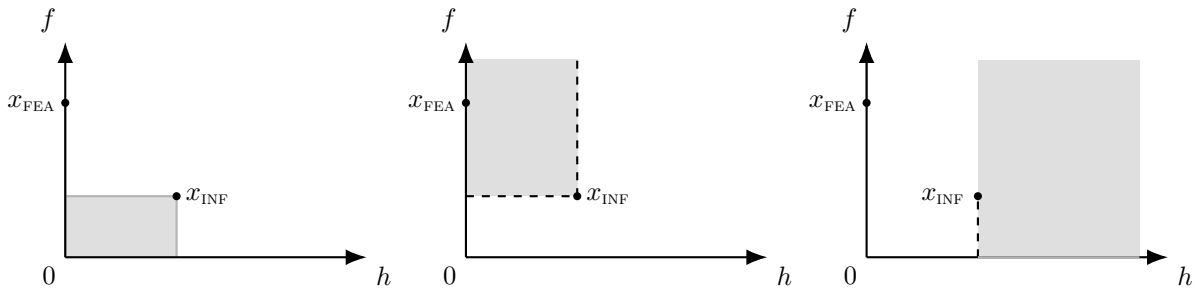
$$P_{(k)} = P_{\text{FEA}} \cup P_{\text{INF}}. \tag{14}$$

The incumbent updates rely on dominance in the (h, f) space. Among feasible points, a point dominates another if it has a lower objective value. Among infeasible points, a point $\boldsymbol{x}$ dominates a point $\boldsymbol{y}$ if

$$h(\boldsymbol{x}) \leq h(\boldsymbol{y}) \quad \text{and} \quad f(\boldsymbol{x}) \leq f(\boldsymbol{y}), \tag{15}$$

with at least one strict inequality. Based on this dominance relation, an iteration with the PB has three possible outcomes, as illustrated in Figure 2. In each subfigure, the horizontal and vertical axes represent the constraint violation h and objective value f, respectively. The gray region represents the region of objective and constraint violation values associated with the iteration outcome indicated in the caption. Solid boundaries are included in the gray region, whereas dashed boundaries are excluded.

A *dominating iteration*, illustrated in Figure 2a, occurs when a trial point dominates either the feasible or infeasible incumbent. An *improving iteration*, illustrated in Figure 2b, occurs when no incumbent is dominated, but an infeasible trial point has a lower constraint violation than the current infeasible incumbent. Otherwise, the iteration is *unsuccessful*, as illustrated in Figure 2c. These outcomes determine the updates of the incumbents and of the barrier $h_{(k)}^{\max}$. For more details on the PB updates, see [8, Chapter 12].



(a) Dominating iteration. **(b) Improving iteration.** **(c) Unsuccessful iteration.**

Figure 2: Possible iteration outcomes with the PB (adapted from [17], originally inspired by [8]).

2.4 Categorical distances

The treatment of categorical variables in CatADS follows the CatMADS neighborhood mechanism, where categorical neighborhoods are constructed from problem-dependent distances [6]. These distances determine which categorical components are included in the categorical poll. Two strategies are available to construct them. The first uses mixed-variable GPs to learn similarities between categorical components, as presented in Section 2.4.1. The second constructs empirical Wasserstein distances directly from the observed objective and constraint values, as presented in Section 2.4.2.

2.4.1 GP-based categorical distances

The first strategy constructs GP-based neighborhoods and was introduced in [7]. A mixed-variable GP is fitted to the objective function using the data collected during the optimization. The fitted categorical kernel induces a problem-specific distance $d_f : \mathcal{X}^{\text{cat}} \times \mathcal{X}^{\text{cat}} \to \mathbb{R}_+$ between categorical components. Categorical components with similar behavior with respect to the objective function have smaller distances.

For the constraint functions, one mixed-variable GP is fitted to each constraint. For a constraint g_j, let $\hat{g}_j : \mathcal{X} \to \mathbb{R}$ denote the prediction provided by its associated GP. During the categorical poll, the quantitative variables are fixed at the incumbent values. For notational simplicity, $\hat{g}_j(\boldsymbol{x}^{\text{cat}})$ denotes $\hat{g}_j(\boldsymbol{x}^{\text{qnt}}_{(k)}, \boldsymbol{x}^{\text{cat}})$ throughout this section: $\hat{g}_j$ is considered as a function $\hat{g}_j : \mathcal{X}^{\text{cat}} \to \mathbb{R}$. If two categorical components are predicted to be feasible, comparing their degree of feasibility is not important for constructing the categorical neighborhood [7]. For this reason, the prediction is transformed into a nonnegative violation measure

$$\hat{g}_j^+(\boldsymbol{x}^{\text{cat}}) = \begin{cases} 0, & \text{if } \hat{g}_j(\boldsymbol{x}^{\text{cat}}) \leq 0 \text{ (predicted to be feasible)}, \\ \psi(\hat{g}_j(\boldsymbol{x}^{\text{cat}})), & \text{otherwise}, \end{cases} \tag{16}$$

where ψ normalizes positive predicted violations into $[0, 1]$. Then, a categorical pseudo-distance associated with g_j is defined by

$$d_j(\boldsymbol{x}^{\text{cat}}, \boldsymbol{y}^{\text{cat}}) = \left| \hat{g}_j^+(\boldsymbol{x}^{\text{cat}}) - \hat{g}_j^+(\boldsymbol{y}^{\text{cat}}) \right|, \tag{17}$$

which is, by construction, zero when both $\boldsymbol{x}^{\text{cat}}$ and $\boldsymbol{y}^{\text{cat}}$ are predicted to satisfy g_j. The pseudo-distances associated with all constraint functions are then aggregated into the constraint pseudo-distance

$$d_{\boldsymbol{g}}(\boldsymbol{x}^{\text{cat}}, \boldsymbol{y}^{\text{cat}}) = \left(\sum_{j \in J} d_j(\boldsymbol{x}^{\text{cat}}, \boldsymbol{y}^{\text{cat}})^p \right)^{1/p}, \qquad p \geq 1. \tag{18}$$

Categorical components with similar predicted constraint violations therefore have smaller pseudo-distances.

When the GPs associated with the objective and constraint functions are updated, the corresponding categorical distances can also be updated. This allows the distances to adapt as new evaluations are collected during the optimization.

2.4.2 Empirical Wasserstein categorical distances

The second strategy constructs categorical distances directly from the known points. For each categorical variable, the objective values associated with each of its categories define an empirical distribution. Figure 3 illustrates the empirical distributions of the objective values for three categories of a categorical

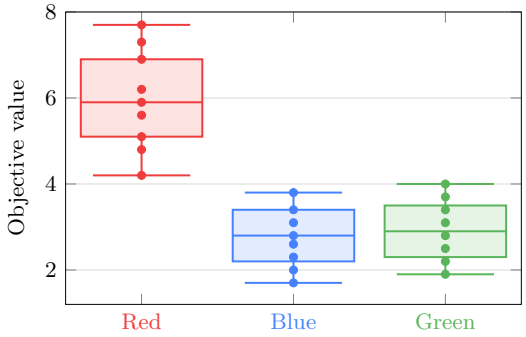


Figure 3: Empirical distributions of objective values for three categories.

variable. The distributions associated with the Blue and Green categories are similar, while the Red category is more distinct. The empirical Wasserstein distance is used to quantify these differences between categories.

Wasserstein distances are first constructed for each categorical variable and then aggregated into a single distance between categorical components. Constructing a categorical distance directly for complete categorical components would require observations for every possible categorical component. The number of possible components grows as the product of the numbers of categories per variable, and many of them may therefore never be observed during the optimization. The variable-wise construction instead relies on observations for the individual categories of each variable. The number of individual categories that must be represented therefore grows with the sum of the numbers of categories per variable, rather than their product.

The construction is first presented for the objective function, followed by the construction used for the constraints. Recall that $\mathbb{X}_{(k)}$ denotes the set of known points. For a given categorical variable x_i^{cat}, consider one of its categories a. Then, let $F_{i,a}(z)$ denote the fraction of known objective values associated with category a that are smaller than or equal to z:

$$F_{i,a}(z) = \frac{\left|\left\{\boldsymbol{x} \in \mathbb{X}_{(k)} : x_i^{\text{cat}} = a \text{ and } f(\boldsymbol{x}) \leq z\right\}\right|}{\left|\left\{\boldsymbol{x} \in \mathbb{X}_{(k)} : x_i^{\text{cat}} = a\right\}\right|}. \tag{19}$$

Next, consider another category b of the same categorical variable, with $F_{i,b}$ defined as in Equation (19). Let $z_{(1)} < z_{(2)} < \cdots < z_{(L)}$ denote the sorted distinct objective values observed with either $x_i^{\text{cat}} = a$ or $x_i^{\text{cat}} = b$ in $\mathbb{X}_{(k)}$. For the i-th categorical variable x_i^{cat}, the empirical one-dimensional Wasserstein distance between the two categories is

$$d_i(a,b) = \sum_{\ell=1}^{L-1} \left|F_{i,a}(z_{(\ell)}) - F_{i,b}(z_{(\ell)})\right| \left(z_{(\ell+1)} - z_{(\ell)}\right). \tag{20}$$

where $\left|F_{i,a}(z_{(\ell)}) - F_{i,b}(z_{(\ell)})\right|$ measures how differently the two categories distribute objective values up to $z_{(\ell)}$, and $z_{(\ell+1)} - z_{(\ell)}$ is the distance between two consecutive and distinct observed objective values with $x_i^{\text{cat}} = a$ or $x_i^{\text{cat}} = b$ in $\mathbb{X}_{(k)}$.

The one-dimensional distances are then aggregated into a distance $d_f : \mathcal{X}^{\text{cat}} \times \mathcal{X}^{\text{cat}} \to \mathbb{R}_+$ between categorical components:

$$d_f(\boldsymbol{x}^{\text{cat}}, \boldsymbol{y}^{\text{cat}}) = \left(\sum_{i \in I^{\text{cat}}} d_i \left(x_i^{\text{cat}}, y_i^{\text{cat}}\right)^2 \right)^{1/2}. \tag{21}$$

A similar construction is used for the constraint functions. The difference is that the observed constraint values are first transformed. For each constraint g_j, define the constraint violation function $h_j : \mathcal{X} \to \mathbb{R}_+$ by

$$h_j(\boldsymbol{x}) = \max\{0, g_j(\boldsymbol{x})\}. \tag{22}$$

When $g_j(\boldsymbol{x}) \leq 0$, the constraint is satisfied and $h_j(\boldsymbol{x}) = 0$. Otherwise, $h_j(\boldsymbol{x}) = g_j(\boldsymbol{x})$ measures its violation. The empirical Wasserstein distances are then constructed from the observed values of h_j, rather than directly from those of g_j. For each constraint g_j and each categorical variable x_i^{cat}, this defines a distance $d_{j,i}$ between the categories of x_i^{cat}. These distances are then aggregated across the categorical variables and the constraints:

$$d_{\boldsymbol{g}}(\boldsymbol{x}^{\text{cat}}, \boldsymbol{y}^{\text{cat}}) = \left(\sum_{j \in J} \sum_{i \in I^{\text{cat}}} d_{j,i} \left(x_i^{\text{cat}}, y_i^{\text{cat}}\right)^2 \right)^{1/2}. \tag{23}$$

The definitions above are stated in terms of the set of known points $\mathbb{X}_{(k)}$. In NomadBBO, the empirical Wasserstein distances are constructed once using the points from the initial DoE and remain fixed throughout the optimization. This avoids progressively unbalanced samples between categories as the optimization converges. Indeed, some categorical components may be evaluated much more often than others, with many points concentrated in a small region and therefore having similar objective values. Using the initial DoE instead provides a more uniformly distributed set of observations that is not influenced by the points selected during the optimization. The initial DoE must therefore provide at least one valid observation for each category of every categorical variable.

2.5 Categorical poll

At iteration k, $\boldsymbol{x}_{(k)}^{\text{cat}}$ denotes the categorical component of the incumbent. For constrained problems, the objective distance d_f and constraint pseudo-distance $d_{\boldsymbol{g}}$ are used to rank the categorical components distinct from $\boldsymbol{x}_{(k)}^{\text{cat}}$. For unconstrained problems, the ranking only uses d_f. The ranking determines which components are considered the most promising neighbors of $\boldsymbol{x}_{(k)}^{\text{cat}}$. The ranking rules depend on the feasibility of the incumbent associated with the categorical component $\boldsymbol{x}_{(k)}^{\text{cat}}$. Let $p_{(k)}$ and $s_{(k)}$ denote the primary and secondary ranking functions, respectively. If $\boldsymbol{x}_{(k)}$ is feasible, they are defined by

$$p_{(k)}(\cdot) = d_{\boldsymbol{g}}(\boldsymbol{x}_{(k)}^{\text{cat}}, \cdot), \qquad s_{(k)}(\cdot) = d_f(\boldsymbol{x}_{(k)}^{\text{cat}}, \cdot). \tag{24}$$

The constraint distance is prioritized to favor components with similar constraint behavior and help maintain feasibility. If $\boldsymbol{x}_{(k)}$ is infeasible, their roles are reversed:

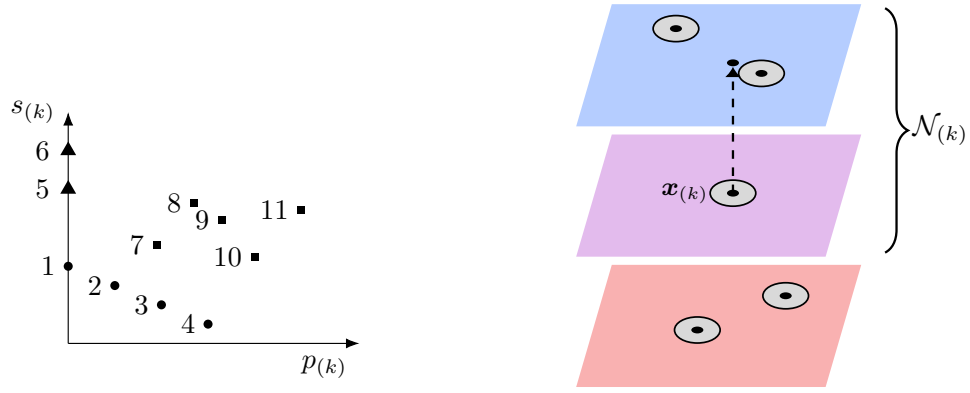
$$p_{(k)}(\cdot) = d_f(\boldsymbol{x}_{(k)}^{\text{cat}}, \cdot), \qquad s_{(k)}(\cdot) = d_{\boldsymbol{g}}(\boldsymbol{x}_{(k)}^{\text{cat}}, \cdot). \tag{25}$$

In this case, the categorical poll is more flexible with respect to feasibility, so the objective distance is prioritized.

Since the categorical domain is finite, all candidate categorical components can be compared at each iteration. For each candidate $\boldsymbol{y}^{\text{cat}} \neq \boldsymbol{x}_{(k)}^{\text{cat}}$, the values $p_{(k)}(\boldsymbol{y}^{\text{cat}})$ and $s_{(k)}(\boldsymbol{y}^{\text{cat}})$ are computed, which places the component in the $(p_{(k)}, s_{(k)})$ space. The candidate components are then ordered according to their positions in this space using three successive rules:

1. The components on the first Pareto front are ranked first, by increasing $p_{(k)}$ and then $s_{(k)}$.
2. Among the remaining components, those with $p_{(k)} = 0$ are ranked next, by increasing $s_{(k)}$.
3. The remaining components are ranked by increasing $p_{(k)}$ and then $s_{(k)}$.

Figure 4a illustrates these three steps for eleven candidate categorical components. The circles correspond to the first Pareto front and are ranked from 1 to 4. The triangles correspond to the remaining components with $p_{(k)} = 0$ and are ranked 5 and 6 by increasing $s_{(k)}$. Finally, the squares correspond to the remaining components and are ranked from 7 to 11 by increasing $p_{(k)}$ and then $s_{(k)}$. The numbers indicate the final ordering. For unconstrained problems, the components are simply ranked by increasing $d_f(\boldsymbol{x}^{\text{cat}}_{(k)}, \boldsymbol{y}^{\text{cat}})$.



(a) Three-step ranking of the categorical components (adapted from [7]).

(b) Categorical poll.

Figure 4: Illustration of the categorical ranking and poll. Panel (a) illustrates the ranking of candidate categorical components. Panel (b) illustrates a categorical poll in $\mathcal{X} = \{\text{Blue}, \text{Purple}, \text{Red}\} \times \mathbb{R}^2$.

The parameter $m_{(k)} \in \{1, 2, \ldots, |\mathcal{X}^{\text{cat}}|\}$ specifies the number of categorical components in the neighborhood. The neighborhood $\mathcal{N}_{(k)}$ contains the incumbent categorical component $\boldsymbol{x}^{\text{cat}}_{(k)}$ and the first $m_{(k)} - 1$ components in the ranking. Figure 4b illustrates an example with $m_{(k)} = 2$. The incumbent categorical component is Purple, while the highest-ranked remaining component is Blue. The neighborhood $\mathcal{N}_{(k)}$ therefore contains these two categorical components. The categorical poll fixes the quantitative variables at their incumbent values and evaluates the neighboring Blue categorical component.

2.6 Extended poll

The optional *extended poll* is performed when both the search and poll steps are unsuccessful. It performs additional quantitative polls around points from the categorical poll that almost, but did not, improve the corresponding incumbent. Since these additional polls can be computationally expensive, only sufficiently promising categorical poll points are selected. The *extended-poll trigger parameter* $\xi \in \overline{\mathbb{R}}$ controls the accepted relative deterioration in objective value. The *extended-poll trigger* is then defined at the incumbent $\boldsymbol{x}_{(k)}$ as

$$t_\xi(\boldsymbol{x}_{(k)}, \boldsymbol{y}) = \begin{cases} 1, & \text{if } 0 \leq f(\boldsymbol{y}) - f(\boldsymbol{x}_{(k)}) \leq \xi |f(\boldsymbol{x}_{(k)})|, \\ 0, & \text{otherwise.} \end{cases} \tag{26}$$

Under the progressive barrier, a feasible point is compared with the feasible incumbent, while an infeasible point is compared with the infeasible incumbent and must also satisfy the current barrier. The selected points are therefore [6]

$$\mathcal{E}_{(k)} = \left\{ \boldsymbol{y} \in P^{\text{cat}}_{(k)} : \left(t_\xi(\boldsymbol{x}_{\text{FEA}}, \boldsymbol{y}) = 1 \text{ and } h(\boldsymbol{y}) = 0\right) \text{ or } \left(t_\xi(\boldsymbol{x}_{\text{INF}}, \boldsymbol{y}) = 1 \text{ and } 0 < h(\boldsymbol{y}) \leq h^{\max}_{(k)}\right) \right\}. \tag{27}$$

For each $\boldsymbol{y} \in \mathcal{E}_{(k)}$, a sequence of quantitative polls is initiated around $\boldsymbol{y}$. As long as a poll produces an improving point, the next quantitative poll is performed around that point. The sequence terminates when a new incumbent is found or when a quantitative poll fails to produce an improvement. The poll sizes are kept fixed throughout the extended poll. However, each new evaluation is added to the set of known points, so the exclusion regions and the punctured space are updated throughout the sequence.

3 Software architecture

NomadBBO separates the optimization workflow into Python and C++ components that are connected via Cython. Figure 5 illustrates the interactions between these components throughout the optimization workflow.

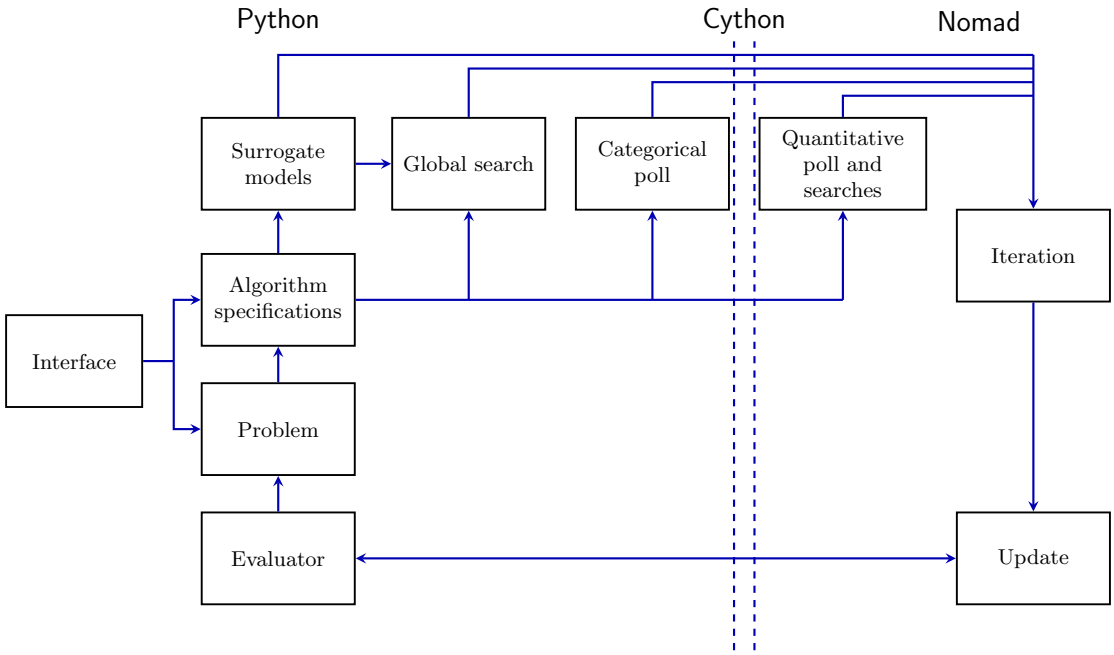


Figure 5: Software architecture of NomadBBO**.**

The Python layer provides the user interface for defining the problem and configuring the optimization algorithm. The problem definition includes the objective and constraint functions provided by the user through an evaluator. As illustrated in Figure 5, the algorithm specifications determine the Python optimization components used during the run, including the surrogate models, global search, and categorical poll. They also control the Nomad algorithmic options, including the quantitative poll and searches. The problem definition and algorithm specifications are passed to Nomad via the Cython integration layer. The surrogate models can support both the global search and the ordering of trial points within the Nomad iteration.

The Python and C++ components communicate via Cython throughout the optimization. Trial points generated by the Python components are passed to the Nomad iteration, where they are combined with the quantitative trial points. Nomad coordinates the evaluation of trial points with the Python evaluator. The objective and constraint values are returned to Nomad, which performs constraint handling, incumbent updates, and the remaining algorithmic updates. The updated incumbents, algorithmic parameters, and newly available data then become available to the optimization mechanisms through the connections illustrated in Figure 5. For example, the update node is connected to the surrogate-model node through the evaluator, problem, and algorithm-specification nodes. The surrogate

models can therefore be updated from the available data, while the categorical and quantitative polls use the updated incumbents and algorithmic parameters provided by the update.

The following subsections describe the Python interface, the Python optimization components, the Cython integration, and the Nomad backend. The final subsection summarizes their interactions within the NomadBBO algorithm. Their organization follows the left-to-right workflow illustrated in Figure 5.

3.1 Python interface

The Python interface includes two elements specified by the user: the problem definition and the optimization algorithm configuration. A problem is defined through the `ProblemDefinition` class, which specifies the variables, evaluation function, blackbox outputs, and optional initial points. Variables are created using the `Real`, `Integer`, `Binary`, and `Choice` classes for continuous, integer, binary, and categorical variables, respectively. The evaluation functions are connected to NomadBBO through the `evaluate` method. This method receives a mixed-variable point, evaluates the user-provided function, and returns the objective and inequality constraint values in the format expected by NomadBBO.

Listing 1 illustrates the initialization of a problem with one categorical, one integer, and one continuous variable, two inequality constraints, and two initial points. Initial points are optional and

```
class MyProblem(ProblemDefinition):
def __init__(self, X0s=None):
variables = {
    material: Choice(options=[Steel, Aluminum, Titanium]),
    x_int:    Integer(bounds=(0, 10)),
    x_cont:   Real(bounds=(0, 5)),
}

super().__init__(
vars=variables,
bbot=[OBJ, PB, PB],
X0s=X0s,
)

def evaluate(self, X):
f, g1, g2 = simulator(X[material], X[x_int], X[x_cont])
return {OBJ: f, PB: [g1, g2]}

# Initialize the problem with two initial points
problem = MyProblem(X0s=[[Steel, 5, 2.5], [Aluminum, 3, 4.0]])
```

Listing 1: Example of a problem definition in NomadBBO.

can be combined with an automatically generated DoE. When additional points are required, the DoE completes the initial sample.

The algorithm is configured through either the `MixedOptimizer` or `QuantitativeOptimizer` class. `MixedOptimizer` is used for problems containing categorical variables. `QuantitativeOptimizer` is used for problems containing only continuous, integer, and/or binary variables. The optimizer must be compatible with the initialized problem. A default configuration can be created without specifying any parameter, while individual algorithmic components can be modified when needed. The problem and optimizer are then passed to the `optimize` function, as illustrated in Listing 2.

The main user-configurable parameters of the optimizers are summarized in Table 2. The first group is shared by `QuantitativeOptimizer` and `MixedOptimizer`, while the second contains the additional parameters of `MixedOptimizer`.

```
# Initialize the optimizer with default settings
optimizer = MixedOptimizer()

# Run the optimization with the problem above
result = optimize(problem, optimizer)
```

Listing 2: Example of an optimizer configuration and optimization call in NomadBBO**.**

Table 2: Main configuration parameters of the NomadBBO **optimizers.**

Parameter	Role	Default	Values
Common optimization parameters			
`seed`	Random seed	0	Nonnegative integer
`initialDOEgenerator`	Initial DoE	Latin hypercube	DoE generator
`quantPollType`	Type of quantitative poll	`ADS`	`ADS`, `MADS`
`isBOSearchUsed`	Bayesian Search	`False`	`True`, `False`
`modelUpdateHardCap`	Maximum surrogate updates	500	Positive integer
`efiStopTol`	BO search stopping tolerance	10^{-3}	Positive real
`efiStopPatience`	BO search stopping patience	3	Positive integer
Additional parameters of `MixedOptimizer`			
`catPollType`	Type of categorical poll	`QUANTILE`	`QUANTILE`, `SURROGATE`, `EXHAUSTIVE`
`nbOfNeighbors`	Categorical poll size	Automatic	$\{1, 2, \ldots, L^{\text{cat}} - 1\}$
`catKernelType`	Categorical kernel	`GOWER`	`GOWER`, `HOMO_HSPHERE`, `CONT_RELAX`
`extendedPollTrigger`	Extended-poll trigger ξ	0.05	$\overline{\mathbb{R}}$
`updateModel`	Surrogate-model updates	`False`	`True`, `False`

The default configuration uses ADS for the quantitative poll and a Latin hypercube for the initial DoE. Bayesian optimization is disabled by default. When enabled, surrogate models are repeatedly updated and the BO search is stopped when its maximum expected feasible improvement becomes smaller than a fraction `efiStopTol` of the best value previously obtained for `efiStopPatience` consecutive iterations. The number of surrogate-model updates is also capped by `modelUpdateHardCap`, since repeatedly fitting the models becomes increasingly expensive as the number of evaluated points grows.

For mixed-variable problems, additional parameters control the categorical components of the algorithm. The categorical poll can use the Wasserstein-based `QUANTILE` strategy, the GP-based `SURROGATE` strategy, or an exhaustive enumeration. `QUANTILE` is used by default. The categorical poll size can be specified manually. Otherwise, it is initialized automatically from the number of possible categorical components using the rule $\min\left\{L^{\text{cat}} - 1, \max\left\{2, \left\lfloor\sqrt{L^{\text{cat}}}\right\rfloor\right\}\right\}$. The formula is adapted from [6] and it ensures that 1) the categorical poll contains at least two points to help avoid cycling between the same categorical components; 2) its size does not exceed the number of components distinct from the incumbent; and 3) its size generally scales proportionally to $\sqrt{L^{\text{cat}}}$. The extended poll is used by default with a trigger parameter $\xi = 0.05$, allowing categorical poll points with a sufficiently small objective deterioration to initiate additional quantitative polls. When a GP is required, the categorical kernel is selected among the mixed-variable kernels available in SMT 2.0. The `CONT_RELAX` kernel uses a binary encoding with continuous relaxation, `HOMO_HSPHERE` uses a homoscedastic hypersphere formulation, and `GOWER` uses the Gower distance. The latter is used by default as the least computationally expensive option. For the surrogate-based categorical poll, `updateModel` determines whether the surrogate model is refitted as new evaluations are collected. If Bayesian optimization is disabled and `updateModel=False`, the surrogate model and the associated categorical distances are constructed once after the initial DoE and then kept fixed. Bayesian optimization instead requires repeated model updates and therefore enables them automatically.

For further details on the Python interface and optimizer configuration, please refer to the NomadBBO TestPyPI page.

3.2 Python optimization components

Several optimization components of NomadBBO are implemented directly in Python. These components operate on the evaluated points and optimization information returned by Nomad throughout the optimization. The corresponding objective and constraint values, together with the variable types and other problem information, are stored and made available to the Python components.

The categorical poll is implemented in Python. The `SURROGATE` strategy uses mixed-variable Gaussian processes from SMT 2.0 [24] to construct the GP-based categorical distances described in Section 2.4.1. The `QUANTILE` strategy instead computes empirical Wasserstein distances using `scipy.stats.wasserstein_distance` [26]. The ranking of categorical components, neighborhood construction, and generation of categorical trial points are all performed in Python. The generated points are passed to Nomad for blackbox evaluation.

The surrogate models can also be used to order trial points before blackbox evaluation. For unconstrained problems, points with lower predicted objective values are prioritized, while for constrained problems the ordering also accounts for predicted feasibility.

Finally, the Global Search component shown in Figure 5 can use Python optimization methods to generate additional trial points. This enables hybridization with optimization methods from other Python libraries. The current implementation provides a Bayesian optimization Search using the mixed-variable GP models from SMT 2.0. For this Search, a GP is fitted to the objective function f and each constraint function g_j. For the objective function, the *Expected Improvement* $\mathrm{EI}(\boldsymbol{x})$ favors points that are predicted to improve the current best objective value and/or have high predictive uncertainty [19]. Points with high uncertainty are typically located in less explored regions of the domain. For each constraint, the *Probability of Feasibility* $\mathrm{PoF}_j(\boldsymbol{x})$ measures how likely the inequality $g_j(\boldsymbol{x}) \leq 0$ is to be satisfied according to the corresponding GP. The *Expected Feasible Improvement* (EFI) weights the EI by the probability that all constraints are satisfied, assuming independent GPs for the constraints [25], and is defined as

$$\mathrm{EFI}(\boldsymbol{x}) = \mathrm{EI}(\boldsymbol{x}) \prod_{j \in J} \mathrm{PoF}_j(\boldsymbol{x}). \tag{28}$$

Maximizing the EFI selects a trial point that is promising for improving the objective, may lie in a less explored region, and is likely to be feasible. For numerical stability, NomadBBO instead maximizes the logarithm of the EFI. This defines an unconstrained mixed-variable acquisition subproblem that is solved with the PyMOO mixed-variable genetic algorithm. Its solution defines a trial point that is passed to Nomad for blackbox evaluation. The Bayesian Search is available to both `QuantitativeOptimizer` and `MixedOptimizer`. As a global surrogate-based Search, it complements the local direct-search mechanism of CatADS.

3.3 Cython integration

The Python and C++ components of NomadBBO are connected through a Cython interface. This layer does more than convert data between the two languages. The main `optimize` function is implemented at the Cython level and coordinates the connection between the Python problem definition, the optimizer, and the Nomad backend. It prepares the initial DoE before converting the problem information and optimization parameters into the format expected by Nomad.

Once an optimization run starts, communication with the Python components is handled through callbacks registered with Nomad. The callbacks allow Python components to be inserted directly into the corresponding steps of the Nomad optimization workflow. The main callbacks are:

- **Function evaluations**. Sends trial points from Nomad to the Python problem definition for evaluation, then provides the resulting objective and constraint values to Nomad;
- **Data management**. Sends evaluated points to the Python side to update the stored data and, when required, the surrogate models;
- **Ordering of trial points**. Sends trial points to Python and returns surrogate predictions to Nomad for ordering;
- **Categorical poll**. Generates categorical trial points on the Python side and returns them to Nomad;
- **Global search**. Generates additional trial points on the Python side, for example using Bayesian optimization, and returns them to Nomad.

The Cython interface also handles data conversion between Python and Nomad. Python components mainly manipulate NumPy arrays and Python objects, while Nomad uses its own C++ data structures. Cython provides the functions required to translate data between these representations during the optimization. These conversions are used whenever information is exchanged through the callbacks described above. At termination, the solutions and optimization statistics returned by Nomad are converted back to a Python representation for the user.

3.4 NOMAD backend

The Nomad backend executes the main optimization loop of NomadBBO. The standard Nomad framework has been extended specifically for NomadBBO to integrate its additional Python-side components. In particular, new callbacks were added to Nomad at specific algorithmic steps to communicate with the Python layer, as described in Section 3.3. The trial points returned by these callbacks are then integrated into the usual Nomad workflow. The backend coordinates the evaluation of trial points, the progressive-barrier constraint handling, the feasible and infeasible incumbents, and the outcome of each iteration. It also performs the corresponding algorithmic updates, including the poll and exclusion sizes for quantitative variables, and checks the stopping criteria.

The usual Nomad mechanisms for continuous and integer variables are also performed on the C++ side. The quantitative poll uses the Householder-based direction generation implemented in Nomad. NomadBBO uses the `ORTHO 2N` option, where each generated direction is paired with its opposite. The speculative search [3, 20] is used after a successful quantitative poll and generates additional trial points along the last successful direction. The quadratic search of Nomad [15] is also used on the quantitative variables. At each iteration, quadratic models of the objective and constraint functions are constructed around the incumbents, while the categorical variables remain fixed. For a feasible incumbent, the model subproblem seeks a point minimizing the objective model while satisfying the modeled constraints. For an infeasible incumbent, it minimizes the objective model while remaining below the current barrier. For more details on the speculative and quadratic searches, see [10].

3.5 NomadBBO algorithm

Algorithm 2 presents a summarized view of the NomadBBO algorithm, focusing on its main optimization steps and their implementation across the different software layers. The labels on the right indicate whether each operation is primarily performed in Python, at the Cython integration level, or in the Nomad backend. The evaluation of trial points is performed opportunistically. As soon as a successful trial point is found: 1) the remaining trial points in the current step are skipped; 2) the subsequent steps generating trial points are skipped; and 3) the algorithm proceeds directly to the update.

4 Numerical experiments

The numerical experiments focus on mixed-variable optimization. This setting is emphasized because handling categorical variables is the main optimization capability introduced by NomadBBO beyond the

Algorithm 2: NomadBBO **algorithm.**

0. Initialization
- Define the problem with `ProblemDefinition` **[Python]**
- Configure the optimization algorithm **[Python]**
- Prepare the initial DoE **[Cython]**
- Configure Nomad **[Cython]**
- Register the Python callbacks **[Cython]**
- Evaluate the initial points **[Python]**
- Initialize the barrier and incumbents **[NOMAD]**

While *no stopping criterion is met* **do**

1. Search
- Speculative search, if triggered **[NOMAD]**
- Quadratic search **[NOMAD]**
- Global search, if enabled **[Python]**
- Compute surrogate predictions, if applicable **[Python]**
- Order search points **[NOMAD]**
- Evaluate search points opportunistically **[Python]**

2. Poll
- Generate quantitative poll points **[NOMAD]**
- Generate categorical poll points, if applicable **[Python]**
- Compute surrogate predictions, if applicable **[Python]**
- Order poll points **[NOMAD]**
- Evaluate poll points opportunistically **[Python]**

3. Extended poll

If the extended poll is triggered
- Generate extended poll points **[NOMAD]**
- Compute surrogate predictions, if applicable **[Python]**
- Order extended poll points **[NOMAD]**
- Evaluate extended poll points **[Python]**

4. Update
- Update stored evaluation data **[Python]**
- Update surrogate models, if applicable **[Python]**
- Update the barrier and incumbents **[NOMAD]**
- Update poll and exclusion sizes **[NOMAD]**

5. Termination
- Check stopping criteria **[NOMAD]**

end

Return the optimization results **[Cython]**

standard capabilities of Nomad. The experiments use problems from the Cat-Suite collection [18] and compare NomadBBO with existing mixed-variable optimization solvers. Unconstrained and constrained problems are analyzed separately. Additional experiments, including tests of the Bayesian search on quantitative problems, will be added in future versions of this manuscript.

4.1 Experimental setup for data profiles

Data profiles are the principal benchmarking tool used in this work to assess the performance of NomadBBO. They compare the relative performance of several solvers over a collection of problems with respect to a computational budget [22]. For the experiments, an instance is defined as the combination of a given problem and a random seed. Different seeds therefore correspond to different instances of the same problem.

Let $\mathcal{P}$ denote the set of instances considered in a given data profile, and let $\mathcal{S}$ denote the set of solvers in the comparison. The convergence test for an instance $p \in \mathcal{P}$ depends on an initial objective value $f_0 \in \mathbb{R}$. The value of f_0 depends on the experimental setup of the specific data profile. For example, it may be the objective value of an initial point or the best objective value in a common initial DoE [9]. The precise definition of f_0 is specified for each data profile.

A solver $s \in \mathcal{S}$ τ-solves an instance $p \in \mathcal{P}$ if it produces a feasible solution $\boldsymbol{x}_{\text{FEA}} \in \Omega$ whose objective reduction is within a tolerance τ of the best reduction obtained by any solver on that instance [22]. More precisely, the τ-convergence test is defined as

$$f_0 - f(\boldsymbol{x}_{\text{FEA}}) \geq (1-\tau)(f_0 - f_\star), \tag{29}$$

where $f_\star$ is the best feasible objective value obtained by any solver on the instance and $\tau \in [0,1]$ is a given tolerance.

For an instance $p \in \mathcal{P}$, let $n_p = n_p^{\text{cat}} + n_p^{\text{bin}} + n_p^{\text{int}} + n_p^{\text{cont}}$ denote its total number of variables.

A data profile represents the fraction of instances τ-solved by a solver $s \in \mathcal{S}$ as a function of the computational budget [22]. It is defined as

$$\text{data}_s(\kappa) := \frac{1}{|\mathcal{P}|} \left| \left\{ p \in \mathcal{P} : \frac{k_{p,s}}{n_p + 1} \leq \kappa \right\} \right| \in [0,1], \tag{30}$$

where $k_{p,s} \geq 0$ is the number of evaluations required by solver s to τ-solve instance p. Simply put, κ represents a computational budget measured in multiples of $(n_p + 1)$ evaluations.

For all data profiles in this work, the maximum evaluation budget for an instance p is set to $250 n_p$.

4.2 Data profiles for selecting the NomadBBO configuration

Two configurations of NomadBBO are compared to determine which one is used in the subsequent benchmarks. The first uses the default NomadBBO configuration. The second configuration, denoted NomadBBO$_{\text{BO}}$, enables the Bayesian search and uses the SURROGATE strategy for the categorical poll. For this configuration, the GOWER categorical kernel is used. All other parameters for Bayesian optimization and surrogate-model updates remain at their default values presented in Table 2. The CatMADS prototype provides a direct baseline for assessing the improvements introduced by CatADS over its predecessor. It is used as presented in [6].

The comparison is performed on the 30 unconstrained and 30 constrained mixed-variable problems from the Cat-Suite collection [18]. Each problem is instantiated with three different random seeds, resulting in a total of 180 instances. The initial DoE consists of 20% of the evaluation budget and is common to all three implementations.

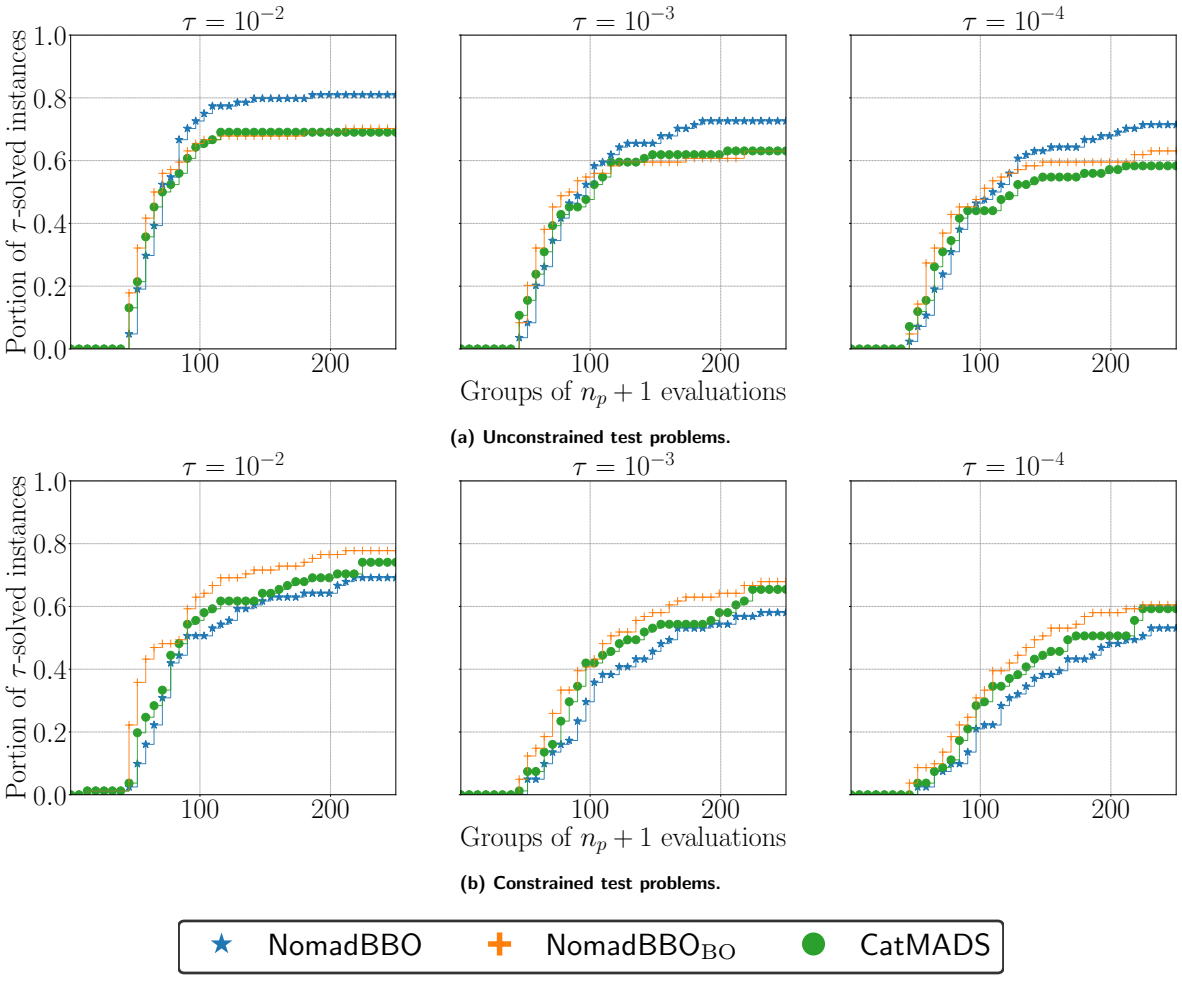
For these data profiles, the reference value f_0 is chosen as

- the least objective function value in the common initial DoE, for unconstrained problems [9];
- the smallest objective function value among the first feasible solutions found by the three implementations, for constrained problems [9]. This strategy allows the implementations to be compared even when the initial DoE of an instance does not contain a feasible solution.

A data profile is better when its curve is higher and rises more rapidly. For a fixed budget κ, the vertical value indicates the fraction of instances that are τ-solved. For example, in Figure 6a, with $\tau = 10^{-2}$ and $\kappa = 100$, NomadBBO τ-solves approximately 75% of the instances, compared with approximately 65%–70% for NomadBBO$_{\text{BO}}$ and the CatMADS prototype.

The unconstrained data profiles in Figure 6a show that the default NomadBBO configuration outperforms the two surrogate-based methods for the three considered tolerances. For the constrained problems in Figure 6b, the differences are smaller. NomadBBO$_{\text{BO}}$ generally performs best. The CatMADS prototype follows closely, while the default NomadBBO configuration generally performs the worst. However, the differences remain relatively limited compared with those observed on the unconstrained problems.

An important difference between these implementations is their computational cost. Both NomadBBO$_{\text{BO}}$ and the CatMADS prototype use surrogate models to construct their categorical polls.



(a) Unconstrained test problems.

(b) Constrained test problems.

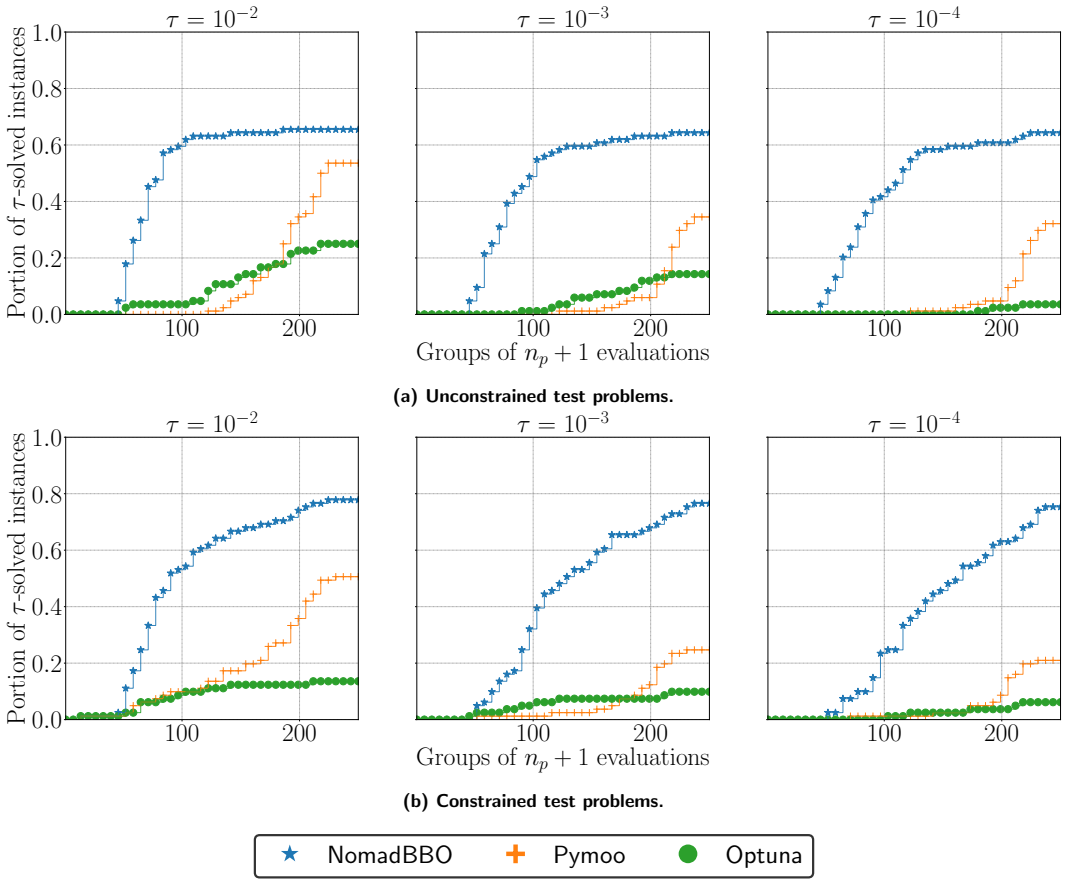
Figure 6: Data profiles comparing the NomadBBO configurations with the CatMADS prototype.

In contrast, NomadBBO uses much less costly Wasserstein distances. On an 11th-generation Intel i7-11800H CPU at 2.30 GHz, the most expensive constrained instances considered here can require approximately two hours with NomadBBO$_{\text{BO}}$ and one hour with the CatMADS prototype. The longest runs with the default NomadBBO configuration require at most a few minutes. The default NomadBBO configuration performs substantially better on the unconstrained problems and remains competitive on the constrained problems. It is also considerably faster. For these reasons, it is selected for the comparisons with external solvers in the next subsection.

4.3 Data profiles against general-purpose mixed-variable solvers

The selected NomadBBO configuration is benchmarked against Optuna and PyMOO, two accessible general-purpose solvers for mixed-variable blackbox optimization discussed in Section 1.3. Both provide computationally efficient methods that are well suited to comparisons over many problems and large evaluation budgets. The same Cat-Suite instances, evaluation budgets, initial DoE, and reference values f_0 as in the previous subsection are used. For Optuna and PyMOO, only the evaluation budget is specified, and both solvers start from the same initial DoE as NomadBBO. Standalone BO solvers are not included in this benchmark because updating a GP requires operations on a covariance matrix constructed from the available data. These operations scale as $\mathcal{O}(N^3)$ with the number of observations N and become computationally expensive for the large evaluation budgets considered here.

The data profiles comparing NomadBBO with the solvers on the unconstrained and constrained instances are presented in Figures 7a and 7b.



(a) Unconstrained test problems.

(b) Constrained test problems.

Figure 7: Data profiles comparing NomadBBO with the general-purpose mixed-variable solvers Optuna and PyMOO.

The data profiles in Figure 7 show that NomadBBO clearly outperforms both PyMOO and Optuna for all three tolerances, with an even larger advantage on the constrained problems.

5 Discussion

This work introduces two complementary contributions. The first is CatADS, a direct-search method that extends ADS to mixed-variable blackbox optimization by combining its quantitative poll with a categorical poll based on problem-dependent neighborhoods. It also introduces a new categorical-poll strategy based on empirical Wasserstein distances as a computationally inexpensive alternative to surrogate-based neighborhoods [7]. The second and main contribution is NomadBBO, an accessible and modular Python library that integrates these optimization mechanisms with the efficient C++ backend of Nomad. The library combines mixed-variable optimization, constraint handling, theoretical guarantees, and the integration of external optimization methods within a single framework.

The numerical results are promising. On the mixed-variable problems from Cat-Suite, the default NomadBBO configuration clearly outperforms the general-purpose solvers PyMOO and Optuna for both unconstrained and constrained problems. The comparison between the different NomadBBO configurations also highlights an important computational tradeoff. Surrogate-based categorical polls can improve performance on constrained problems, but are significantly more expensive computationally.

The Wasserstein-based strategy is much cheaper while remaining competitive. This tradeoff motivates the use of the default NomadBBO configuration.

NomadBBO is under active development and currently focuses on single-objective problems with inequality constraints. Planned extensions include equality-constraint handling and multiobjective optimization, which are already supported to different extents by general-purpose libraries such as PyMOO and Optuna. Support for parallel blackbox evaluations is also planned. These developments will broaden the range of problems that can be addressed by NomadBBO. Further numerical studies will consider other problem classes, including quantitative problems, as well as different variants of NomadBBO.

NomadBBO is currently distributed as a beta package through TestPyPI. A stable release on PyPI is planned after broader benchmarking and further user testing help identify potential issues.

Conflict of interest statement

The authors state that there are no conflicts of interest.

Statement on the use of IA tools

The authors used ChatGPT to assist with generating the figures, listings and tables. It was also used to improve English formulations and organize the software section. ChatGPT was not used to generate numerical results or scientific contributions. The authors remain fully responsible for the scientific content of this work.

References

[1] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, 2019.

[2] S. Alarie, C. Audet, A.E. Gheribi, M. Kokkolaras, and S. Le Digabel. Two decades of blackbox optimization applications. EURO Journal on Computational Optimization, 9:100011, 2021.

[3] C. Audet and J.E. Dennis, Jr. Mesh Adaptive Direct Search Algorithms for Constrained Optimization. SIAM Journal on Optimization, 17(1):188–217, 2006.

[4] C. Audet and J.E. Dennis, Jr. A Progressive Barrier for Derivative-Free Nonlinear Programming. SIAM Journal on Optimization, 20(1):445–472, 2009.

[5] C. Audet, T. Denorme, Y. Diouane, S. Le Digabel, and C. Tribes. Adaptive direct search algorithms for constrained optimization. Technical Report G-2025-53, Les cahiers du GERAD, 2025.

[6] C. Audet, Y. Diouane, E. Hallé-Hannan, S. Le Digabel, and C. Tribes. CatMADS: Mesh Adaptive Direct Search for constrained blackbox optimization with categorical variables. Technical Report G-2025-42, Les cahiers du GERAD, 2026. To appear in SIAM Journal on Optimization.

[7] C. Audet, Y. Diouane, E. Hallé-Hannan, S. Le Digabel, and C. Tribes. Surrogate-based categorical neighborhoods for mixed-variable blackbox optimization. Technical Report G-2026-16, Les cahiers du GERAD, 2026.

[8] C. Audet and W. Hare. Derivative-Free and Blackbox Optimization. Springer Series in Operations Research and Financial Engineering. Springer, Cham, Switzerland, 1st edition, 2017.

[9] C. Audet, W. Hare, and C. Tribes. A summary of benchmarking constrained, multi-objective and surrogate-assisted optimization methods. Optimization Letters, 2026.

[10] C. Audet, S. Le Digabel, V. Rochon Montplaisir, and C. Tribes. Algorithm 1027: NOMAD version 4: Nonlinear optimization with the MADS algorithm. ACM Transactions on Mathematical Software, 48(3):35:1–35:22, 2022.

[11] C. Audet, S. Le Digabel, and C. Tribes. Dynamic scaling in the mesh adaptive direct search algorithm for blackbox optimization. Optimization and Engineering, 17(2):333–358, 2016.

[12] C. Audet, S. Le Digabel, and C. Tribes. The Mesh Adaptive Direct Search Algorithm for Granular and Discrete Variables. SIAM Journal on Optimization, 29(2):1164–1189, 2019.

[13] J. Blank and K. Deb. Pymoo: Multi-Objective Optimization in Python. IEEE Access, 8:89497–89509, 2020.

[14] T.D. Choi and C.T. Kelley. Superlinear Convergence and Implicit Filtering. SIAM Journal on Optimization, 10(4):1149–1162, 2000.

[15] A.R. Conn and S. Le Digabel. Use of quadratic models with mesh-adaptive direct search for constrained black box optimization. Optimization Methods and Software, 28(1):139–158, 2013.

[16] E. Hallé-Hannan. Cadre mathématique pour l'optimisation de boîtes noires avec variables métas et catégorielles. Master's thesis, Polytechnique Montréal, 2022. Available at https://publications.polymtl.ca/10286/.

[17] E. Hallé-Hannan. Contributions to Constrained Mixed-Variable and Hierarchical Blackbox Optimization. PhD thesis, Polytechnique Montréal, 2026. Available at https://publications.polymtl.ca/76526/1/2026_EdwardHalleHannan.pdf.

[18] E. Hallé-Hannan, C. Audet, Y. Diouane, S. Le Digabel, and C. Tribes. Cat-Suite: A collection of optimization problems with categorical and quantitative variables for benchmarking. Technical Report G-2025-39, Les cahiers du GERAD, 2025.

[19] D.R. Jones, M. Schonlau, and W.J. Welch. Efficient Global Optimization of Expensive Black Box Functions. Journal of Global Optimization, 13(4):455–492, 1998.

[20] S. Le Digabel. Algorithm 909: NOMAD: Nonlinear Optimization with the MADS algorithm. ACM Transactions on Mathematical Software, 37(4):44:1–44:15, 2011.

[21] S. Le Digabel and S.M. Wild. A taxonomy of constraints in black-box simulation-based optimization. Optimization and Engineering, 25(2):1125–1143, 2024.

[22] J.J. Moré and S.M. Wild. Benchmarking Derivative-Free Optimization Algorithms. SIAM Journal on Optimization, 20(1):172–191, 2009.

[23] P. Saves. High-dimensional multidisciplinary design optimization for aircraft eco-design. PhD thesis, ONERA and ISAE-SUPAERO, 2024. Available at https://theses.hal.science/ONERA-MIP/tel-04439128v1.

[24] P. Saves, R. Lafage, N. Bartoli, Y. Diouane, J.H. Bussemaker, T. Lefebvre, J.T. Hwang, J. Morlier, and J.R.R.A Martins. SMT 2.0: A Surrogate Modeling Toolbox with a focus on Hierarchical and Mixed Variables Gaussian Processes. Advances in Engineering Software, 188:103571, 2024.

[25] B. Talgorn, S. Le Digabel, and M. Kokkolaras. Statistical Surrogate Formulations for Simulation-Based Design Optimization. Journal of Mechanical Design, 137(2):021405–1–021405–18, 2015.

[26] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. Nature Methods, 17:261–272, 2020.