https://pubsonline.informs.org/journal/ijoc

INFORMS JOURNAL ON COMPUTING
Vol. 37, No. 1, January–February 2025, pp. 107–124
ISSN 1091-9856 (print), ISSN 1526-5528 (online)

QHDOPT: A Software for Nonlinear Optimization with Quantum Hamiltonian Descent

Samuel Kushnir,[a] Jiaqi Leng,[b,c,d,*] Yuxiang Peng,[a,c] Lei Fan,[e,f] Xiaodi Wu[a,c]

[a] Department of Computer Science, University of Maryland, College Park, Maryland 20742; [b] Department of Mathematics, University of Maryland, College Park, Maryland 20742; [c] Joint Center for Quantum Information and Computer Science, University of Maryland, College Park, Maryland 20742; [d] Department of Mathematics and Simons Institute for the Theory of Computing, University of California, Berkeley, California 94720; [e] Department of Engineering Technology, University of Houston, Houston, Texas 77204; [f] Department of Electrical and Computer Engineering, University of Houston, Houston, Texas 77204

*Corresponding author

Contact: samkushnir@gmail.com, https://orcid.org/0009-0007-7595-0819 (SK); jiaqil@terpmail.umd.edu, https://orcid.org/0000-0002-9276-2832 (JL); ypeng15@umd.edu, https://orcid.org/0000-0003-0592-7131 (YP); lfan8@central.uh.edu, https://orcid.org/0000-0003-2157-310X (LF); xwu@cs.umd.edu, https://orcid.org/0000-0001-8877-9802 (XW)

Received: January 17, 2024
Revised: June 30, 2024; September 2, 2024; September 26, 2024
Accepted: September 28, 2024
Published Online in Articles in Advance: November 25, 2024

https://doi.org/10.1287/ijoc.2024.0587

Abstract. We develop an open-source, end-to-end software (named QHDOPT), which can solve nonlinear optimization problems using the quantum Hamiltonian descent (QHD) algorithm. QHDOPT offers an accessible interface and automatically maps tasks to various supported quantum backends (i.e., quantum hardware machines). These features enable users, even those without prior knowledge or experience in quantum computing, to utilize the power of existing quantum devices for nonlinear and nonconvex optimization tasks. In its intermediate compilation layer, QHDOPT employs SimuQ, an efficient interface for Hamiltonian-oriented programming, to facilitate multiple algorithmic specifications and ensure compatible cross-hardware deployment. The detailed documentation of QHDOPT is available at https://github.com/jiaqileng/QHDOPT.

History: Accepted by Giacomo Nannicini, Area Editor for Quantum Computing and Operations Research. Accepted for Special Issue.

Funding: This work was supported by the U.S. Department of Energy's Advanced Research Projects Agency–Energy [Grant DE-SC0020273], the Alfred P. Sloan Foundation, the Simons Foundation [Simons Investigator Award 825053], the Simons Quantum Postdoctoral Fellowship, the National Science Foundation [Grants CCF-1816695, CCF-1942837, and ECCS-2045978], the Unitary Fund, and the Air Force Office of Scientific Research [Grant FA95502110051].

Supplemental Material: The software that supports the findings of this study is available within the paper and its Supplemental Information (https://pubsonline.informs.org/doi/suppl/10.1287/ijoc.2024.0587) as well as from the IJOC GitHub software repository (https://github.com/INFORMSJoC/2024.0587). The complete IJOC Software and Data Repository is available at https://informsjoc.github.io/.

Keywords: Quantum Hamiltonian Descent • nonlinear optimization • quantum optimization • SimuQ

1. Introduction

Nonlinear optimization, also known as nonlinear programming (NLP), is a branch of mathematical optimization concerned with solving problems in which the objective function, constraints, or both exhibit nonlinearity. Although nonlinear optimization problems are common in various application fields such as engineering, management, economics, and finance, these problems are in general nonconvex with complicated landscape features like multiple local stationary points, valleys, and plateaus. As the number of variables grows, the complexity of the problem could grow rapidly, posing a significant challenge in obtaining globally optimal solutions.

Several open-source and commercial software packages, including Ipopt (Kawajir et al. 2006), Gurobi (Gurobi Optimization, LLC 2021), and CPLEX (Bliek1ú et al. 2014), have been developed to tackle large-scale nonlinear optimization problems. Although these optimizers can incorporate powerful heuristics to enhance performance for certain problem instances, there is no polynomial-time guarantee for these optimizers because nonlinear optimization is generally NP-hard. Often, the problem structure is unknown, and there is no commonly agreed-upon *go-to* optimizer for nonlinear optimization in practice.

Quantum computers are emerging technologies that can leverage the laws of quantum mechanics to offer theoretical and practical advantages over classical computers in solving large-scale computational problems. Unlike their classical counterparts, quantum computers utilize a unique phenomenon known as quantum tunneling to accelerate the solution of nonconvex optimization problems. Specifically, a quantum particle can pass through a high potential barrier that would be insurmountable classically because of insufficient energy. This exotic behavior enables a quantum computer to bypass suboptimal solutions, efficiently navigating the complex landscape of nonlinear optimization.

Recently, Leng et al. (2023b) proposed a novel quantum algorithm named quantum Hamiltonian descent (QHD). QHD is inspired by the observation that many first-order (i.e., gradient-based) methods can be interpreted as dynamical processes governed by physical laws. For example, it has been shown that the celebrated Nesterov accelerated gradient descent algorithm can be modeled by a time-dependent Lagrangian mechanical system that would find local minima in the system (Su et al. 2016, Wibisono et al. 2016). By upgrading the classical Lagrangian mechanics to quantum mechanics, we end up with a minimum-finding quantum process, just like gradient descent. Additionally, this quantum dynamical process demonstrates the quantum tunneling effect, making it a competitive candidate for solving nonconvex optimization problems. Simulating this quantum dynamical process on a quantum computer gives rise to QHD, a simple but powerful quantum algorithm for continuous optimization, especially nonlinear problems with nonconvex objective functions. A follow-up work by Leng et al. (2023a) shows that QHD can solve a family of hard optimization instances in polynomial time, whereas an empirical study suggests that these problem instances are intractable for many classical optimization algorithms such as branch-and-bound, stochastic gradient descent, the interior point method, etc.

A key feature of QHD is that it is formulated as a quantum evolution, which can be simulated on both digital and analog quantum computers. This feature allows us to implement QHD to tackle real-world tasks with near-term realizable quantum computers. Digital quantum computers perform computation by applying a sequence of elementary quantum gates to an initial quantum state. These machines exhibit provable quantum advantages over classical (digital) computers for certain computational tasks; however, they require a large number of digital (i.e., error-corrected) qubits. Although there has recently been a groundbreaking experimental demonstration of early fault tolerance (Google Quantum AI 2023, Singh et al. 2023, Sivak et al. 2023, Bluvstein et al. 2024), existing digital quantum computers have not yet reached the size necessary to accelerate the solution of real-world problems in application domains such as management, finance, and engineering (Beverland et al. 2022, Dalzell et al. 2023). Analog quantum computers solve computational tasks by configuring and emulating a real quantum system and then performing quantum measurements. These devices are easier to fabricate, control, and scale (O'Brien et al. 2009, Saffman 2016, Wendin 2017), although they are unavoidably noisy, and no general error correction technique is currently practical (Lloyd and Slotine 1998, Atalaya et al. 2021). Leng et al. (2023b) proposed a systematic technique named *Hamming encoding* that enables us to implement QHD to solve quadratic programming (QP) problems on analog quantum computers with an Ising Hamiltonian. This technique is exemplified in solving 75-dimensional nonconvex QP problems, where the noisy real-machine implementation of QHD outperforms existing open-source nonlinear optimization software like Ipopt.

In this paper, we develop `QHDOPT`, an end-to-end implementation of QHD for nonlinear optimization. A notable feature of `QHDOPT` is that it supports the deployment of QHD to multiple quantum computing hardware, including gate-based quantum computers such as IonQ, and analog quantum computers such as D-Wave. `QHDOPT` provides a user-friendly interface, with which a nonlinear optimization problem can be specified via either matrix/numeric or symbolic description. Then, the implementation of QHD is fully automatized, and the (approximate) optimal solutions will be returned once the computation is completed. The midlevel compilation and cross-hardware deployment are achieved by utilizing `SimuQ`, a framework for programming and compiling quantum Hamiltonian systems, for Hamiltonian-oriented programming (HOP; Peng et al. 2024).

1.1. Organization

The rest of this paper is organized as follows.[1] In Section 1.2, we explain the general problem formulation for nonlinear optimization problems that can be processed and solved by `QHDOPT`. In Section 1.3, we discuss the workflow of `QHDOPT`, including the quantum backend and classical refinement. In Section 1.4, we discuss several unique design features of `QHDOPT`, especially the multibackend compatibility achieved by incorporating the HOP framework. In Section 2, we briefly review the QHD algorithm and its implementation on both digital and analog quantum computers. Then, in Section 3, we sketch the workflow of the software, including all major steps in the implementation of QHD and classical postprocessing. Section 4 provides two worked examples of modeling and solving nonlinear optimization problems. In Section 5, we review the current state and trend of quantum optimization software. We conclude with a comparison of `QHDOPT` with other available open-source optimizers in Section 6.

1.2. Problem Formulation: Box-Constrained Nonlinear Optimization

The package QHDOPT solves nonlinear programming problems of the following form:

$$\min_x f(x_1, \dots, x_n) = \underbrace{\sum_{i=1}^{n} g_i(x_i)}_{\text{univariate part}} + \underbrace{\sum_{j=1}^{m} p_j(x_{k_j}) q_j(x_{\ell_j})}_{\text{bivariate part}}, \tag{1.1a}$$

$$\text{s.t. } L_i \le x_i \le U_i, \ \forall i \in \{1, \dots, n\}, \tag{1.1b}$$

where $x_1, \dots, x_n$ are n variables subject to the box constraint $x_i \in [L_i, U_i] \subset \mathbb{R}$ for each $i = 1, \dots, n$, and the indices $k_j, \ell_j \in \{1, \dots, n\}$ and $k_j \neq \ell_j$ for each $j = 1, \dots, m$. The functions $g_i(x_i)$, $p_j(x_{k_j})$, and $q_j(x_{\ell_j})$ are real univariate differentiable functions defined on $\mathbb{R}$. Note that the univariate part in (1.1a) has at most n terms because we can always combine separate univariate functions of a fixed variable x_i into a single one. However, there is no upper bound for the integer m (i.e., the number of bivariate terms).[2]

The nonlinear optimization problem (1.1) is in general NP-hard (Hochbaum 2007) and can be used to model several common classes of optimization problems, including linear programming, quadratic programming, and polynomial optimization (with box constraints). In the following examples, we show how to formulate some standard nonlinear optimization problems in the form of (1.1a).

Example 1 (Box-Constrained Quadratic Programming). A quadratic programming problem with a box constraint takes the form

$$\min_x f(x) := \frac{1}{2} x^\top Q x + b^\top x \tag{1.2a}$$

$$\text{s.t. } 0 \le x \le 1, \tag{1.2b}$$

where $Q \in \mathbb{R}^{n \times n}$ is a symmetric matrix, and b is a real-valued vector of dimension n. The objective function can be written as

$$f(x) = \sum_{i=1}^{n} \left(\frac{1}{2} Q_{i,i} x_i^2 + b_i x_i \right) + \sum_{1 \le k < \ell \le n} Q_{k,\ell} x_k x_\ell.$$

This function is represented by (1.1a) by choosing

$$g_i(x_i) = \frac{1}{2} Q_{i,i} x_i^2 + b_i x_i, \qquad \forall i = 1, \dots, n, \tag{1.3a}$$

$$p_j(x_{k_j}) = Q_{k_j,\ell_j} x_{k_j}, \quad q_j(x_{\ell_j}) = x_{\ell_j}, \qquad \forall j \in \left\{1, \dots, \frac{n(n-1)}{2}\right\}. \tag{1.3b}$$

Here (k_j, ℓ_j) are the jth pair in the enumeration $\{(k, \ell) : 1 \le k < \ell \le n\}$.

Although the problem formulation can handle only box constraints, we note that many optimization problems with more sophisticated constraints can be reformulated in the form of (1.1) by adding the constraints as a penalty term in the objective function.

Example 2 (Spherical Constraints). Consider the optimization problem with n variables:

$$\min_x f(x) := \sum_{j=1}^{n} \alpha_j x_j \tag{1.4a}$$

$$\text{s.t. } \sum_{j=1}^{n} x_j^2 = 1, \tag{1.4b}$$

where α_j are real scalars for all $j = 1, \dots, n$. The feasible set of this problem is the n-dimensional sphere with radius one, which cannot be directly recast as a box in the form of (1.1b). Meanwhile, we observe that all the variables must take values between zero and one because the unit sphere is contained in the unit (hyper)cube. Therefore, we can reformulate (1.4) to a box-constrained optimization problem by the penalty method:

$$\min_x f(x) := \sum_{j=1}^{n} \alpha_j x_j + \lambda \left(\sum_{j=1}^{n} x_j^2 - 1 \right)^2, \tag{1.5a}$$

$$\text{s.t. } 0 \le x \le 1. \tag{1.5b}$$

This new problem can be handled by our software QHDOPT because the objective function (1.5a) involves only uni- and bivariate monomials. As the penalty coefficient $\lambda > 0$ grows, we can show that the solution to the box-constrained problem (1.5) will eventually converge to the optimal solution to the original problem (1.4).

It is worth noting that the problem formulation supported by QHDOPT is restrictive, and there exist many general nonlinear optimization problems that cannot be directly expressed in (1.1). For example, our formulation cannot deal with objective functions involving trivariate monomials (e.g., xyz). Although, in theory, QHD can handle box-constrained optimization models given access to ideal quantum hardware, in QHDOPT, we limit the appearance of trivariate parts or higher to cater to the current quantum hardware restrictions.

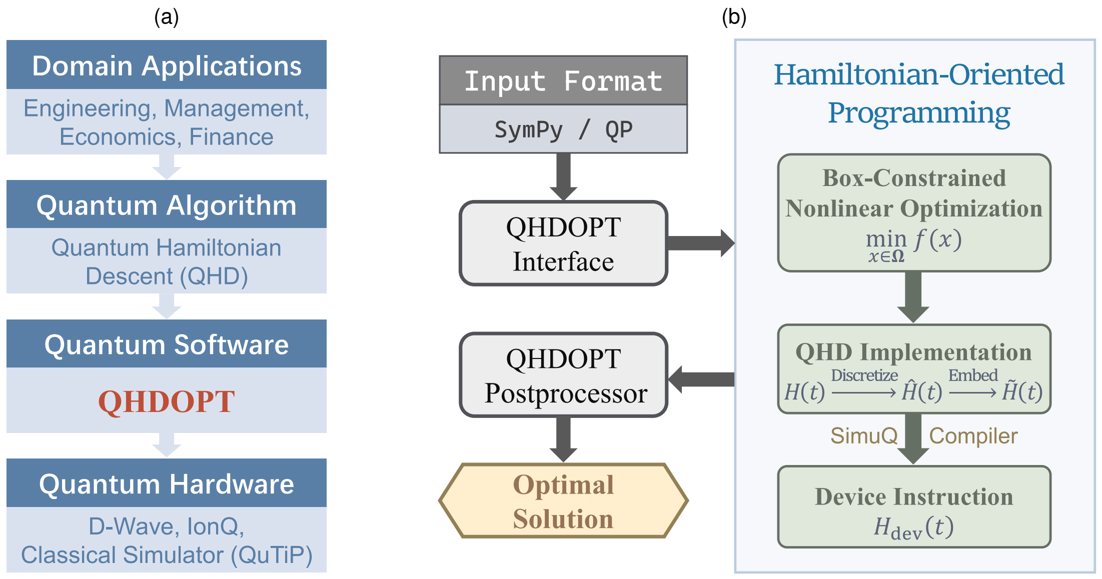
Additionally, we note that there may be several corner cases that are representable by (1.1) but would require an excessively long time for QHDOPT to parse and solve. For example, when the objective function involves thousands of bivariate functions, it might take QHDOPT minutes to compile and implement the automatic differentiation subroutine based on JAX. We advise users to prioritize the use cases with low-degree polynomials, bounded exponential functions, and simple trigonometric functions.

1.3. Solving Problems in QHDOPT

QHDOPT utilizes the quantum Hamiltonian descent algorithm to facilitate the solution of nonlinear and nonconvex optimization problems. Theoretically, quantum Hamiltonian descent, when running with an ideal fault-tolerant quantum computer, can solve many optimization problems up to global optimality given a sufficiently long run time (Leng et al. 2023b). However, at the current stage, because of the lack of fault tolerance, we can implement quantum Hamiltonian descent only in a *low-precision* and *noisy* manner, which significantly reduces the solution quality promised by the theoretical guarantee. To mitigate the noisy performance of near-term quantum hardware with limited resources, we adopt a hybrid quantum-classical computing workflow in QHDOPT to achieve optimal performance, as illustrated in Figure 1(b).

1.3.1. Preprocessing and Problem Encoding. First, we map a box-constrained nonlinear optimization problem to a quantum-mechanical system with finite degrees of freedom. This reduced quantum model can be regarded as a finite-precision approximation of the original QHD model. Then, this quantum model is embedded into a larger quantum system that is natively executable using one of the supported quantum backends. This process is called *Hamiltonian programming*. Although the quantum hardware "sees" only a reduced version of the original problem, the *Hamiltonian embedding* technique (Leng et al. 2024) ensures that the spatial structure inherited from the original problem is preserved and naturally encoded in the quantum operator. Therefore, QHDOPT allows us to run a coarse-grained version of QHD on near-term quantum devices.

Figure 1. (Color online) An Overview of QHDOPT


(a)
Domain Applications
Engineering, Management, Economics, Finance
Quantum Algorithm
Quantum Hamiltonian Descent (QHD)
Quantum Software
QHDOPT
Quantum Hardware
D-Wave, IonQ, Classical Simulator (QuTiP)
(b)
Input Format
SymPy / QP
QHDOPT Interface
QHDOPT Postprocessor
Optimal Solution
Hamiltonian-Oriented Programming
Box-Constrained Nonlinear Optimization
$\min_{x\in\Omega} f(x)$
QHD Implementation
$H(t) \xrightarrow{\text{Discretize}} \hat{H}(t) \xrightarrow{\text{Embed}} \tilde{H}(t)$
SimuQ
Compiler
Device Instruction
$H_{\text{dev}}(t)$

Notes. Panel (a) shows building the stack of quantum computing for nonlinear optimization. Panel (b) shows the workflow of QHDOPT, inspired by the Hamiltonian-oriented programming paradigm.

1.3.2. Deployment and Decoding. Then, the quantum operator that encodes the original nonlinear optimization problem is constructed and executed on a quantum backend. Currently, `QHDOPT` supports three backends: the D-Wave quantum computer, the IonQ quantum computer, and a classical simulator based on QuTiP (a Python-based quantum simulation software). The measurement results from quantum devices are in 0–1 format (i.e., binaries), which requires a decoder to recover the corresponding solution in the continuous space (e.g., the unit box).

1.3.3. Classical Refinement. Limited by the size and coherent time of current quantum devices, the quantum-generated solutions are of low precision and intrinsically noisy. `QHDOPT` relies on classical local search algorithms, such as first- and second-order methods, to improve numerical precision. Currently, `QHDOPT` supports two local optimizers: a general-purpose interior point method (Ipopt) and a truncated Newton method (TNC) implemented in SciPy. Although we do not include other local search subroutines in `QHDOPT`, we note that generic local optimizers allowing box constraints should work as well.

Because `QHDOPT` leverages local search algorithms as refiners, the output solutions are necessarily locally optimal (i.e., first- or second-order stationary points, depending on the choice of refinement subroutine). That being said, we would like to note that the QHD algorithm, when executed on a large fault-tolerant quantum computer, is able to find the global minimum for a large family of nonconvex functions with mild assumptions, provided that the run time is sufficiently long (Leng et al. 2023b, theorem 2). The performance of `QHDOPT` for practical problems, however, heavily depends on the quality of near-term quantum devices, which are often of limited scale and prone to physical noise. Meanwhile, it is also possible to refine the quantum-generated solutions using a global solver (e.g., Gurobi, BARON). In this case, the global optimality is guaranteed, but the postprocessing time could be significantly longer. Because of the limited time frame, we leave a global-solver-based refinement as future work.

1.4. Unique Design Features

In what follows, we discuss a few unique design features of our software.

1.4.1. Hamiltonian-Oriented Programming. `QHDOPT` exploits the QHD algorithm to solve nonlinear optimization problems. This quantum algorithm is formulated as a Hamiltonian simulation (i.e., simulating the evolution of a quantum-mechanical system), encompassing a novel abstraction of computation on quantum devices, which we call Hamiltonian-oriented programming. In contrast to the conventional circuit-based quantum computation paradigm, where theorists describe quantum algorithms in terms of quantum circuits, the HOP paradigm describes quantum algorithms as a single or a sequence of quantum Hamiltonian evolution. This new paradigm enables us to build a stack of quantum applications by leveraging the native programmability of quantum hardware in the development of quantum algorithms and software, as illustrated in Figure 1(a). The HOP paradigm is empowered by `SimuQ`, a recent framework for programming and compiling quantum Hamiltonian systems by Peng et al. (2024). In `SimuQ`, the programming and simulation of quantum Hamiltonian systems are wrapped in user-friendly Python methods. This makes the high-level programming and deployment of Hamiltonian-oriented quantum algorithms accessible to users with little exposure to real-machine engineering and manipulation. A detailed discussion on the Hamiltonian programming and compilation in `QHDOPT` is available in Section 3.

1.4.2. Multibackend Compatibility. In `QHDOPT`, we utilize `SimuQ` as an intermediate layer for the programming of QHD and leverage the `SimuQ` compiler to realize multibackend compatibility. Through `SimuQ`, `QHDOPT` initially constructs a hardware-agnostic Hamiltonian representation of QHD (i.e., Hamiltonian embedding) that can be deployed on various quantum backends, including D-Wave devices, IonQ devices, and classical simulators via QuTiP (Johansson et al. 2012).

1.4.3. Automatic Differentiation. `QHDOPT` relies on JAX, a high-performance numerical computing library, to perform automatic differentiation of smooth, nonlinear objective functions. This feature enables `QHDOPT` to seamlessly postprocess quantum-generated solutions using local search optimizers.

2. Quantum Hamiltonian Descent

In our software, we utilize QHD to solve box-constrained nonlinear optimization problems as described in (1.1). QHD solves a continuous optimization problem by simulating a quantum dynamical system governed by an evolutionary partial differential equation called the *Schrödinger equation*. Here, we give a high-level review of this quantum algorithm, and more details can be found in Leng et al. (2023b).

2.1. Mathematical Formulation and Interpretation

Consider a nonlinear objective function $f(x)$ with a box constraint $\Omega = \{(x_1, \ldots, x_n) \in \mathbb{R}^n : L_i \leq x_i \leq U_i, \ \forall i = 1, \ldots, n\}$. To solve this optimization problem, QHD requires simulating the following Schrödinger equation over the feasible set Ω with Dirichlet boundary condition, that is, $\Psi(t,x) = 0$ for $x \in \partial\Omega$,

$$i\frac{\partial}{\partial t}\Psi(t,x) = \left[e^{\varphi_t}\left(-\frac{1}{2}\Delta\right) + e^{\chi_t}f(x)\right]\Psi(t,x), \tag{2.1}$$

subject to an initial state $\Psi(t,x) = \Psi_0(x)$. Here, the operator $\Delta := \sum_{i=1}^{n}\frac{\partial^2}{\partial x_i^2}$ is the Laplacian operator defined in the interior of Ω, and the time-dependent functions e^{φ_t} and e^{χ_t} control the total energy distribution of the quantum system. In practice, the initial state $\Psi_0(x)$ is often chosen as a quantum state that is easy to prepare, for example, a Gaussian state or a uniformly random state. For general (nonconvex) optimization problems, it is observed that an inverse polynomially decaying e^{φ_t} and polynomially increasing e^{χ_t} (e.g., $\varphi_t = -\log(1+\gamma t^2)$, $\chi_t = \log(1+\gamma t^2)$ with a positive γ) work well for many test problems (Leng et al. 2023b). With a Gaussian initial state and smooth time-dependent functions, the dynamics generated by (2.1) can be simulated using $\tilde{\mathcal{O}}(nT)$ elementary gates and $\tilde{\mathcal{O}}(T)$ queries to the objective function f (Childs et al. 2022).

Physically, Equation (2.1) describes the time evolution of a quantum particle in the box Ω. The time-dependent functions e^{φ_t} and e^{χ_t} control the total energy distribution of this quantum particle: when their ratio, e^{φ_t}/e^{χ_t}, is large, the kinetic energy dominates and the particle tends to bounce around; otherwise, the potential energy takes over and the particle tends to stay still. If we choose these functions such that $\lim_{t\to\infty} e^{\varphi_t}/e^{\chi_t} = 0$, the kinetic energy of the system is dissipated over time, and eventually the quantum particle will take a low-energy configuration. At this point, if we measure this quantum particle, the measured position (which must lie in the feasible set Ω) is likely to give an approximate solution to the problem $f(x)$. In some sense, QHD can be regarded as a quantum version of Polyak's heavy ball method (Polyak 1964, Attouch et al. 2000).

QHD describes a quantum particle exploring the optimization landscape $f(x)$. When a high-energy barrier emerges, the quantum particle may leverage the quantum tunneling effect to go through the barrier and find a lower local minimum. However, simulating the quantum evolution (2.1) with a classical computer would require exponential space and time, making this idea impractical as a classical optimization algorithm. On the other hand, the evolution (2.1) can be efficiently simulated using a quantum computer, which makes QHD a genuine quantum algorithm that can leverage the quantum tunneling effect for nonconvex optimization. Theoretically, it has been shown that QHD can efficiently find the global minimum for certain nonconvex problems with exponentially many local minima, whereas many classical optimizers such as simulated annealing (SA) and stochastic gradient descent (SGD) appear to require a much longer time to obtain a global solution (Leng et al. 2023a). Numerical experiments also show that QHD outperforms classical first- and second-order methods in a broad class of nonconvex problems with many local stationary points (Leng et al. 2023b).

2.2. Real-Machine Implementation

Quantum Hamiltonian descent is formulated as a *Hamiltonian simulation* task, that is, solving a quantum Schrödinger equation as in (2.1). Although efficient quantum algorithms, such as those proposed by Childs et al. (2022), can tackle this simulation task exponentially faster than any known classical algorithms, these quantum simulation algorithms require large fault-tolerant quantum computers. Such ideal quantum computing hardware has not yet been realized because of the immature progress of quantum technology.

To fully exploit the limited programmability of current quantum hardware such as D-Wave and IonQ, `QHDOPT` employs a technique named Hamiltonian embedding (Leng et al. 2024) to implement QHD. This technique enables us to map the QHD Hamiltonian to a larger Hamiltonian, and the latter can be *natively* simulated on existing quantum devices. This real-machine implementation technique is detailed in Section 3.2.

3. The Workflow of `QHDOPT`

3.1. Modeling of Nonlinear Problems

`QHDOPT` offers support for two Python-based input formats: the SymPy format for *symbolic* input and the QP format for *numerical* input (i.e., arrays). These two input formats enable users to define their target optimization problems both efficiently and with great flexibility.

3.1.1. SymPy Format. `SymPy` (Meurer et al. 2017) is a Python package that supports symbolic expression processes. Users can specify $f(x)$ in (1.1) by declaring variables in `SymPy` and constructing the expression, as in the code snippet in Figure 2(a). Here, we import necessary functions like exp from `SymPy` and `QHD` from package

Figure 2. (Color online) Input Formats in `QHDOPT`

(a) An example using the SymPy input format

```
1 from qhdopt import QHD
2 from sympy import symbols, exp
3
4 x, y = symbols("x y")
5 f = y**1.5 - (y-0.75) * exp(4*x)
6 model = QHD.SymPy(f, [x, y])
```

(b) An example using the QP input format

```
1 from qhdopt import QHD
2
3 Q = [[-8, 3],
4      [3, -4]]
5 b = [3, -1]
6 model = QHD.QP(Q, b)
```

Note. `QHDOPT` supports both symbolic and numerical input formats.

`QHDOPT` in Lines 1 and 2. We declare variables `x` and `y` in `SymPy` using symbols in Line 3, where the passed string is for `SymPy` to print the expressions. In Line 4, we construct the function $f(x)$, where `y**1.5` represents the exponential $y^{1.5}$, `exp(4*x)` represents e^{4x}, and so on. Last, we create a QHD model instance in Line 5 and pass `f` and a symbol list `[x, y]` to it, informing the QHD model the target optimization function is f with symbols x and y.

3.1.2. QP Format. For users with specific interests in QP, we provide a more efficient input model for them. To specify a QP instance with objective function $f(x) = \frac{1}{2}x^\top Qx + b^\top x$, we can directly input the matrices Q and b, as in the code snippet in Figure 2(b). First, we construct Q by a nested Python list or a NumPy array in Lines 2 and 3. It is required that Q forms a symmetric square matrix. Then we input the vector b as `b`. Similar to `SymPy`, we construct the instance by calling the `QP` method from `QHDOPT` and pass `Q`, `b` into it.

3.2. Hamiltonian Programming and Compilation

Once a nonlinear optimization problem $f(x)$ is defined using one of the supported input formats, `QHDOPT` will form a Hamiltonian description of the corresponding QHD algorithm, as described in (2.1). This Hamiltonian description serves as an intermediate layer in the compilation stack and is independent of the choice of the backend (i.e., hardware agnostic). Although `QHDOPT` automates this process, making manual execution unnecessary in most cases, we provide detailed discussions for readers who are interested in gaining a deeper understanding of our software's design.

There are two major steps in the construction of the Hamiltonian description of QHD, namely, *spatial discretization* and *Hamiltonian embedding*.

3.2.1. Spatial Discretization. First, we need to perform spatial discretization of the QHD Hamiltonian (which is an unbounded operator) so that it can be described by a finite-dimensional quantum system. For a thorough and mathematically rigorous discussion, readers are encouraged to refer to Leng et al. (2023b, appendix F.2.1). Given a nonlinear optimization in the form of (1.1), the QHD Hamiltonian reads the following:

$$H(t) = e^{\varphi_t}\left(-\frac{1}{2}\Delta\right) + e^{\chi_t}\left(\sum_{i=1}^{n} g_i(x_i) + \sum_{j=1}^{m} p_j(x_{k_j})q_j(x_{\ell_j})\right),$$

which acts on any L^2-integrable functions over the feasible set $\Omega = \{(x_1,\ldots,x_n) \in \mathbb{R}^n : L_i \le x_i \le U_i,\ \forall i = 1,\ldots,n\}$. Here, for simplicity, we assume the feasible set is the unit box, that is, $L_i = 0$ and $U_i = 1$ for all $i = 1,\ldots,n$. We utilize the centered finite difference scheme to discretize this differential operator. Suppose that we divide each dimension of the unit box Ω using N quadrature points $\{0, h, \ldots, (N-2)h, 1\}$ (where $h = 1/(N-1)$); the resulting discretized QHD Hamiltonian is an N^n-dimensional operator of the form

$$\hat{H}(t) = e^{\varphi_t}\left(-\frac{1}{2}L_d\right) + e^{\chi_t}F_d, \tag{3.1}$$

where (assuming $k_j < \ell_j$ for all $j = 1,\ldots,m$)

$$L_d = \sum_{i=1}^{n} I\otimes\cdots\otimes \underbrace{L}_{\text{the } i\text{th operator}} \otimes\ldots I,$$

$$F_d = \sum_{i=1}^{n} I\otimes\cdots\otimes \underbrace{D(g_i)}_{\text{the } i\text{th operator}} \otimes\ldots I + \sum_{j=1}^{m} I\otimes\cdots\otimes \underbrace{D(p_j)}_{\text{the } k_j\text{th operator}} \otimes\cdots\otimes \underbrace{D(q_j)}_{\text{the } \ell_j\text{th operator}} \otimes\ldots I.$$

Here, I is the N-dimensional identity operator, and L and $D(g)$ are N-dimensional matrices given by (g is a differentiable function defined on $[0,1]$, and $g_i := g(ih)$ for $i = 0, \ldots, N-1$)

$$L = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & \\ 1 & -2 & 1 & \\ \cdots & \cdots & \cdots & \cdots \\ & 1 & -2 & 1 \\ & & 1 & -2 \end{bmatrix}, \quad D(g) = \begin{bmatrix} g_0 & & & & \\ & g_1 & & & \\ \cdots & \cdots & \cdots & & \cdots \\ & & g_{N-2} & & \\ & & & & g_{N-1} \end{bmatrix}.$$

The tridiagonal L matrix corresponds to the finite difference discretization of the second-order differential operator $\frac{d^2}{dx^2}$, and the diagonal matrix $D(g)$ corresponds to the finite difference discretization of the univariate function $g(x)$. Note that L has a global phase $-2/h^2$; that is, $L = L' - 2/h^2$, with L' containing only the off-diagonal part of L. Because the global phase does not affect the quantum evolution (therefore, the result of the QHD algorithm), we replace L with L' in the rest of the discussion.

3.2.2. Hamiltonian Embedding. The discretized QHD Hamiltonian, as described in (3.1), is a Hermitian matrix with an explicit tensor product decomposition structure. This particular structure allows us to leverage the Hamiltonian embedding technique (Leng et al. 2024) to construct a surrogate Hamiltonian $\tilde{H}(t)$ such that the QHD algorithm (i.e., simulating the Hamiltonian $\hat{H}(t)$) can be executed by simulating $\tilde{H}(t)$. In our case, the surrogate Hamiltonian $\tilde{H}(t)$ is an Ising-type quantum Hamiltonian that involves at most nN qubits and $\max(n,m)N$ two-body interaction terms. This means $\tilde{H}(t)$ can be efficiently simulated on current quantum computers, including IonQ's trapped ion systems and D-Wave's quantum annealer.

To construct the Hamiltonian embedding of $\hat{H}(t)$, the first step is to build the Hamiltonian embeddings of the N-by-N matrices L' and $D(g)$ (for arbitrary differentiable function g). Both are sparse matrices, so we can utilize the embedding schemes provided in Leng et al. (2024, section 2.3). `QHDOPT` allows users to choose from three embedding schemes: Hamming,[3] unary, and one-hot.[4] In Table 1, we list the details of these embedding schemes when applied to L' and $D(g)$. We note that the Hamming embedding scheme only works for quadratic programming, whereas the other two schemes (unary, one-hot) work for a broader class of nonlinear functions such as exponential functions. To be consistent with our source code, we adopt the left-to-right zero-indexing system for bits/qubits, for example, $1_0 0_1 1_2 1_3$.

In Table 1, the integer r represents the number of qubits used to embed an N-dimensional matrix. The operators $\mathbf{X}_k$, $\mathbf{Y}_k$, and $\mathbf{n}_k$ are the Pauli-X, Pauli-Y, and number operator acting at site k, respectively,

$$\mathbf{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \mathbf{n} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Because the Hamming embedding scheme is allowed only for quadratic programming, we do not consider the Hamming embedding for general nonlinear functions g. Instead, we consider only the embedding of the identity and quadratic functions, that is, $g(x) = x$ and $g(x) = x^2$; their corresponding Hamming embeddings are $\mathcal{E}_1 = \frac{1}{r}\sum_{k=0}^{r-1}\mathbf{n}_k$ and $\mathcal{E}_2 = (\mathcal{E}_1)^2$, respectively.

Now, we denote $\mathcal{E}^{(i)}[A]$ as a Hamiltonian embedding of an N-by-N Hermitian matrix A acting on sites $(i-1)r, (i-1)r+1, \ldots, ir-1$, where $i = 1, \ldots, n$. Then, using the rules of building Hamiltonian embeddings (Leng et al. 2024, theorem 2), we obtain an nr-qubit Hamiltonian that embeds the discretized QHD Hamiltonian $\hat{H}(t)$:

$$\tilde{H}(t) = e^{\varphi_t} \quad -\frac{1}{2}\sum_{i=1}^{n}\mathcal{E}^{(i)}[L']\Bigg) + e^{\chi_t}\left(\sum_{i=1}^{n}\mathcal{E}^{(i)}[D(g_i)] + \sum_{j=1}^{m}\mathcal{E}^{(k_j)}[D(p_j)]\mathcal{E}^{(\ell_j)}[D(q_j)]\right). \tag{3.2}$$

Table 1. Embedding Schemes Supported by `QHDOPT`

Embedding scheme	Supported input format	Supported backend	Number of qubits	Embedding of L'	Embedding of $D(g)$
Hamming	QP, SymPy	QuTiP, IonQ, D-Wave	$r = N-1$	$\sum_{k=0}^{r-1}\mathbf{X}_k/h^2$	Only for QP, see discussions in Section 3.2.2
Unary	QP, SymPy	QuTiP, IonQ, D-Wave	$r = N-1$	$\sum_{k=0}^{r-1}\mathbf{X}_k/h^2$	$\sum_{k=0}^{r-1}(g_{r-k} - g_{r-k-1})\mathbf{n}_k + g_0\mathbf{I}$
One-hot	QP, SymPy	QuTiP, IonQ	$r = N$	$\sum_{k=0}^{r-2}(\mathbf{X}_k\mathbf{X}_{k+1} + \mathbf{Y}_k\mathbf{Y}_{k+1})/(2h^2)$	$\sum_{k=0}^{r-1} g_{r-1-k}\mathbf{n}_k$

Example 3 (One-Hot Embedding for x_1x_2). We give a simple example for the Hamiltonian embedding of the discretized QHD Hamiltonian when the objective function is $f(x_1, x_2) = x_1x_2$. This objective involves only a single bivariate term with $p(x) = q(x) = x$. We use the one-hot embedding with $N = r = 3$. Then, the Hamiltonian embeddings of L' and $D(x)$ are

$$\mathcal{E}[L'] = \frac{1}{2h^2}(\mathbf{X}_0\mathbf{X}_1 + \mathbf{X}_1\mathbf{X}_2 + \mathbf{Y}_0\mathbf{Y}_1 + \mathbf{Y}_1\mathbf{Y}_2), \quad \mathcal{E}[D(x)] = \mathbf{n}_0 + \frac{1}{2}\mathbf{n}_1,$$

respectively. As a result, the full Hamiltonian embedding reads

$$\tilde{H}(t) = \frac{2e^{\varphi_t}}{h^2}(\mathbf{X}_0\mathbf{X}_1 + \mathbf{X}_1\mathbf{X}_2 + \mathbf{X}_3\mathbf{X}_4 + \mathbf{X}_4\mathbf{X}_5 + \mathbf{Y}_0\mathbf{Y}_1 + \mathbf{Y}_1\mathbf{Y}_2 + \mathbf{Y}_3\mathbf{Y}_4 + \mathbf{Y}_4\mathbf{Y}_5) + \\ e^{\chi_t}\left(\mathbf{n}_0 + \frac{1}{2}\mathbf{n}_1\right)\left(\mathbf{n}_3 + \frac{1}{2}\mathbf{n}_4\right).$$

In `QHDOPT`, we use `SimuQ` to construct the Hamiltonian embedding $\tilde{H}(t)$. The users need to specify only the number of qubits r (for each continuous variable), the embedding scheme, and a desired backend in the `model.optimize()` function, as detailed in the next subsection.

3.3. Deployment and Postprocessing

When the Hamiltonian embedding $\tilde{H}(t)$ of a given problem is built, it can be executed on a supported quantum backend by running `optimize()` (refer to Section 4 for sample code). The quantum measurement results are then retrieved from the executing backend in the form of bitstrings. Following this, `QHDOPT` implements a series of classical postprocessing subroutines. These include decoding the raw measurement results (i.e., bitstrings) into low-resolution solutions and refining them via a classical local solver. The refined solutions are then returned to the users as final results.

3.3.1. Deployment on Quantum Devices. Currently, `QHDOPT` supports three backend devices for deployment, including classical simulators (e.g., QuTiP), IonQ, and D-Wave. For all three backend devices, the quantum register is initialized to the uniform superposition state. On the IonQ device, the uniform superposition state can be prepared using a single layer of Hadamard gates; on the D-Wave device, the uniform superposition state is the default initial state, and it can be prepared in microseconds. When deployed on IonQ, `QHDOPT` uses $\varphi_t = -\log(1 + \gamma t^2)$ and $\chi_t = \log(1 + \gamma t^2)$ for the time-dependent functions (see Section 2.1 for details). The time-dependent functions on D-Wave are more restricted, and they can be specified only as piecewise linear functions. We find the default annealing schedule (20 ms) provided by the D-Wave device usually works well in practice. We also showcase user-specified time-dependent functions (annealing schedules) in a notebook in the "examples" folder.

Here, we demonstrate the deployment procedures in `QHDOPT` using the D-Wave backend, although the same process applies to the other two backends. In Figure 3, a snippet of the source code for the function `QHD.dwave_exec()` is displayed. After programming the Hamiltonian embedding and the quantum system realizing QHD (Lines 2 and 3), we initiate an abstract D-Wave machine (Line 5). Then `QHDOPT` employs `SimuQ` to

Figure 3. (Color online) Deploying and Executing QHD on the D-Wave Quantum Computer

```
 1  def dwave_exec(self, verbose=0):
 2      Hamiltonian, T = ...
 3      self.qs.add_evolution(Hamiltonian, T) # Hamiltonian embedding
 4
 5      dwp = DWaveProvider(self.api_key) # initiate D-Wave machine
 6      ...
 7      dwp.compile(self.qs, self.shots)
 8      ...
 9      dwp.run()
10      ...
11
12      self.raw_result = dwp.results()
13      raw_samples = []
14      for i in range(self.shots):
15          raw_samples.append(QHD.spin_to_bitstring(self.raw_result[i]))
16
17      return raw_samples
```

Figure 4. (Color online) Bitstring-to-Vector Decoder in QHDOPT

```
def bitstring_to_vec(self, bitstring, d, r):
    if self.embedding_scheme == 'unary':
        return QHD.unary_bitstring_to_vec(bitstring, d, r)
    elif self.embedding_scheme == 'onehot':
        return QHD.onehot_bitstring_to_vec(bitstring, d, r)
    elif self.embedding_scheme == 'hamming':
        return QHD.hamming_bitstring_to_vec(bitstring, d, r)
    else:
        raise Exception("Illegal embedding scheme.")
```

compile the Hamiltonian embedding into low-level device instructions readable by D-Wave (Line 7) using `SimuQ's DWaveProvider()`, effectively generating Hamiltonian $H_{\text{dev}}(t)$ on the D-Wave devices. Next, the instructions are sent to the D-Wave quantum computer to execute (Line 9), and the raw quantum samples are collected by QHDOPT as bitstrings (Lines 12–17).

3.3.2. Decoding.

As we have seen, the real-machine results are in the bitstring format because they are retrieved by computational basis measurements in the quantum computer. These bitstrings need to be converted to floating-point arrays via the built-in decoder, as presented in Figure 4. This decoder maps a bitstring to a floating-point array that represents a low-resolution solution to the input optimization problem. For example, if we use the unary embedding for a two-dimensional problem with resolution parameter $r = 4$, the decoder will map an eight-bit string to a length two array. For example, 00010011 is mapped to $[0.25, 0.5]$. More details of the embedding schemes and their decoding are available in Leng et al. (2024).

3.3.3. Refinement.

Limited by the size of current quantum devices, in most cases, we can only use a small resolution parameter (e.g., $r = 8$) in the real-machine implementation of QHD. Therefore, the retrieved measurement results are merely low-resolution solutions to the specified optimization problem. To improve the precision of the solutions, QHDOPT then postprocesses the measurement results using local search classical optimization methods.

In principle, any generic local optimizers allowing box constraints should work as well; because of the limited resources, we provide two classical refinement options for the users in QHDOPT, including the TNC using SciPy and the interior point method using Ipopt. TNC is a quasi-Newton method, and the interior point method exploited by Ipopt is a second-order method. These classical refiners require the gradient and/or Hessian information of the objective functions. For quadratic programming problems (using the QP input format), the gradient and Hessian can be computed explicitly:

$$\nabla f(x) = Qx, \quad Hf(x) = Q.$$

For more general nonlinear optimization problems specified using the SymPy input format, QHDOPT computes the gradient and Hessian information by employing JAX (Frostig et al. 2018), a high-performance numerical computing library developed by Google, to perform autodifferentiation.

The postprocessed results can be retrieved from `model.post_processed_samples`. By default, the postprocessing subroutine is enabled and automatically executed by running `optimize()`. However, users can also disable postprocessing by specifying `optimize(fine_tune=False)`. In this case, `model.post_processed_samples` returns None type.

4. Examples Using QHDOPT

In this section, we exhibit two simple examples showcasing the use cases of QHDOPT.

4.1. Quadratic Programming

We first consider a two-dimensional quadratic programming problem, whose objective function is defined as follows:

$$f(x,y) = -x^2 + xy - \frac{1}{2}y^2 + \frac{3}{4}x - \frac{1}{4}y = \frac{1}{2}[x \quad y]\begin{bmatrix} -2 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{3}{4} & -\frac{1}{4} \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}, \tag{4.1}$$

for $x, y \in [0,1]$.

Figure 5. (Color online) Solving a Quadratic Programming Problem Using `QHDOPT`

```
 1  from qhdopt import QHD
 2
 3  # Using QP input format
 4  Q = [[-2, 1],[1, -1]]
 5  b = [3/4, -1/4]
 6  model = QHD.QP(Q, b, bounds=(0,1))
 7
 8  # Deployment # 1: D-Wave (default embedding scheme: unary)
 9  model.dwave_setup(8, api_key="DWAVE_API_KEY")
10  model.optimize()
11
12  # Deployment # 2: IonQ (default embedding scheme: one-hot)
13  model.ionq_setup(6, api_key="IONQ_API_KEY", time_discretization=30)
14  model.optimize()
15
16  # Model 3: QuTiP simulator (default embedding scheme: one-hot)
17  model.qutip_setup(6, post_processing_method="IPOPT")
18  model.optimize()
```

In Figure 5, we exemplify using `QHDOPT` to solve this QP problem with all three backends. Note that the application programming interface (API) key (not included in `QHDOPT`) is required to access cloud-based quantum computers such as D-Wave and IonQ. By default, `QHD.optimize()` automatically executes the SciPy TNC method to fine-tune the raw quantum measurement data. To switch to the Ipopt optimizer in the fine-tuning step, one can specify `post_processing_method="IPOPT"` in the setup, as shown in line 17.

4.2. Nonlinear Optimization Involving Exponential Function

Next, we consider the following nonlinear minimization problem with the objective function

$$f(x,y) = y^{3/2} - e^{4x}\left(y - \frac{3}{4}\right), \quad x,y \in [0,1]. \tag{4.2}$$

This objective function $f(x, y)$ is not a polynomial; it involves a fractional power and an exponential function. In Figure 6(a), we illustrate a sample code that runs `QHDOPT` to solve the problem defined above. The function is constructed using the SymPy input format, then deployed on the D-Wave quantum computer with resolution parameter $r = 8$. We may set `verbose = 1` to print a detailed summary of this execution, including the best-so-far coarse and fine-tuned solutions, as well as a total run-time breakdown, as shown in Figure 6(b).

5. The State of Software for Quantum Optimization

Software packages are crucial for lowering the barrier to developing and implementing quantum programs across broad user communities. Upon examining the current landscape of quantum software for mathematical optimization, we observe that the majority of the software dedicated to quantum optimization focuses on

Figure 6. (Color online) Solving a Nonlinear Optimization Problem Using `QHDOPT`

(a) Deploying QHD on D-Wave

```
from sympy import symbols, exp
from qhdopt import QHD

# Using SymPy input format
x, y = symbols("x y")
f = y**1.5 - exp(4*x) * (y-0.75)
model = QHD.SymPy(f, [x, y],
                  bounds=(0,1))

# Deploying QHD on the D-Wave device
model.dwave_setup(8, api_key="API_KEY")
model.optimize(verbose=1)
```

(b) Execution summary

```
* Coarse solution
    Minimizer: [1. 0. 1.]
    Minimum: -1.0
* Fine-tuned solution
    Minimizer: [1. 0. 1.]
    Minimum: -1.0
    Success rate: 1.0
* Runtime breakdown
    SimuQ compilation: 0.000 s
    Backend QPU runtime: 0.119 s
    Backend overhead time: 3.825 s
    Decoding time: 0.019 s
    Fine-tuning time: 0.161 s
* Total time: 4.124 s
```

Note. The backend overhead time includes network transmission time, queue time, etc.

addressing combinatorial and discrete optimization problems, with limited options available for continuous optimization.

Generally, a combinatorial optimization problem can be reformulated as a quadratically unconstrained binary optimization (QUBO) problem, the solution of which is believed to be a promising application of quantum computing (Quintero and Zuluaga 2023). There is a rich collection of libraries for quantum and quantum-inspired optimization that can be employed to generate QUBO reformulations, including `QUBO.jl` (Xavier et al. 2023), `Amplify` (Matsuda 2020), `PyQUBO` (Zaman et al. 2021), and `qubovert` (Iosue 2022). These QUBO problems can be tackled by several methods, such as quantum annealing, quantum approximate optimization algorithm (QAOA; Farhi et al. 2014), and other hybrid approaches (Yamamoto et al. 2017). D-Wave's `Ocean SDK` (D-Wave Quantum Systems Inc. 2023) enables users to interface with their direct QPU (i.e., quantum annealer) and hybrid solvers and retrieve results. QuEra's `Bloqade.jl` (QuEra Computing Inc. 2023) is a high-level language for configuring programmable Rydberg atom arrays that can be used to implement annealing-type quantum algorithms and discrete optimization problems like QUBO (Nguyen et al. 2023). The Los Alamos Advanced Network Science Initiative has also released a package, named `QuantumAnnealing.jl` (Morrell et al. 2024), for the simulation and execution of quantum annealing. Besides, several software packages have been published for programming quantum circuits, including IBM's `Qiskit` (Aleksandrowicz et al. 2019), Google's `Cirq` (The Cirq Developers 2019), Amazon's `Braket SDK` (Amazon Web Services 2024), Microsoft's `Q#` (Singhal et al. 2022), and Xanadu's `PennyLane` (Bergholm et al. 2018). These tools can be used to deploy QAOAs on gate-based quantum computers.

Although there have been a few proposals for solving continuous optimization problems using quantum or hybrid computing devices such as photonic quantum computers (Verdon et al. 2019) and coherent continuous variable machines (Khosravi et al. 2022), we are not aware of a software library customized for nonlinear continuous optimization problems. In practice, some nonlinear optimization problems, such as quadratic programming, may be reformulated as QUBO problems and handled by the aforementioned software tools. However, it remains unclear whether this approach could lead to robust quantum advantages.

6. Comparison with Existing Tools

As we discussed in Section 3, `QHDOPT` first obtains some low-resolution solutions by executing the QHD algorithm for a nonlinear optimization problem through Hamiltonian embedding. Next, the software employs a classical local search strategy for fast postprocessing of the raw quantum results. It is of interest to understand to what extent the quantum component (i.e., the noisy implementation of QHD) improves the overall performance of `QHDOPT`. To this end, we have designed a benchmark test to evaluate the performance of `QHDOPT` for nonlinear and nonconvex optimization problems.

6.1. Test Problems

We demonstrate the performance of `QHDOPT` using 15 randomly generated nonlinear optimization instances, all with unit box constraints. Problem Instances 1–5 are NLP problems involving two or three continuous variables, as detailed in Table 2. Problem Instances 6–10 are QP problems drawn from the benchmark devised in Leng et al. (2023b). Problem Instances 11–15 are NLP problems involving exponential functions, as specified in the following expression:

$$f(x) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} Q_{i,j} e^{x_i} e^{x_j} + \sum_{i=1}^{N} b_i e^{-x_i}. \tag{6.1}$$

The last 10 test instances (6–15) are intermediate-scale problems with 50 continuous variables, ranging from zero to one. To ensure the successful mapping of these test problems to quantum computers with limited connectivity, these test problems are generated in a way such that their Hessians are sparse matrices.[5]

Table 2. Problem Instances 1–5 for Nonlinear Programming

Test index	Problem description
1	$f(x,y) = -4x^2 + 3xy - 2y^2 + 3x - y$
2	$f(x,y) = -2\left(x - \frac{1}{3}\right)^2 + y^2 - \frac{1}{3}y\log\left(3x + \frac{1}{2}\right) + 5\left(x^2 - y^2 - x - \frac{1}{2}\right)^2$
3	$f(x,y) = y^{3/2} - e^{4x}\left(y - \frac{3}{4}\right)$
4	$f(x,y,z) = (2y-1)^2\left(z - \frac{2}{5}\right) - (2x-1)z + y\left(2x - \frac{3}{2}\right)^2$
5	$f(x,y,z) = 2e^{-x} * (2z-1)^2 - 3\left(2y - \frac{7}{10}\right)^2 e^{-z} + \log(x+1)\left(y - \frac{4}{5}\right)$

Note. All the test instances are nonconvex problems with unit box constraints $[0,1]^n$.

These instances were generated in a largely random manner, and each possesses multiple local solutions, making them fairly challenging for classical optimization software. In our experiment, we observed that local solvers, such as Ipopt, cannot find globally optimal (or even approximately optimal) solutions unless a large number of random initial guesses are tried. BARON, a highly optimized commercial solver for global optimization, can find globally optimal solutions to sparse quadratic programming problems in under one second, but it takes several minutes to certify global optimality for nonlinear programming problems that involve exponential-type objectives.

6.2. Experiment Setup and Results

In this subsection, we discuss the basic setup of the experiment and the numerical results. We test `QHDOPT` with two different postprocessing optimizers (i.e., Ipopt and SciPy TNC) using the randomly generated nonlinear programming instances discussed in the previous section. As a comparison, we also run the two classical optimizers on the same test instances using uniformly random initialization. These two classical optimizers are assessed as baselines to illustrate the quantum advantage introduced by the D-Wave-implemented QHD. The classical components in both experiments, including the decoding of D-Wave samples and classical refinement, were executed on a 2022 MacBook Pro laptop with an Apple M2 chip. All the code is available in the online repository (Kushnir et al. 2024). Our findings assert that QHD, when implemented with D-Wave, brings a significant advantage compared with the standalone use of classical optimizers.

6.2.1. Experiment Setup for `QHDOPT`. We evaluate `QHDOPT` on this benchmark using the D-Wave Advantage_system6.3 as the quantum backend. For a fair comparison, we use the unary embedding scheme for all instances, including quadratic and nonquadratic problems. The anneal time is set to be the default value, that is, 20 ms. The total quantum run time per shot (see the "QPU" columns in Table 3) is calculated as the arithmetic mean of the "qpu_access_time" reported by D-Wave, which includes the programming, state preparation, annealing, and decoding. Note that we do not include the transmission time and the task queuing time in our report. We test `QHDOPT` with two postprocessing optimizers (i.e., Ipopt and SciPy TNC), and the classical postprocessing (more precisely, classical refinement) time is reported in the "Classical refine" columns in Table 3. The standard deviation of the classical refinement time is also reported in the parenthesis. Except for the initial guesses, both solvers use the default parameters as provided with their Python API.

6.2.2. Baseline Using Classical Optimizers. As a comparison, we also test three classical optimizers for the same set of problems: Ipopt, SciPy TNC, and BARON. The first two optimizers are initialized with 1,000 uniformly

Table 3. Performance of QHDOPT on 15 Randomly Generated Nonlinear Programming Test Instances

	QHD+Ipopt				QHD+TNC			
Test index	QPU	Classical refine	SP	TTS	QPU	Classical refine	SP	TTS
1	1.15e−3	2.03e−1 (5.87e−4)	9.96e−1	2.05e−1	1.15e−3	3.19e−2 (1.41e−5)	9.84e−1	3.68e−2
2	9.97e−4	3.16e−1 (8.08e−3)	9.23e−1	5.69e−1	9.97e−4	5.24e−2 (1.16e−4)	9.12e−1	1.11e−1
3	1.08e−3	3.35e−1 (2.88e−3)	9.40e−1	5.50e−1	1.08e−3	2.96e−2 (1.17e−4)	9.82e−1	3.52e−2
4	1.25e−3	1.6e+0 (5.48e−3)	7.98e−1	4.60e+0	1.25e−3	1.28e−1 (2.68e−4)	8.67e−1	2.95e−1
5	1.23e−3	3.64e−1 (1.08e−3)	9.62e−1	5.14e−1	1.23e−3	5.75e−2 (3.34e−5)	9.82e−1	6.73e−2
6	1.69e−3	2.03e−2 (5.96e−3)	2.39e−1	3.74e−1	1.69e−3	1.57e−3 (4.85e−4)	2.08e−1	6.51e−2
7	2.04e−3	2.34e−2 (1.64e−3)	1.20e−1	9.42e−1	2.04e−3	1.99e−3 (4.92e−4)	1.20e−1	1.49e−1
8	1.49e−3	1.83e−2 (1.93e−3)	9.23e−1	3.96e−2	1.49e−3	1.08e−3 (2.81e−4)	9.78e−1	5.14e−3
9	2.01e−3	1.81e−2 (3.90e−3)	4.36e−1	1.82e−1	2.01e−3	2.35e−3 (3.48e−4)	3.92e−1	4.44e−2
10	2.11e−3	2.02e−2 (5.62e−3)	1.09e−1	8.94e−1	2.11e−3	2.91e−3 (5.21e−4)	1.06e−1	2.06e−1
11	2.04e−3	9.91e−2 (2.24e−2)	3.42e−1	1.21e+0	2.04e−3	9.69e−3 (6.67e−4)	9.97e−1	1.17e−2
12	2.14e−3	9.25e−2 (1.38e−2)	4.45e−1	7.57e+0	2.14e−3	8.94e−3 (1.19e−3)	8.67e−1	3.32e−2
13	2.09e−3	1.62e−1 (2.08e−2)	2e−3	3.78e+2	2.09e−3	1.19e−2 (8.99e−4)	2.11e−1	2.80e−1
14	1.94e−3	5.89e−2 (1.33e−2)	3.83e−1	6.06e−1	1.94e−3	6.94e−3 (7.84e−4)	7.46e−1	3.56e−2
15	2.13e−3	8.26e−2 (1.13e−2)	2.53e−1	1.36e+0	2.13e−3	8.75e−3 (6.67e−4)	8.88e−1	3.26e−2

Notes. "SP" represents success probability. The unit of the quantum run time (i.e., "QPU"), classical postprocessing time (i.e., "Classical refine"), and time-to-solution (i.e., "TTS") is seconds. The standard deviation of the classical postprocessing time is shown in parentheses. The lowest TTS in a row is underlined.

Table 4. Performance of Classical Optimizers on the Same 15 Randomly Generated Test Instances as a Baseline

	Ipopt			TNC			BARON
Test index	Avg. run time	SP	TTS	Avg. run time	SP	TTS	Run time
1	1.33e+1 (2.22e−3)	5.72e−1	7.19e+1	2.82e−1 (1.29e−5)	5.64e−1	1.57e+0	1.00e−2
2	1.29e+1 (4.87e−3)	5.44e−1	7.56e+1	5.86e−1 (2.72e−4)	5.15e−1	3.73e+0	1.00e−2
3	2.09e+1 (1.61e−2)	6.78e−1	8.51e+1	6.85e−1 (1.08e−4)	5.61e−1	3.83e+0	1.00e−2
4	9.74e+0 (4.71e−3)	7.54e−1	3.20e+1	5.31e−1 (2.68e−4)	6.87e−1	2.10e+0	1.00e−2
5	1.43e+1 (2.78e−3)	6.27e−1	6.67e+1	6.42e−1 (3.30e−5)	6.23e−1	3.03e+0	1.00e−2
6	3.77e−2 (1.39e−2)	1.70e−1	9.41e−1	6.69e−3 (1.47e−3)	6.00e−3	5.12e+0	1.90e−1
7	5.44e−2 (1.50e−2)	8.00e−3	3.12e+1	5.00e−3 (7.01e−4)	2.00e−3	1.15e+1	4.00e−2
8	4.78e−2 (1.39e−2)	4.05e−1	4.30e−1	5.04e−3 (1.59e−3)	2.40e−2	9.57e−1	3.00e−2
9	4.83e−2 (1.22e−2)	7.00e−3	3.17e+1	5.83e−3 (8.76e−4)	1.00e−3	2.68e+1	3.00e−2
10	5.38e−2 (2.14e−2)	3.20e−2	7.65e+0	7.18e−3 (1.03e−3)	6.00e−3	5.50e+0	5.00e−2
11	9.88e−2 (3.23e−2)	9.28e−1	1.98e−1	9.13e−3 (1.01e−3)	7.00e−1	3.65e−2	1.20e+2
12	1.15e−1 (2.70e−2)	1.21e−1	4.15e+0	8.88e−3 (1.27e−3)	1.71e−1	2.22e−1	1.20e+2
13	9.99e−2 (2.65e−2)	1.3e−2	3.52e+1	9.19e−3 (1.08e−3)	2.77e−1	1.38e−1	1.20e+2
14	1.10e−1 (4.73e−2)	6.30e−2	7.82e+0	9.56e−3 (1.15e−3)	1.08e−1	3.92e−1	1.20e+2
15	9.07e−2 (2.81e−2)	4.40e−2	9.34e+0	9.06e−3 (1.02e−3)	2.74e−1	1.36e−1	1.20e+2

Notes. "SP" represents success probability. The unit of the classical optimizer run time (i.e., "Avg. run time" and "Run time") and time-to-solution (i.e., "TTS") is seconds. The standard deviation of the run time is shown in parentheses. The lowest TTS in a row is underlined.

random guesses in the unit box $[0,1]^d$ (where d is the problem dimension), and the run-time data for the 1,000 runs have been collected. For a fair comparison, we use the same random seeds for both methods. BARON is executed to generate global solutions with a two-minute timeout. Except for the initial guesses, all solvers use the default parameters as provided with their Python API. Table 4 shows the run times of the three classical solvers: for Ipopt and TNC, the arithmetic mean (and standard deviation) of the run time is reported; for BARON, the total run time is reported.

Note that for the last five test instances, BARON failed to certify the global optimality of the obtained solutions within the two-minute timeout window. In Table 5, we further investigate BARON's solution quality. Our results suggest that although BARON can find solutions as good as those from the other tested solvers in a comparable timescale, a much longer time is required to prove the global optimality of the obtained solutions. Therefore, we regard the solution returned by BARON as the global minimum.

Table 5. Performance of BARON

Test index	Best found obj.	BARON Result 1	BARON Result 2
1	−3	−3	−3
2	0.354	0.354	0.354
3	−12.650	−12.650	−12.650
4	−0.882	−0.882	−0.882
5	−4.196	−4.196	−4.196
6	−1.188	−1.188	−1.188
7	−2.110	−2.110	−2.110
8	−1.809	−1.809	−1.809
9	−2.269	−2.269	−2.269
10	−2.305	−2.305	−2.305
11	−31.256	−31.256	−31.256
12	−66.618	−66.618	−66.618
13	−56.762	−56.762	−56.762
14	−25.357	−25.357	−5.357
15	−59.342	−59.188	−59.342

Notes. BARON Result 1 (Result 2) indicates BARON's best-found objective function value (obj.) given a run time no longer than TNC's (Ipopt's) time-to-solution as reported in Table 4. Except for instance 15, BARON always finds the optimal solution in the given time windows.

Table 6. Time-to-Solution Data of All Four Methods (Unit: Seconds)

Test index	QHD+Ipopt	QHD+TNC	Ipopt	TNC
1	2.05e−1	3.68e−2	7.19e+1	1.57e+0
2	5.69e−1	1.11e−1	7.56e+1	3.73e+0
3	5.50e−1	3.52e−2	8.51e+1	3.83e+0
4	4.60e+0	2.95e−1	3.20e+1	2.10e+0
5	5.14e−1	6.73e−2	6.67e+1	3.03e+0
6	3.74e−1	6.51e−2	9.41e−1	5.12e+0
7	9.42e−1	1.49e−1	3.12e+1	1.15e+1
8	3.96e−2	5.14e−3	4.30e−1	9.57e−1
9	1.82e−1	4.44e−2	3.17e+1	2.68e+1
10	8.94e−1	2.06e−1	7.65e+0	5.50e+0
11	1.21e+0	1.17e−2	1.98e−1	3.65e−2
12	7.57e+0	3.32e−2	4.15e+0	2.22e−1
13	3.78e+2	2.80e−1	3.52e+1	1.38e−1
14	6.06e−1	3.56e−2	7.82e+0	3.92e−1
15	1.36e+0	3.26e−2	9.34e+0	1.36e−1

Note. The lowest TTS per instance is underlined.

6.2.3. Performance Metric. In the experiments, we use *time-to-solution* (*TTS*) as the key metric to evaluate the performance of various optimization methods. TTS is defined as the total run time required by a method to achieve at least 0.99 success probability. It can be calculated using the following formula:

$$\text{TTS} = t_0 \times \left\lceil \frac{\ln(1-0.99)}{\ln(1-p_s)} \right\rceil,$$

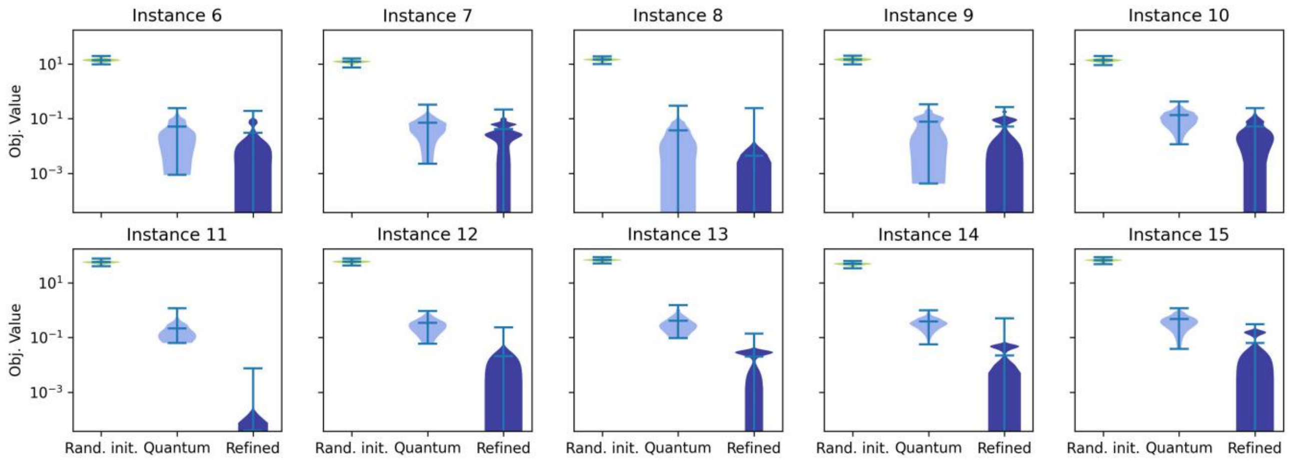
where t_0 is the (average) run time per shot, and p_s is the success probability. For all the 15 test instances, the time-to-solution data of four methods (QHD+Ipopt, QHD+TNC, Ipopt, and TNC) are presented in Table 6. For the experiments involving QHD (i.e., the results in Table 3), t_0 is calculated as the sum of average QPU time and average classical refinement time; for the experiments that involve only classical optimizers (i.e., the results in Table 4), t_0 is equivalent to the (average) classical run time. The success probability p_s is estimated by the fraction of "successful" events in the 1,000 samples/trials. Here, a result x' is considered successful if the optimality gap $f(x') - f(x^*)$ is less than 0.001, where x^* is the solution obtained by BARON.

6.3. Interpretation of the Experiment Results

Based on the time-to-solution data as reported in Table 6, we observed that QHDOPT (QHD plus a classical optimizer) always outperforms the standalone use of a classical optimizer for the 15 randomly generated test instances. As for the two local optimizers, we find that SciPy TNC works better than Ipopt as a postprocessing subroutine. Although the two optimizers usually return refined samples with comparable success probability, SciPy TNC always shows a lower TTS due to a notably faster run time. This is potentially because TNC is a quasi-Newton method that does not need to solve the full Newton linear system in the iterations.

Another interesting finding is that the classical refinement times of quantum-generated samples (see "Classical refine" in Table 3) in QHDOPT are significantly shorter than the average run time of the direct use of local optimizers (see "Avg. run time" in Table 4). For example, in Test Instance 1, the average postprocessing time (using Ipopt) for a quantum-generated initial guess is 0.2 s, whereas the average run time of Ipopt given uniformly random guesses is 13 s. To further investigate this phenomenon, we plot the distribution of objective function values corresponding to three different sample groups, including (1) randomly generated initial guesses, (2) quantum (D-Wave)–generated samples, and (3) TNC refined samples (using quantum-generated samples as initial guesses), as shown in Figure 7.[6] Although the quantum-generated solutions are limited by low precision, it is observed that they are still qualitatively better than random initial guesses. In the subsequent postprocessing, QHDOPT performs a local search subroutine to refine solution quality by improving numerical accuracy. In other words, the quantum sampler in QHDOPT can be regarded as a fast and efficient warm start that devises initial guesses of better quality.

Figure 7. (Color online) Comparison of Solution Quality Using Randomly Generated Initial Guesses, Quantum-Generated Samples, and Classically Refined Solutions



7. Conclusion and Future Work

QHDOPT is the first open-source software leveraging quantum devices for nonconvex nonlinear optimization problems, providing an accessible interface for domain experts without quantum computing knowledge. Exploiting the idea of Hamiltonian-oriented programming, it efficiently uses quantum devices by implementing the quantum Hamiltonian descent algorithm with the SimuQ framework. We demonstrated QHDOPT's effectiveness through examples and benchmarks, showing its advantage over classical solvers, especially in large, complex instances. However, the current limitations of quantum device programmability and scalability constrain our benchmarks' scale. Although QHD shows promise in solving complex optimization problems, further empirical studies are needed for real-world performance evaluation.

There are several avenues for future development of QHDOPT. First, it is desired to broaden the problem class that can be handled by QHDOPT. Currently, because of hardware limitations, QHDOPT supports only the optimization of box-constrained nonlinear problems defined as a sum of univariate and bivariate functions. We anticipate that, shortly, QHDOPT can be extended to address more complicated objective functions as quantum technology and quantum algorithm design continue to coevolve. Second, although local search algorithms work well to improve the precision of quantum-generated samples, to obtain a global optimality guarantee, it might be promising to replace the refinement/postprocessing subroutine in QHDOPT with global optimizers (e.g., those based on branch-and-bound). Third, with further progress on quantum engineering, QHDOPT is expected to support more quantum devices from different platforms, including commercial or laboratory devices, which is essential to understanding the advantage of QHDOPT given different combinations of embedding schemes and quantum devices. Last but not least, QHDOPT can be expanded into a plugin for various domain-specific tools, including those in engineering, management, finance, and economics. Adaptions to specific domains are invaluable for users to better utilize quantum devices for their domain problems. Our overarching goal is to establish a user-friendly tool, empowering individuals and organizations to harness the power of quantum devices to solve challenging problems in the real world.

Acknowledgments

Samuel Kushnir and Jiaqi Leng contributed equally to this work. Most of this work was completed at the University of Maryland.

Endnotes

[1] This paper is not intended to be a comprehensive tutorial or documentation on QHDOPT. Instead, we direct the readers to https://github.com/jiaqileng/QHDOPT for the source code, examples, tutorials, and documentation.

[2] It is generally impossible to combine a sum of products into a single product form. For example, we cannot find two univariate functions $p(x)$ and $q(y)$ such that $p(x)q(y) = \sin(x)y + xe^y$.

[3] The details of Hamming embedding can be found in Leng et al. (2023b, appendix F.3). Note that this embedding scheme is referred to as "Hamming encoding" in Leng et al. (2023b, p. 52).

[4] More precisely, the one-hot embedding we implemented in `QHDOPT` is referred to as "penalty-free one-hot" embedding in Leng et al. (2024, p. 11, Table 1).

[5] Detailed expressions of these test instances are provided in the software repository; see the "examples" folder.

[6] In Figure 7, we plot objective function values only for high-dimensional problems, that is, Test Instances 6–15.

References

Aleksandrowicz G, Alexander T, Barkoutsos P, Bello L, Ben-Haim Y, Bucher D, Cabrera-Hernández FJ, et al. (2019) Qiskit: An open-source framework for quantum computing. Accessed November 4, 2024, https://zenodo.org/records/2562111.

Amazon Web Services (2024) Amazon Braket Python SDK. https://github.com/amazon-braket/amazon-braket-sdk-python.

Atalaya J, Zhang S, Niu MY, Babakhani A, Chan H, Epstein JM, Whaley KB (2021) Continuous quantum error correction for evolution under time-dependent Hamiltonians. *Physica. Rev. A* 103(4):042406.

Attouch H, Goudou X, Redont P (2000) The heavy ball with friction method, I. The continuous dynamical system: Global exploration of the local minima of a real-valued function by asymptotic analysis of a dissipative dynamical system. *Comm. Contemporary Math.* 2(1):1–34.

Bergholm V, Izaac J, Schuld M, Gogolin C, Ahmed S, Ajith V, Alam MS, et al. (2018) PennyLane: Automatic differentiation of hybrid quantum-classical computations. Preprint, submitted November 12, https://arxiv.org/abs/1811.04968.

Beverland ME, Murali P, Troyer M, Svore KM, Hoeffler T, Kliuchnikov V, Low GH, Soeken M, Sundaram A, Vaschillo A (2022) Assessing requirements to scale to practical quantum advantage. Preprint, submitted November 14, https://arxiv.org/abs/2211.07629.

Bliek1ú C, Bonami P, Lodi A (2014) Solving mixed-integer quadratic programming problems with IBM-CPLEX: A progress report. *Proc. 26th RAMP Symposium (Hosei University, Tokyo)*.

Bluvstein D, Evered SJ, Geim AA, Li SH, Zhou H, Manovitz T, Ebadi S, et al. (2024) Logical quantum processor based on reconfigurable atom arrays. *Nature* 626(7997):58–65.

Childs AM, Leng J, Li T, Liu JP, Zhang C (2022) Quantum simulation of real-space dynamics. *Quantum* 6:860.

D-Wave Quantum Systems Inc. (2023) D-Wave Ocean SDK. https://github.com/dwavesystems/dwave-ocean-sdk.

Dalzell AM, Clader BD, Salton G, Berta M, Lin CYY, Bader DA, Stamatopoulos N, et al. (2023) End-to-end resource analysis for quantum interior-point methods and portfolio optimization. *PRX Quantum* 4(4):040325.

Farhi E, Goldstone J, Gutmann S (2014) A quantum approximate optimization algorithm. Preprint, submitted November 14, https://arxiv.org/abs/1411.4028.

Frostig R, Johnson MJ, Leary C (2018) Compiling machine learning programs via high-level tracing. *SysML Conf. 2018 (Stanford, CA)*.

Google Quantum AI (2023) Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614(7949):676–681.

Gurobi Optimization, LLC (2021) Gurobi optimizer reference manual. Accessed November 4, 2024, https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/11.0/refman.pdf.

Hochbaum DS (2007) Complexity and algorithms for nonlinear optimization problems. *Ann. Oper. Res.* 153:257–296.

Iosue JT (2022) Qubovert. https://github.com/jtiosue/qubovert.

Johansson JR, Nation PD, Nori F (2012) QuTiP: An open-source Python framework for the dynamics of open quantum systems. *Comput. Phys. Comm.* 183(8):1760–1772.

Kawajir Y, Laird C, Wachter A (2006) Introduction to Ipopt: A tutorial for downloading, installing, and using Ipopt. Technical report, Carnegie Mellon University, Pittsburgh.

Khosravi F, Yildiz U, Scherer A, Ronagh P (2022) Non-convex quadratic programming using coherent optical networks. Preprint, submitted September 9, https://arxiv.org/abs/2209.04415.

Kushnir S, Leng J, Peng Y, Fan L, Wu X (2024) QHDOPT: A software for nonlinear optimization with quantum Hamiltonian descent. http://dx.doi.org/10.1287/ijoc.2024.0587.cd, https://github.com/INFORMSJoC/2024.0587.

Leng J, Zheng Y, Wu X (2023a) A quantum-classical performance separation in nonconvex optimization. Preprint, submitted November 1, https://arxiv.org/abs/2311.00811.

Leng J, Hickman E, Li J, Wu X (2023b) Quantum Hamiltonian Descent. Preprint, submitted March 2, https://arxiv.org/abs/2303.01471.

Leng J, Li J, Peng Y, Wu X (2024) Expanding hardware-efficiently manipulable Hilbert space via Hamiltonian embedding. Preprint, submitted January 16, https://arxiv.org/abs/2401.08550.

Lloyd S, Slotine JJE (1998) Analog quantum error correction. *Phys. Rev. Lett.* 80(18):4088.

Matsuda Y (2020) Research and development of common software platform for Ising machines. *Proc. 2020 IEICE General Conf.* (Institute of Electronics, Information and Communication Engineers, Tokyo).

Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, et al. (2017) SymPy: Symbolic computing in Python. *PeerJ Comput. Sci.* 3:e103.

Morrell Z, Vuffray M, Misra S, Coffrin C (2024) QuantumAnnealing: A Julia package for simulating dynamics of transverse field ising models. Preprint, submitted April 22, https://arxiv.org/abs/2404.14501.

Nguyen MT, Liu JG, Wurtz J, Lukin MD, Wang ST, Pichler H (2023) Quantum optimization with arbitrary connectivity using Rydberg atom arrays. *PRX Quantum* 4(1):010316.

O'Brien JL, Furusawa A, Vučković J (2009) Photonic quantum technologies. *Nature Photonics* 3(12):687–695.

Peng Y, Young J, Liu P, Wu X (2024) SimuQ: A framework for programming quantum Hamiltonian simulation with analog compilation. Hicks M, ed. *Proc. ACM Programming Languages* (Association for Computing Machinery, New York), 2425–2455.

Polyak BT (1964) Some methods of speeding up the convergence of iteration methods. *USSR Comput. Math. Math. Phys.* 4(5):1–17.

QuEra Computing Inc. (2023) Quera Bloqade.jl. https://github.com/QuEraComputing/Bloqade.jl.

Quintero RA, Zuluaga LF (2023) QUBO formulations of combinatorial optimization problems for quantum computing devices. Pardalos PM, Prokopyev OA, eds. *Encyclopedia of Optimization* (Springer, Cham).

Saffman M (2016) Quantum computing with atomic qubits and Rydberg interactions: Progress and challenges. *J. Phys. B Atomic Molecular Optical Phys.* 49(20):202001.

Singh K, Bradley C, Anand S, Ramesh V, White R, Bernien H (2023) Mid-circuit correction of correlated phase errors using an array of spectator qubits. *Science* 380(6651):1265–1269.

Singhal K, Hietala K, Marshall S, Rand R (2022) Q# as a quantum algorithmic language. Preprint, submitted June 7, https://arxiv.org/abs/2206.03532.

Sivak V, Eickbusch A, Royer B, Singh S, Tsioutsios I, Ganjam S, Miano A, et al. (2023) Real-time quantum error correction beyond break-even. *Nature* 616(7955):50–55.

Su W, Boyd S, Candes EJ (2016) A differential equation for modeling Nesterov's accelerated gradient method: Theory and insights. *J. Machine Learn. Res.* 17(153):1–43.

The Cirq Developers (2019) Cirq. https://github.com/quantumlib/Cirq.

Verdon G, Arrazola JM, Brádler K, Killoran N (2019) A quantum approximate optimization algorithm for continuous problems. Preprint, submitted February 1, https://arxiv.org/abs/1902.00409.

Wendin G (2017) Quantum information processing with superconducting circuits: A review. *Rep. Progress Phys.* 80(10):106001.

Wibisono A, Wilson AC, Jordan MI (2016) A variational perspective on accelerated methods in optimization. *Proc. Natl. Acad. Sci. USA* 113(47):E7351–E7358.

Xavier PM, Ripper P, Andrade T, Garcia JD, Maculan N, Neira DEB (2023) QUBO.jl: A Julia ecosystem for quadratic unconstrained binary optimization. Preprint, submitted July 5, https://arxiv.org/abs/2307.02577.

Yamamoto Y, Aihara K, Leleu T, Kawarabayashi Ki, Kako S, Fejer M, Inoue K, Takesue H (2017) Coherent Ising machines—Optical neural networks operating at the quantum limit. *npj Quantum Inform.* 3(1):49.

Zaman M, Tanahashi K, Tanaka S (2021) PyQUBO: Python library for mapping combinatorial optimization problems to QUBO form. *IEEE Trans. Comput.* 71(4):838–850.