

QUBO.jl: A Julia Ecosystem for Quadratic Unconstrained Binary Optimization

Pedro Maciel Xavier[◊], Pedro Ripper^{*}, Tiago Andrade, Joaquim Dias Garcia

PSR, {pedroxavier,pedroripper,tiago.andrade,joaquim}@psr-inc.com

[◊]PESC - COPPE - Federal University of Rio de Janeiro

^{*}DEE - Pontifical Catholic University of Rio de Janeiro

Nelson Maculan

Federal University of Rio de Janeiro, maculan@cos.ufrj.br

David E. Bernal Neira

Research Institute of Advanced Computer Science, Universities Space Research Association

Quantum Artificial Intelligence Laboratory, NASA Ames Research Center

Davidson School of Chemical Engineering, Purdue University, dbernaln@purdue.edu

We present `QUBO.jl`, an end-to-end Julia package for working with QUBO (Quadratic Unconstrained Binary Optimization) instances. This tool aims to convert a broad range of JuMP problems for straightforward application in many physics and physics-inspired solution methods whose standard optimization form is equivalent to the QUBO. These methods include quantum annealing, quantum gate-circuit optimization algorithms (Quantum Optimization Alternating Ansatz, Variational Quantum Eigensolver), other hardware-accelerated platforms, such as Coherent Ising Machines and Simulated Bifurcation Machines, and more traditional methods such as simulated annealing. Besides working with reformulations, `QUBO.jl` allows its users to interface with the aforementioned hardware, sending QUBO models in various file formats and retrieving results for subsequent analysis. `QUBO.jl` was written as a JuMP / MathOptInterface (MOI) layer that automatically maps between the input and output frames, thus providing a smooth modeling experience.

Key words: QUBO, Quantum Computing, JuMP, Ising Solvers

1. Introduction

Over the past two decades, the Quadratic Unconstrained Binary Optimization (QUBO) mathematical programming framework grew in popularity and relevance across many research fields such as finance (Orús et al. 2019, Herman et al. 2023), chemistry (Cao et al. 2019), engineering (Bernal Neira et al. 2022), and others (Biamonte et al. 2017, Rieffel et al. 2019). This increasing attention trend arose primarily due to steep improvements in computing paradigms whose standard task is to sample high-quality solutions for this kind of problem. Apart from that, QUBO formulations are now known for being well-suited when

representing many non-convex global optimization problems, especially those of combinatorial and discrete nature (Orús et al. 2019, Rieffel et al. 2019, Bernal Neira et al. 2022).

In this regard, quantum computers stand out as one of the main platforms for running QUBO formulations. Although not presenting real-world application results yet, Quantum Computing (QC) has been gaining much traction, backed by investments from the public and private sectors interested in the expected benefits of using quantum computers (Deshpande 2022).

From this standpoint, the current state of QC, known as the NISQ Era (Preskill 2018) is characterized by big-tech companies, startups, and national institutes improving and developing quantum systems, which are currently susceptible to noisy operations and decoherence (Deshpande 2022).

However, in a few years, QC is predicted to show some practical results in areas such as Optimization (Preskill 2018). Moreover, interesting proofs-of-concept have arisen, highlighting the potential of quantum technologies for addressing problems of this form (Tasseff et al. 2022). This advent will increase the demand for a trained workforce to program these quantum computers due to the fact that Operations Research specialists usually lack the required background. As a consequence, the capacity to adapt to this new scenario will be key to thriving with this new framework of computation, making companies and research institutes spend time and resources to stay on track with the technology.

For this undertaking, one challenge is to be able to reformulate optimization models into a format amenable to Quantum Computing. Furthermore, Operations Research specialists must be capable of operating on these new architectures, where QUBO arises as of their standard optimization format, although presenting different interfaces. Additionally, it will be crucial to transform raw results from different hardware into useful information to be analyzed.

To address these issues, we have developed the `QUBO.jl` package (Xavier et al. 2023b), an all-in-one tool for working with QUBO problems. This software works as a wrapper for three other packages. The first one, `ToQUBO` (Xavier et al. 2022) handles reformulation of general optimization problems into the QUBO form. The second, `QUBODrivers.jl` (Xavier et al. 2023a) is responsible for hardware interfacing, for sending QUBOs and analyzing QUBO inputs and generated solution pools. Finally, the third one, `QUBOTools.jl` (Xavier et al.

2023c) was envisioned to handle QUBO file conversions from different architectures and provide some utilities for the other two packages, such as graphing recipes.

Having all three packages combined into `QUBO.jl`, the user will be able to evaluate the potential of quantum and hardware-accelerated computers when addressing their problems without prior knowledge of these systems, providing a smooth introduction to optimization via hardware accelerators, such as quantum computers.

In this work, we will present `QUBO.jl`'s packages separately. This paper is organized as follows. Section 3 briefly discusses the adopted reformulation methods for generating QUBO problems from nonlinear Mixed-integer optimization problems. Section 3 follows with a general explanation of our reformulation layer, which is responsible for providing a smooth modeling experience to the user who is familiar with the `Julia` mathematical programming modeling package, `JuMP` (Lubin et al. 2023). The proposed QUBO sampler interface for `JuMP`, which is implemented in the `QUBODrivers.jl` module, is presented next in Section 4. Next, Section 5 briefly introduces our package for handling different QUBO file formats, `QUBOTOOLS.jl`. The current state of quantum software packages is analyzed in Section 6, while Section 7 sets a benchmark with other existing projects, and Section 8 points out conclusions and next steps in the development of this project. We also provide in Appendix A a discussion on integer-to-binary encoding comparison, while our new encoding method is presented in Appendix B. Finally, we provide details on our benchmarking environment in Appendix C.

1.1. Quadratic Unconstrained Binary Optimization (QUBO)

An optimization problem is in the QUBO form if it is written as:

$$(\text{QUBO}) : \min_{\mathbf{x} \in \mathbb{B}^n} \mathbf{x}' \mathbf{Q} \mathbf{x} = \min_{[x_1, \dots, x_n] \in \mathbb{B}^n} 2 \sum_{i=1}^n \sum_{j=i+1}^n q_{i,j} x_i x_j + \sum_{i=1}^n q_{i,i} x_i \quad (1)$$

where $\mathbf{Q} \in \mathbb{R}^{n \times n}$ is a symmetric matrix with elements $q_{i,j}$. When \mathbf{Q} is not diagonal, solving this problem is known to be, in general, NP-Hard (Pardalos and Jha 1992).

1.2. Ising Model

An optimization problem is in the Ising form if it is written as:

$$(\text{Ising}) : \min_{\mathbf{s} \in \{\pm 1\}^n} \mathbf{s}' \mathbf{J} \mathbf{s} + \mathbf{h}' \mathbf{s} = \min_{[s_1, \dots, s_n] \in \{\pm 1\}^n} \sum_{i=1}^n \sum_{j=i+1}^n J_{i,j} s_i s_j + \sum_{i=1}^n h_i s_i \quad (2)$$

where $\mathbf{J} \in \mathbb{R}^{n \times n}$ is strictly upper triangular and $\mathbf{h} \in \mathbb{R}^n$.

There is a bijective relation between Ising and QUBO models, where one could easily encode variables from one model to the other as:

$$\begin{aligned}\mathbf{s} &= 2\mathbf{x} - \mathbf{1} \\ \mathbf{x} &= \frac{1}{2}(\mathbf{s} + \mathbf{1})\end{aligned}$$

1.3. Mixed-Integer Nonlinear Programming

An optimization problem is in the Mixed-Integer Nonlinear Programming (MINLP) form if it can be written as:

$$\begin{aligned}(\text{MINLP}) : \min_{\mathbf{y}, \mathbf{z}} & f(\mathbf{y}, \mathbf{z}) \\ \text{s.t.} & \mathbf{g}(\mathbf{y}, \mathbf{z}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{y}, \mathbf{z}) = \mathbf{0} \\ & \mathbf{y} \in \mathbb{R}^n; \mathbf{z} \in \mathbb{Z}^m\end{aligned}\tag{3}$$

where $f : \mathbb{R}^n \times \mathbb{Z}^m \rightarrow \mathbb{R}$ denotes the objective function, which by convention is defined to be minimized although $\min_{\mathbf{y}, \mathbf{z}} f(\mathbf{y}, \mathbf{z}) = -\max_{\mathbf{y}, \mathbf{z}} -f(\mathbf{y}, \mathbf{z})$, \mathbf{y} and \mathbf{z} denote continuous and integer variables, respectively, and \mathbf{g} and \mathbf{h} denote inequality and equality vector constraints, respectively.

Notice that we do not make any assumption on the convexity or linearity of the functions $f, \mathbf{g}, \mathbf{h}$, making MINLP a very flexible modeling paradigm. The possibility of modeling linear and nonlinear constraints together with discrete and continuous variables makes it able to represent Universal Turing Machines (Liberti and Marinelli 2014), and has many applications in science and engineering (Belotti et al. 2013, Kronqvist et al. 2019).

We are particularly interested when the nonlinear objective and constraints can be written as quadratics, i.e., $f(\mathbf{y}, \mathbf{z}) = [\mathbf{y}; \mathbf{z}]' \mathbf{Q} [\mathbf{y}; \mathbf{z}] + \ell' [\mathbf{y}; \mathbf{z}] + c$. A problem with both quadratic constraints and quadratic objective is known as Mixed-Integer Quadratically Constrained Quadratic Program (MIQCQP), while the case with linear constraints, i.e., $\mathbf{h}(\mathbf{y}, \mathbf{z}) = A[\mathbf{y}; \mathbf{z}] - \mathbf{b}$, and quadratic objective is known as Mixed-integer Quadratic Program (MIQP).

$$\begin{aligned}(\text{MIQP}) : \min_{\mathbf{y}, \mathbf{z}} & [\mathbf{y}; \mathbf{z}]' \mathbf{Q} [\mathbf{y}; \mathbf{z}] + \ell' [\mathbf{y}; \mathbf{z}] + c \\ \text{s.t.} & A[\mathbf{y}; \mathbf{z}] = \mathbf{b} \\ & \mathbf{y} \in \mathbb{R}^n; \mathbf{z} \in \mathbb{Z}^m\end{aligned}\tag{4}$$

where $\mathbf{Q} \in \mathbb{R}^{(n+m) \times (n+m)}$ is symmetric, $\ell \in \mathbb{R}^{(n+m)}$, $c \in \mathbb{R}$, $A \in \mathbb{R}^{(n+m) \times p}$, $\mathbf{b} \in \mathbb{R}^p$.

MIQP models are notorious for encompassing both discrete and simple nonlinear behavior. They are widely used in areas such as Finance, Economics, Computer Science, Operations Research, and Engineering (Misener and Floudas 2012). Every QUBO can also be seen as a MIQP since $\mathbb{B} \subset \mathbb{Z}$. This aspect allows one to use commercial MIQP solvers, such as Xpress (FICO 2023) and CPLEX (Cplex 2009), to work with QUBO instances. It is worth mentioning that some optimization software, e.g., Gurobi (Gurobi Optimization, LLC 2023) and SCIP (Rehfeldt et al. 2023), have released specific routines to handle QUBO models. On the other hand, it is possible to approximate MIQP problems (Liberti 2009) through QUBO modeling by applying the variable certain encoding and constraint mapping methods, as discussed in Section 3.

2. QUBO.jl ecosystem

QUBO.jl was developed to become a bridge between Operations Research specialists and quantum or quantum-inspired optimization. Having said that, wrapping the previously mentioned packages, ToQUBO.jl, QUBODrivers.jl and QUBOTools.jl, it turns into a self-contained ecosystem that can cover most tasks related to QUBO and Ising optimization. From the diagram depicted in Figure 1, one can better understand how QUBO.jl works and how its packages communicate.

First, ToQUBO.jl acts as an interface for JuMP users to generate QUBO instances. As they send their JuMP model to ToQUBO.jl, after reformulating, it caches the QUBO problem as a MOI model. Then, this MOI model can either be sent to a general classical MIQP solver or, as we have labeled, a QUBO solver, that can be a classical, quantum, or quantum-inspired hardware.

In order for a QUBO model, generated by ToQUBO.jl, to be sent to a QUBO solver, we have developed QUBODrivers.jl, where users can define an interface to access different QUBO-amenable solvers. After sending a QUBO to one of these solvers, QUBODrivers.jl retrieves the results for the user, which can be later evaluated by some of its own analysis tools or sent to QUBOTools.jl, which provides plotting recipes for a visual representation of the results.

As some emerging architectures require QUBOs to be sent in files, another feature from QUBOTools.jl is file-conversion handling. This package allows format conversion between different file formats, considering that each hardware demands a specific type.

`QUBOTools.jl` also provides translating MOI QUBO models into a file and vice-versa, presenting a bidirectional conversion between any of the envisioned QUBO model representations.

To summarize, `QUBO.jl` users can effortlessly harness quantum and quantum-inspired optimization methods to sample solutions for any general optimization problem. Therefore, integrating our package into the work of Operation Research specialists would be a frictionless experience, requiring no previous knowledge of quantum technologies.

3. QUBO Compilation

`ToQUBO.jl`'s main goal is to translate JuMP models into the QUBO form, aiming at their submission to solution sampling architectures, such as quantum annealers. To illustrate the proposed pipeline, it might be worth building an analogy by pretending that a QUBO instance is equivalent to “assembly code” when dealing with some of the optimization machines presented above. Within this picture, `ToQUBO.jl` acts as a higher-level language compiler, allowing more general JuMP models to be used as input for solvers that consider the QUBO formalism. Pretty much as `gcc` (Stallman and Weinberg 1987) generates Assembly code from programs written in C, `ToQUBO.jl` is capable of producing optimization programs as a form of lowered code for specialized hardware.

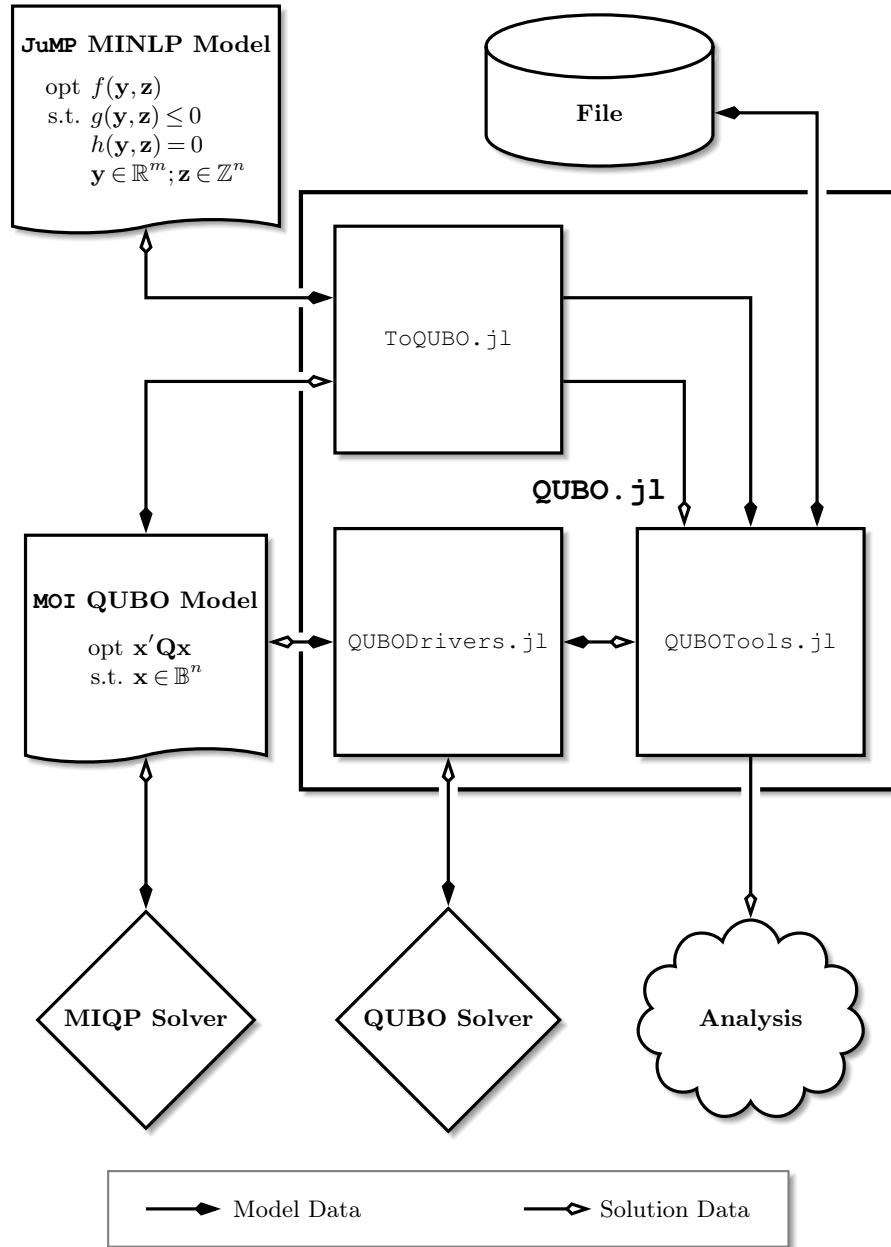
Conceptually speaking, each characteristic of the target model introduces one or more specific steps to the reformulation process. The resulting model must only have binary variables, have no constraints, and be represented by a polynomial whose degree is at most 2.

We also consider an intermediate representation of the problem based on pseudo-Boolean functions (Boros and Hammer 2002), that is, real polynomials on binary variables of the form

$$f(\mathbf{x}) = \sum_{\omega \in \mathcal{P}([n])} c_{\omega} \prod_{j \in \omega} x_j \quad (5)$$

where $\mathcal{P}([n])$ is the power set of the set $[n] = \{1, \dots, n\}$ and $f : \{0, 1\}^n \rightarrow \mathbb{R}$ with $c_{\omega} \in \mathbb{R}$. Let \mathcal{F} be the family of all pseudo-Boolean functions and \mathcal{F}^k its subset containing only polynomials of degree k or less, i.e., $\mathcal{F}^k = \{f : f \in \mathcal{F}, \deg f \leq k\}$. Working with these mathematical objects is a natural choice because, apart from a constant term c_{\emptyset} , optimization of \mathcal{F}^2 over binary variables is equivalent to QUBO.

Figure 1 Diagram of the `QUBO.jl` ecosystem. `ToQUBO.jl` acts as an interface for QUBO reformulation within the `JuMP` framework. The QUBO model generated by `ToQUBO.jl`, or explicitly defined in a `JuMP` problem or a file can later be sent to QUBO solvers with `QUBODrivers.jl`. Finally, `QUBOTools.jl` provides tools for result and model analysis.



Variable Encoding

When attempting to run constrained or non-binary programs on QUBO-amenable solvers, one must be able to encode the problem's variables and functions as a QUBO instance, which is described by binary variables. The conversion from non-binary to binary variables is the first step in the conversion process. In some cases, the reformulation is exact, for

example, when converting bounded integer variables to binary. Real variables, however, must go through discretization before the aforementioned transformation.

There are currently some well-known encoding techniques that are being used in the development of QUBO reformulations. The most prominent are Binary (Tamura et al. 2021), Unary (Tamura et al. 2021), One-Hot (Tamura et al. 2021), Domain-wall (Chancellor 2019) and Bounded-coefficient (Karimi and Ronagh 2019).

Additionally, we introduce with `ToQUBO.jl` a novel method by the name Arithmetic Progression (AP) encoding, see Appendix B. Each representation has its own characteristics, e.g., the number of binary variables (bits) required, the number of terms in the polynomial expansion produced, and the maximum absolute value Δ of its coefficients, which is known to impact the quality of the resulting formulation (Glover et al. 2019). An asymptotic comparison is presented in Table 1, suggesting that the AP encoding paradigm mitigates the growth of the expansion coefficients while keeping the demand for new variables sublinear.

Table 1 Comparison between methods for encoding values for $x \in \{0, \dots, n\}$ into binary variables in `ToQUBO.jl`

Encoding Method	Binary Variables	# of terms		Δ
		Linear	Quadratic	
Binary ¹	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	-	$\mathcal{O}(n)$
Unary ¹	$\mathcal{O}(n)$	$\mathcal{O}(n)$	-	$\mathcal{O}(1)$
One-Hot ¹	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Domain-Wall ²	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Bounded-Coefficient ³	$\mathcal{O}(n)$	$\mathcal{O}(n)$	-	$\mathcal{O}(1)$
Arithmetic Progression	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(\sqrt{n})$	-	$\mathcal{O}(\sqrt{n})$

¹ (Tamura et al. 2021); ² (Chancellor 2019); ³ (Karimi and Ronagh 2019);

Penalty Mechanism

Representing constraints in an unconstrained problem is done by adding penalty terms to the objective function of the final problem. Constraints can be expressed as functions belonging to feasibility sets, e.g., $g_i(\mathbf{x}) \leq 0 \iff g_i(\mathbf{x}) \in S_i, S_i = (-\infty, 0]$. Every constraint $g_i(\mathbf{x}) \in S_i$ is translated into a penalty function $\|g_i(\mathbf{x})\|_{S_i}$ with its corresponding penalty factor ρ_i . Under minimization, for example, the accurate portrayal of the feasible set will require each penalty function to be positive if its constraint is violated and zero otherwise.

$$\begin{aligned} \min_{\mathbf{x}} f(\mathbf{x}) \\ \text{s.t. } g_i(\mathbf{x}) \in S_i \ \forall i \end{aligned} \xrightarrow{\text{Penalization}} \min_{\mathbf{x}} f(\mathbf{x}) + \sum_i \rho_i \|g_i(\mathbf{x})\|_{S_i} \quad (6)$$

Each penalty factor ρ_i should be large enough to ensure that constraints are respected under overall optimality (Glover et al. 2019, Lucas 2014). On the other hand, if the chosen coefficients are too large, they might cause a negative impact on the conditioning of the resulting expression, as discussed in Appendix A.

`ToQUBO.jl` implements the mapping and penalization of several constraint types, which is depicted in Section 7. A common method for embedding linear equality constraints, for instance, will introduce a quadratic expression into the objective function as $A\mathbf{x} = \mathbf{b}$ becomes $(A\mathbf{x} - \mathbf{b})'(A\mathbf{x} - \mathbf{b})$. However, when dealing with other families of constraints, higher-order penalty functions might arise and an additional degree-reduction step known as *quadratzation* is required for them to fit in the QUBO formalism.

Quadratzation

A *quadratzation* is a mapping $\mathcal{Q}: \mathcal{F} \rightarrow \mathcal{F}^2$ such that

$$\forall f \in \mathcal{F}, \forall \mathbf{x} \in \{0, 1\}^n, \min_{\mathbf{w}} \mathcal{Q}\{f\}(\mathbf{x}; \mathbf{w}) = f(\mathbf{x}) \quad (7)$$

where $\mathbf{w} \in \{0, 1\}^m$ is a vector of auxiliary variables.

There are many possible quadratzation methods for writing QUBO models from higher-degree pseudo-Boolean functions (Dattani 2019). By leveraging `Julia`'s multiple dispatch paradigm, `ToQUBO.jl` allows its users to extend the quadratzation interface by implementing their own degree reduction algorithms. For more details on various quadratzation schemes, refer to (Dattani 2019). Currently, two single-term quadratzation techniques have already been implemented: one for negative and the other for positive terms.

The standard reduction method for negative terms, labeled as NTR-KZFD, was introduced by Kolmogorov and Zabih (Kolmogorov and Zabin 2004), and later by Freedman and Drineas (Freedman and Drineas 2005). It reduces a single term, introducing a single auxiliary variable w , as follows.

$$-x_1 x_2 \cdots x_k \xrightarrow{\text{NTR-KZFD}} (k-1)w - \sum_i x_i w \quad (8)$$

Moreover, the default positive term quadratzation technique, named PTR-BG, was developed by Boros and Grubner (Boros and Gruber 2014). It is also focused on single terms with

a degree higher than two and works with $k - 2$ extra variables w_i , where k is the number of variables in the original high-order term, as presented below

$$x_1 x_2 \cdots x_k \xrightarrow{\text{PTR-BG}} \left[\sum_{i=1}^{k-2} w_i \left(k - i - 1 + x_i + \sum_{j=i+1}^k x_j \right) \right] + x_{k-1} x_k \quad (9)$$

Reformulation Layer

`ToQUBO.jl` implements a mathematical program reformulation layer to provide a transparent interface for the user to model, optimize, and collect results from QUBO sampling runs. The source program is written as a regular JuMP model, so all the data is cached in the structures defined in `MathOptInterface` (Legat et al. 2021) to represent variables, constraints, objectives, and additional attributes. Starting from the well-defined mathematical program formulation stored in the `MathOptInterface` (MOI) structure, `ToQUBO.jl` applies the procedures described in Section 3 to convert the original problem into a QUBO form that is cached in a new MOI model. Since the new model is a QUBO defined in MOI, it can be directly forwarded to any solver implementing the interface and supporting the required features, i.e., binary variables and quadratic objectives. Therefore we can solve the problem with QUBO sampling machines or even a regular Mixed Integer Quadratic Programming (MIQP) solver capable of handling non-convex objectives, as long as they have an MOI wrapper. In all these conversions and forwarding steps, relationship maps are kept in memory so that the results from the optimization can be queried by the user. This process is depicted in Figure 1.

The capability of creating such layers is an outstanding feature of JuMP and MOI and was a key motivation for selecting this ecosystem to develop a QUBO reformulator. Previous examples of this strategy are `Dualization.jl` (Bodin et al. 2021), which gets a JuMP model and provides to the solver its dual, `QuadraticToBinary.jl` (Dias Garcia 2021), which converts quadratic constraints into linear constraints with binary variables, and `DisjunctiveProgramming.jl` (Perez et al. 2023), which allows the formulation of disjunctive programs and then automatically converts them into MIPs or MINLPs.

4. Solver Interface

`QUBODrivers.jl` provides a common interface for developing bindings that bring QUBO annealing and sampling platforms to the JuMP environment. By sub-typing `MathOptInterface`’s `AbstractOptimizer`, the `AbstractSampler` standard grants the user many

relevant features, e.g., querying and sorting multiple results, simple QUBO model format validation, and adjustable attributes for fine-tuning runtime execution. By employing `QUBODrivers.jl`, the user is able to define a new MOI-compliant solver interface, thus providing access to different hardware architectures and sampling algorithms with ease.

As presented in Listing 1, everything begins with the `@setup` macro, whose body contains all necessary settings to specify a new optimizer. When declaring such properties, one should first fill in the solver’s name, sense, domain, and version, followed by the solver-specific parameters, which belong to the `attribute` block. This macro leverages Julia’s meta-programming capabilities to circumvent a considerable amount of repetitive code and isolate the user from most of MOI’s internal details.

By overloading the `sample` function, the QUBO model parsed from JuMP is then used to sample and communicate back solutions from an arbitrary underlying procedure. The return of this function must be a `SampleSet`, a special collection designed to provide fast queries and serve as input for analytical tools.

Listing 1: Example of `QUBODrivers.jl` usage to wrap QUBO solvers

```

module RandomSampler

import Random
import MathOptInterface as MOI
import QUBODrivers, QUBOTools

QUBODrivers.@setup Optimizer begin
    name = "Random Sampler"
    sense = :min
    domain = :bool
    version = v"1.0.0"
    attributes = begin
        RandomSeed["seed"]::Union{Integer,Nothing} = nothing
        NumberOfReads["num_reads"]::Integer        = 1_000
    end
end

function QUBODrivers.sample(sampler::Optimizer{T}) where {T}
    # Retrieve Raw Model
    Q, a, b = QUBODrivers.qubo(sampler, Dict)

    # Retrieve User-Defined Attributes
    n          = MOI.get(sampler, MOI.NumberOfVariables())
    num_reads  = MOI.get(sampler, RandomSampler.NumberOfReads())
    seed       = MOI.get(sampler, RandomSampler.RandomSeed())

    Random.seed!(seed)

    # Sample Random States
    samples = QUBOTools.Sample{T,Int}[]

    for _ in 1:num_reads

```

```

        x = Random.rand{0, 1}, n)
        y = QUBOTools.value(Q, x, a, b)

        push!(samples, QUBOTools.Sample{T,Int}(x, y))
    end

    # Return Solution
    return QUBOTools.SampleSet{T,Int}(samples)
end

end # module

```

5. Tools for QUBO

Last but not least, we present `QUBOTools.jl`, a library containing interface definitions and core functionality for managing and analyzing both QUBO models and their solutions. Its development is focused on providing

- Fast and reliable I/O that allows conversion between well-known file formats for QUBO and Ising models
- Generic modeling back-end utilities for powering other applications, including specialized reference implementations for conceptual data structures
- Analytical framework equipped with data queries, conditioning, and performance evaluation metrics and plot recipes
- Routines for random instance generation (see Section 8.1 - Next Steps)

5.1. File Formats

As discussed before, Quantum Operations Research has many agents, including hardware manufacturers, software solution providers, and mathematical optimization experts. An ecosystem as diverse and dynamic as this also has the drawback of having many different file formats and interfaces for specifying and running QUBO models. To address this issue, `QUBOTools.jl` provides an extensible I/O library with built-in *codecs*, making it a file converter compatible with the most used formats.

The *bqpjson* variant, for instance, is a platform-agnostic JSON specification developed by Los Alamos National Laboratory’s Advanced Network Science Initiative to embrace additional aspects of the problem, including related solutions and metadata from the target hardware (Coffrin 2020). Another example, the *QUBO* format, is defined according to D-Wave’s applications for quantum and simulated annealing but is also configurable to establish communication with MQLib. The *Qubist* type, also standardized by D-Wave, is one of the

simplest file standards available, with no support for metadata entries or any other sophisticated feature.

Yet, once able to consistently interpret *Qubist* and a few other file formats, it is possible to gain instant access to many problem databases produced by a wide range of groups (Hen 2019, Kowalsky et al. 2022). Conversely, being capable of writing in several QUBO dialects allows one to leverage the most prominent solvers available when building new applications. By operating in both ways, `QUBOTools.jl` establishes the technical foundation for performing intricate tasks such as benchmarking heterogeneous hardware and formulation analysis.

5.2. Data Structures

A project like `QUBO.jl` spans multiple activities within what could be called the "application layer" of Quantum Optimization Systems. It plays the role of a user-friendly platform by exposing high-level methods while also internally operating closer to the bare metal, e.g., accounting for hardware specifics when needed. Considering the different model descriptions and solver architectures, `QUBOTools.jl` defines a blueprint to coordinate the integration of `QUBO.jl`'s components.

Harnessing `Julia`'s powerful type system, a manifold of generic methods is delineated over a few conceptual data structures. This abstract interface puts the overall design of the ecosystem into perspective, focusing on a functional configuration rather than a static set of tools. With that said, `QUBOTools.jl` delivers a protocol for working with models, solvers, and solutions.

Moreover, each envisioned data structure has a self-contained, ready-to-use reference implementation. The main goal of this approach is to offer the building blocks for other applications, including `QUBO.jl` itself. By exploiting both abstract and concrete functionalities, one might be able to specialize portions of software behavior, either by extending the proposed interface or by the composition of its main artifacts.

5.3. Analytical Tools

Contemplating the variety of quantum hardware in the NISQ Era, which candidates will prevail as the field moves towards Fault-Tolerant QC, is still unknown. Additionally, each device can better perform with a specific QUBO formulation, factoring in variable encoding, constraint mapping, and quadratization techniques. Under these circumstances, benchmarking NISQ systems and QUBO models in the context of Quantum Optimization provides a guiding framework to assess the current state of these emerging solvers.

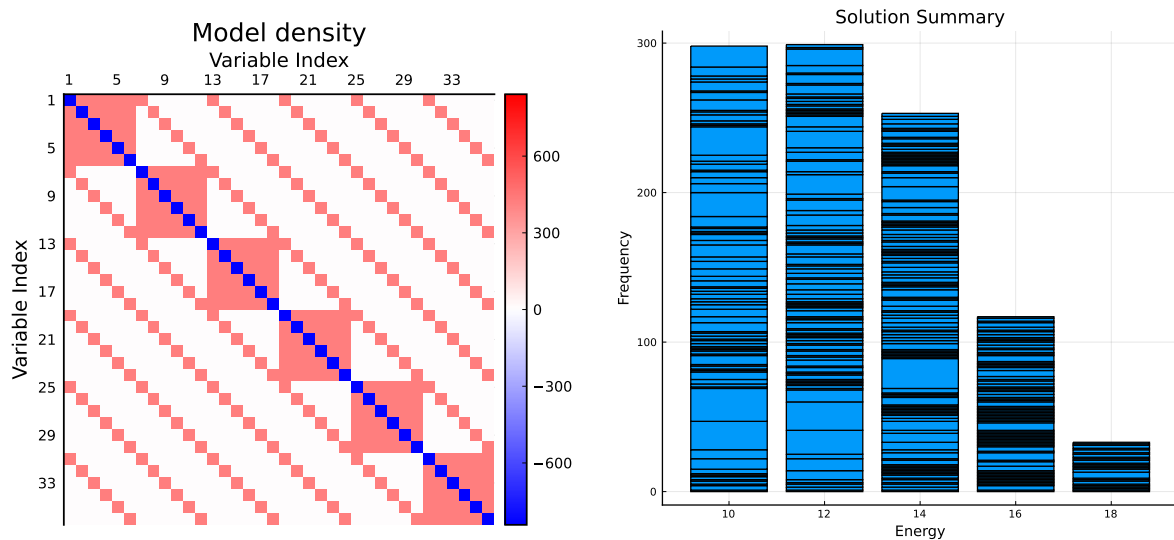
To address this concern, analytical tools were bundled in the package’s library, through which one can probe not just the conditioning and structure of the model but also the quality of the sampled solutions. When drawing comparisons between solvers, `QUBOTools.jl` can gauge time-related indicators, inspect the success rate from a set of runs, and evaluate composite metrics such as the time-to-solution (TTS) (King et al. 2015). Also, in light of model conditioning, one can withdraw statistics from the formulation amongst model density and the connectivity arising from its quadratic relations. The latter could also be useful for choosing between reformulation techniques in the context of `ToQUBO.jl`.

Visualization To enhance the data analysis toolkit, `QUBOTools.jl` counts with a set of predefined visualization schemes. It leverages `Plots.jl`’s (Breloff 2023) submodule `RecipesBase.jl` to define custom plot instructions, considering the different perspectives through which a model or solver can be visually examined. These recipes are a series of drawing statements and are not tied to any specific plotting library, thus considerably reducing the overhead by delaying or avoiding the installation of extra dependencies.

By design, `QUBOTools.jl` aims to cover the most commonly employed graphing paradigms in Quantum Optimization analysis. As of today, plotting recipes for examining the outcomes’ sampling distribution and the model’s matrix density have already been implemented. On its left side, Figure 2 depicts an example of a Traveling Salesperson Problem (TSP) instance, where the color saturation is proportional to the magnitude of each coefficient. On the right side, samples from running the TSP model are sorted by their respective objective values, and the number of reads from each configuration is given by their heights. Results were gathered using the NASA Parallel Tempering open-source solver `PySA` (Mandra and Katzgraber 2018), and samples from different states but similar energy values were stacked together.

Both conditioning metrics and visualization recipes were built based on the aforementioned abstract data structure layout. A consistent set of methods for querying models and solutions is one of the fundamental building blocks of the entire `QUBO.jl` software stack. By wrapping `JuMP` models and their solution sets with constructs that adhere to its interface, `QUBO.jl` allows users to enjoy the presented analytical toolbox seamlessly.

Figure 2 `QUBOTools.jl`'s plot recipes in action: model density (left), from a TSP instance with 6 cities (36 variables) and sampling frequency by energy value (right), using the `PySA` solver.



6. The State of Software for Quantum Computing

Evaluating the current scenario for software packages for Quantum Computing, it is noticeable that Python is the most popular language for their development. Some popular examples are IBM's Qiskit (Treinish et al. 2022), Google's Cirq (Developers 2022), Amazon's Braket SDK (Amazon Web Services 2020) and D-Wave's Ocean SDK (D-Wave 2022). It is also important to mention Microsoft's standalone Q# language (Singhal et al. 2022), to build programs on their cloud services, and Zapata's Orquestra platform (Zapata Computing Inc. 2020) for developing algorithms to run on several quantum hardware from different companies.

As for libraries for quantum and quantum-inspired optimization that also handle QUBO reformulation, Amplify (Matsuda 2022), PyQUBO (Zaman et al. 2021) and quboverter (Iosue 2022) stand out as the most prominent. Besides, IBM's Qiskit (Treinish et al. 2022) and EntropicaLab's OpenQAOA (Sharma et al. 2022) provide QUBO reformulation for models created using docplex (IBM 2015) or their native optimization modules. All these are implemented in Python. Moreover, SATyrus (Lima et al. 2007) is a logic-oriented modeling platform capable of generating QUBO models from SAT-like problem descriptions. Its latest implementation, SATyrus III (Xavier and Lima 2022), served as an initial inspiration for some of the design choices behind `QUBO.jl`.

Other software packages that do not focus on QUBO reformulation, such as Qiskit (Treinish et al. 2022) and Xanadu's PennyLane (Bergholm et al. 2018), allow

users to build arbitrary quantum circuits. This is still relevant to this work, given that there are circuit-based algorithms for the solution of QUBO problems, such as the Quantum Approximate Optimization Algorithm (QAOA) (Farhi et al. 2014), later rebranded as Quantum Alternating Optimization Ansatz (Hadfield et al. 2019), and the Variational Quantum Eigensolver (VQE) (Peruzzo et al. 2014). QUBO problems can then be solved using these algorithms using gate-based quantum computers. Additionally, Fujitsu provides a cloud service for working with QUBO models in their Digital Annealers (Fujitsu 2021).

Even in light of this panorama, *Julia*’s ecosystem for QC has been gaining momentum, even having Amazon recently releasing a `Braket.jl` (Amazon Web Services 2022) version supporting communication with its QC environment, AWS Braket. In addition, QuEra, a quantum hardware company, which is also available on AWS Braket, has published `Bloqade.jl` (QuEra 2023), a software to interface with its systems in *Julia*. Los Alamos Advanced Network Science Initiative has also published the `QuantumAnnealing.jl` (Morrell and Coffrin 2022) package for simulation of quantum annealing algorithms. Other examples include the `Yao.jl` (Luo et al. 2019) framework for experimenting with quantum algorithms, `QuantumCircuitOpt.jl` (Nagarajan et al. 2021) for quantum circuit design optimization. `ToQ.jl` (O’Malley and Vesselinov 2016) is a library provided to interface with D-Wave’s quantum annealers, later continued as `ThreeQ.jl`.

As for Quantum Optimization, *Julia* provides important features on top of the above-mentioned growing environment, as it was designed for high-performance computing (Bezanon et al. 2017). Hence, it allows for the development of efficient algorithms and avoids the two-language problem that occurs when part of an interpreted language, e.g., *Python*, code must be rewritten in a compiled one, e.g., C++, when aiming for performance. This has been the case for `PyQUBO` (Zaman et al. 2021), where the core of the reformulator to QUBO problems had to be rewritten from *Python* to C++. Also, `JuMP` is a widely used library in the Operations Research and Optimization communities. Therefore, `JuMP` extensions are readily available for a large group of users.

Moreover, programming in *Julia* does not exclude one from using quantum computers, e.g., using `Qiskit` to access IBM’s hardware. Building *Python* package wrappers with tools such as `PythonCall` (Rowley 2022) allows exploring the benefits of *Julia* and the wide variety of quantum software packages in *Python*. This review of the software status for quantum computing motivated us to write our tools in *Julia*, leveraging the strengths of

the programming language and the existing package environment for operations research and quantum computing.

7. Comparison between existing tools

As mentioned in the previous section, there are a few available tools for working with QUBO formulations, and the most relevant lie within the Python environment, such as Amplify, PyQUBO, qubover, Qiskit and OpenQAOA. These modules differ mainly in their capabilities, such as Automatic Variable Encoding and Constraint Mapping coverage. From this perspective, we have listed these features for each tool in Table 2, showing that `ToQUBO.jl` has all types of encodings and constraints covered by the mentioned packages.

Table 2 List of supported variable encoding and constraint penalization methods across QUBO modeling platforms

	ToQUBO.jl	PyQUBO	qubover	Qiskit	OpenQAOA	Amplify
Automatic variable encoding methods implemented						
Binary ¹	■ ★	■	■			■ !
Unary ¹	■ ★	■	■	■	■	■ !
One-Hot ¹	■ ★	■				
Domain-Wall ²	■ ★	■				
Bounded-Coefficient ³	■ ★	■				
Arithmetic Progression	■ ★					■ !
Supported automatic constraint reformulation						
$\mathbf{a}'\mathbf{x} \leq b$	■		■	■	■	■
$\mathbf{a}'\mathbf{x} = b$	■		■	■	■	■
$\mathbf{x}'\mathbf{Q}\mathbf{x} + \mathbf{a}'\mathbf{x} \leq b$	■					■
$\mathbf{x}'\mathbf{Q}\mathbf{x} + \mathbf{a}'\mathbf{x} = b$	■					■
SOS1	■					
$\bigwedge_i x_i$ [†]			■			■
$\bigvee_i x_i$ [†]			■			■
$\bigoplus_i x_i$ [†]						■

¹ (Tamura et al. 2021); ² (Chancellor 2019); ³ (Karimi and Ronagh 2019);

[†] For logical constraints, it is assumed that $x_i \in \{0, 1\}$.

★ `ToQUBO.jl` is the only platform in the board whose automatic encoding routines are also applicable to continuous variables. Furthermore, its *Bounded-Coefficient* implementation allows the technique to be used not just with the *Binary* encoding, as proposed in the original paper, but also with the *Unary* and *Arithmetic Progression* modes.

! `Amplify` only supports these encoding methods when introducing integer slack variables for mapping inequality constraints. They are not applicable to regular variables, as their models only accept binary or spin sites so far.

As presented earlier, `QUBODrivers.jl` enables users to interface with QUBO samplers and define new solvers, which work as wrappers, gaining access to different hardware. As of today, `QUBODrivers.jl` provides three initial samplers: *ExactSampler*, *RandomSampler* and *IdentitySampler*.

In addition, we have already released several solver wrapper packages using `QUBODrivers.jl`: `DWave.jl` (Xavier and Ripper 2022a), `DWaveNeal.jl` (Xavier and Ripper 2023a), `MQLib.jl` (Xavier and Ripper 2023b), `QuantumAnnealingInterface.jl` (Xavier and Ripper 2022b), `QiskitOpt.jl` (Xavier and Ripper 2023c), `CIMOptimizer.jl` (Chen et al. 2022, Xavier 2023a), and `PySA.jl` (Mandra and Katzgraber 2018, Xavier 2023b). `DWave.jl` and `DWaveNeal.jl` were developed as D-Wave’s Quantum and Simulated Annealing interfaces, respectively. `MQLib.jl` is a wrapper for a heuristics library for QUBO problems (Dunning et al. 2018) and `QuantumAnnealingInterface.jl` is an interface for Los Alamos National Laboratory’s `QuantumAnnealing.jl` (Morrell and Coffrin 2022). Finally, `QiskitOpt.jl` allows users to send their JuMP instances to run VQE and QAOA algorithms.

On the other hand, `PyQUBO` and `qubover` only offer access to D-Wave’s Simulated and Quantum Annealing hardware as external samplers and do not allow users to define new solvers. Furthermore, besides having access to D-Wave’s systems, `Amplify` provides an interface with Fujitsu’s Digital Annealer, Toshiba’s Bifurcation Machine, Hitachi’s CMOS Annealing Machine, IBM’s gate-based computers, and other Simulated Annealing machines. Finally, `Qiskit` and `OpenQAOA` allow users to run QUBO formulations on QAOA, VQE, and some other Variational Quantum algorithms that can be executed in IBM’s hardware.

It is noticeable that `Amplify` and `QUBO.jl` interface with a similar number of solvers, hardware-wise, as mentioned in Section 4. However, by uplifting Julia’s composability and interoperability features, `QUBODrivers.jl` users can easily define new solver wrappers. As the `QUBO.jl` ecosystem gains momentum, it is expected that our sampler coverage increases, as it is an open-source, extensible environment designed for algorithm researchers, hardware manufacturers, and enthusiasts in general.

This comparison is presented in Table 3.

Moreover, another method to compare QUBO formulation packages is to analyze their execution time to build an actual QUBO model. We have decided to follow `PyQUBO`’s benchmarking framework, presented in its documentation, where many instances of the Traveling

Table 3 External QUBO-amenable samplers coverage for each package

Sampler	Hardware	<code>QUBODrivers.jl</code>	<code>PyQUBO</code>	<code>qubovert</code>	<code>Qiskit</code>	<code>OpenQAOA</code>	<code>Amplify</code>
Simulated Annealing	CPU/ GPU	■ ³	■	■			■
Quantum Annealing	QA ¹	■ ⁴	■		■ ⁹		■
Simulated Quantum Annealing	CPU/ GPU	■ ⁵					■
VQE	QGC ²	■ ⁶			■	■	
QAOA	QGC ²	■ ⁶			■	■	■
MQLib	CPU	■ ⁷					
Heuristics Library	CPU	■ ⁸					
Parallel Tempering	CPU	■ ⁸					
Simulated Bifurcation Machine	GPU/FPGA						■
Simulated Coherent Ising Machine	CPU / GPU	■ ⁹					
Digital Annealing	ASIC						■
CMOS Annealing	CMOS						■

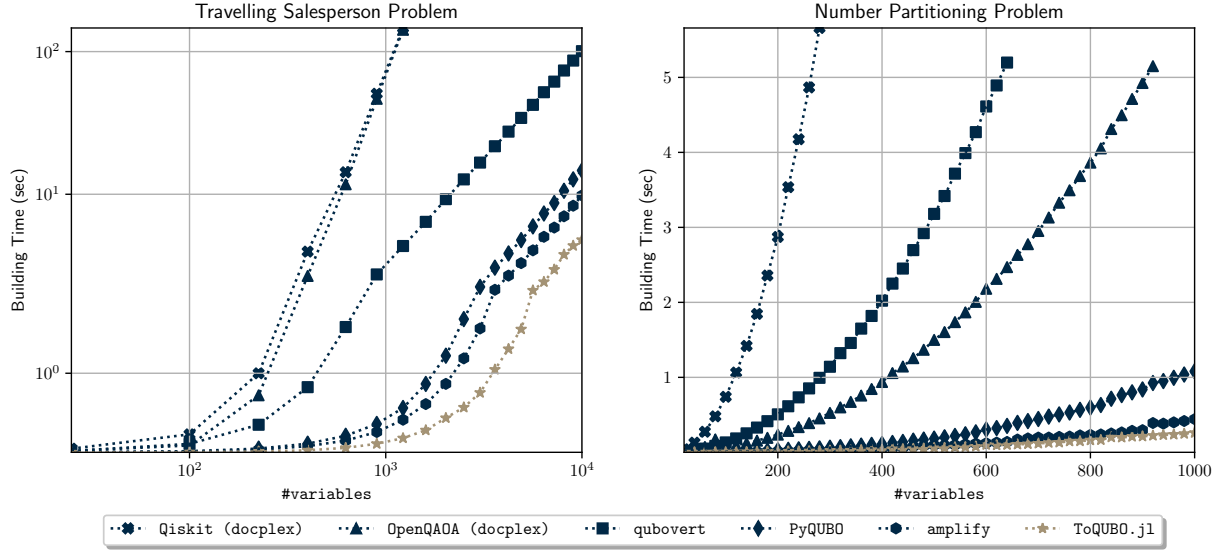
¹ Quantum Annealer; ² Quantum Gate Circuit;³ `DWave.jl` (Xavier and Ripper 2022a); ⁴ `DWaveNeal.jl` (Xavier and Ripper 2023a);⁵ `QuantumAnnealingInterface.jl` (Morrell and Coffrin 2022, Xavier and Ripper 2022b);⁶ `QiskitOpt.jl` (Xavier and Ripper 2023c); ⁷ `MQLib.jl` (Dunning et al. 2018, Xavier and Ripper 2023b);⁸ `PySA.jl` (Xavier 2023b); ⁹ `CIMOptimizer.jl` (Chen et al. 2022, Xavier 2023a);¹⁰ `Dwave-Qiskit-plugin` (D-Wave 2020);

Salesperson Problem (TSP) and the Number Partitioning Problem (NPP) are formulated with a varying number of variables (Lucas 2014). As presented in Figure 3, we evaluate the time to construct a QUBO model from 25 to 10000 variables (5 to 100 cities) for the TSP and from 5 to 1000 variables for the NPP. It is important to mention that for the `Qiskit` and `OpenQAOA` iterations, we create a QUBO from a `docplex` model.

Examining the outcomes for building a QUBO model from TSP and NPP instances - Figure 3 - together with the data from Table 2, it is possible to conclude that `ToQUBO.jl` stands out from its main competitors. In contrast with the other packages, `ToQUBO.jl` contains more features, providing distinguished compatibility with some problems. Besides, `ToQUBO.jl`'s execution time to build a QUBO model is very competitive, being faster than `PyQUBO` and greatly faster than `qubovert`, `Qiskit` and `OpenQAOA`.

Furthermore, under the perspective of programming experience, being developed as an MOI layer, `ToQUBO.jl` allows users to use JuMP for modeling, which is a widely used tool for operations research. Such characteristic is unique, considering `qubovert` and `PyQUBO`'s

Figure 3 Benchmark results for the Travelling Salesperson Problem (TSP) and Number Partitioning Problem (NPP) QUBO building time. Each TSP instance is comprised of $\sqrt{\text{\#variables}}$ cities. The partitioned sets in the NPP are of the form $[\text{\#variables}] = \{1, \dots, \text{\#variables}\}$.



syntax, where in both scenarios, one must get used to a whole new framework for modeling, exclusive for the package. We illustrate this advantage with Listings 2, 3, and 4, used to perform the TSP benchmarking.

Listing 2: `ToQUBO.jl` code to build a QUBO model from a TSP

```
function tsp(n::Integer, D::Matrix{Float64})
    model = Model(ToQUBO.Optimizer)

    @variable(model, x[1:n, 1:n], Bin)

    @objective(model, Min, sum(x[:,k]' * D * x[:, k%n+1] for k = 1:n))

    @constraint(model, [i in 1:n], sum(x[i,:]) == 1)
    @constraint(model, [j in 1:n], sum(x[:,j]) == 1)

    # Compilation
    optimize!(model)

    # f(x) = a * (x' Q x + b)
    Q, a, b = ToQUBO.qubo(model)
end
```

Listing 3: PyQUBO code to build a QUBO model from a TSP

```

def tsp(n: int, D: list[list[float]], lam: float = 1.0):
    # Variables
    x = Array.create('c', (n, n), 'BINARY')

    # Objective Function
    distance = sum(
        D[i][j] * x[i,k] * x[j, (k+1)%n]
        for i in range(n)
        for j in range(n)
        for k in range(n)
    )

    # Constraints
    time_const = 0.0
    for i in range(n):
        time_const += Constraint(
            (sum(x[i,k] for k in range(n)) - 1)**2,
            label="time{}".format(i)
        )

    city_const = 0.0
    for k in range(n):
        city_const += Constraint(
            (sum(x[i,k] for i in range(n)) - 1)**2,
            label="city{}".format(k)
        )

    H = distance + lam * (time_const + city_const)

    model = H.compile()

    qubo, offset = model.to_qubo()

```

Listing 4: qubovet code to build a QUBO model from a TSP graph.

```

def tsp(n: int, D: list[list[float]], lam: float = 1.0):
    model = 0

    # Variables
    x = [[qv.boolean_var(f"x[{i},{k}]", 1) for k in range(n)] for i in range(n)]

    # Objective Function
    for i in range(n):
        for j in range(n):
            for k in range(n):
                model += D[i][j] * x[i][k] * x[j][(k + 1) % n]

    # Constraints
    for i in range(n):
        model.add_constraint_eq_zero(
            sum(x[i][k] for k in range(n)) - 1, lam=lam
        )

    for k in range(n):
        model.add_constraint_eq_zero(
            sum(x[i][k] for i in range(n)) - 1, lam=lam
        )

    qubo = model.to_qubo()

```

Having said that, `ToQUBO.jl`'s performance and features, united with `JuMP`'s mathematical programming ecosystem and `QUBODrivers.jl`, allows `Julia` users to experiment on quantum and classical hardware-accelerated QUBO solvers in a seamless experience, just as they would do with a standard `JuMP` solver.

8. Conclusion

`QUBO.jl` is presented as a feature-rich and easy-to-use tool for working with QUBO formulations, and interfacing with their solvers, standing out from other packages in the Quantum Optimization field. Taking advantage of `ToQUBO.jl`'s fast reformulation routines, users can submit `JuMP` models to emergent QUBO devices as they would do with any other solver. Moreover, leveraging different architectures, each depending upon specific communication protocols, is simplified with `QUBOTools.jl`. In addition, with all three modules being open-source, anyone can contribute to their development and even define new solvers with `QUBODrivers.jl`, as discussed in Section 4, or create new file conversions with `QUBOTools.jl`.

Offering all three packages in `QUBO.jl` as a bundle makes them easy to integrate into optimization specialists' work, as it would be just an abstraction layer between their `JuMP` instances and quantum or classical sampling devices. As our project develops, both academia and industry will be able to use `QUBO.jl` as an all-in-one platform for quantum and hardware-accelerated methods, either by connecting their research pipelines to several available QUBO solvers or by publishing their own experimental hardware and algorithms to this growing ecosystem. Having no prerequisites needed, `QUBO.jl` addresses the gap between classical optimization and hardware-accelerated QUBO solvers, such as coherent Ising machines, quantum annealers, and variational algorithms for optimization.

8.1. Next Steps

We plan to expand the constraint mapping capabilities for future versions of our packages and consider new variable encoding techniques while still fulfilling competitive performance goals. More specifically, it is expected that, by employing the expertise earned from `SATyrus`'s development (Xavier and Lima 2022), it will be possible to enhance `ToQUBO.jl` with support for logical constraints in the short term, increasing its constraint programming repertoire.

Another forthcoming and important application is to run benchmarks for tracking the evolution of QUBO solvers and compare their performance against various problems. Such a

task will probably require specialized infrastructure to ensure consistency and reproducibility, for which the standards set by this work will be relevant. In this sense, it is planned that additional tools will be developed to archive and synthesize relevant QUBO instances for benchmarking. Additionally, it is important to develop new packages to interface with solution-sampling devices to encompass an even broader range of devices and ensure diversity among solvers. We envision that the ecosystem presented here will grow and allow an increasing number of users in operations research to access quantum and quantum-inspired solution methods.

Acknowledgments

The authors thank PSR for the great environment for research and open-source software development. Special thanks to Mario Pereira and Sergio Granville for the motivation, support, and many discussions that led to this work. We would also like to thank Zachary Morrell for reading through an early version of this manuscript and providing useful feedback on it. DEBN acknowledges grant CCF 1918549 of the National Science Foundation (NSF) for funding.

Appendix A: Encoding methods and their coefficients

Let $f \in \mathcal{F}$ be of the form (5). The quantity $\Delta(f) = \max_{\omega} |c_{\omega}|$, the absolute value of the coefficient with greater magnitude, is a rough yet interesting conditioning measure for the problem f represents. As a general recommendation, avoiding large values for $\Delta(f)$ leads to better reformulations. A large value in the largest coefficient affects the performance of the solvers as the precision required to accurately represent the quadratic function depends on the quotient of the smallest and the largest coefficients in absolute value (Bian et al. 2014).

This is one of the reasons why, in a situation where the unary encoding requires $\mathcal{O}(n)$ variables to be implemented, it might still be preferred over the binary method, even though it would demand only $\mathcal{O}(\log n)$ variables.

Appendix B: The Arithmetic Progression Encoding

Given an integer variable $z \in [a, b]$, we can represent it as a linear combination of binary variables with an encoding based on the arithmetic progression. Let $n = b - a$ and $N = \lceil \frac{1}{2}\sqrt{1+8n} - \frac{1}{2} \rceil$. Now consider N binary variables, $\mathbf{x} \in \mathbb{B}^N$. Finally, z can be represented by the following linear combination of these binary variables:

$$z = a + \sum_{i=1}^{N-1} i x_i + \left(n - \frac{N(N-1)}{2} \right) x_N \quad (10)$$

Note that the coefficients multiplying x in the summation are in an arithmetic progression and thus bounded by $N = \mathcal{O}(\sqrt{n})$ as $n - \frac{1}{2}N(N-1) \leq \frac{1}{2}\sqrt{1+8n} - \frac{5}{8} \leq N$. Naturally, it is possible to encode continuous variables by re-scaling (10), N fixed.

This strategy establishes a compromise solution between the resource-oriented binary encoding and the flat coefficients of the unary approach. The bounded-coefficient technique also shares this kind of reasoning (Karimi and Ronagh 2019).

The Fixtars Amplify framework presents a similar proposal but with a slightly different formulation (Matsuda 2022). To our knowledge, the two encoding schemes were developed independently, and there is no other presentation of an analogous method in the literature.

Appendix C: Benchmarking

The reformulation performance benchmark results presented in Figure 3 were all obtained by single-threaded runs on the same computer. Its environment details are summarized in Table 4.

Table 4 Benchmark Environment

System Information		Packages	
Operating System	Linux Ubuntu 22.04	ToQUBO.jl	v0.1.6
Python	CPython 3.10.6	PyQUBO	v1.4.0
Julia	julia 1.9.0	OpenQAOA	v0.1.3
CPU Specifications		qubovet	v1.2.5
Model	Intel® Xeon® E5-2630 v4	Qiskit	v0.41.0
Base Frequency	2.2 GHz	amplify	v0.11.1
Memory		Extras	
Installed Capacity	82 GB	docplex	v2.24.232

All code used in the benchmarks, including visualization recipes, is available online.

<https://github.com/psrenergy/ToQUBO-benchmark>

References

- Amazon Web Services (2020) Amazon Braket Python SDK. URL <https://github.com/aws/amazon-braket-sdk-python>.
- Amazon Web Services (2022) Braket.jl. URL <https://github.com/aws-labs/Braket.jl>.
- Belotti P, Kirches C, Leyffer S, Linderoth J, Luedtke J, Mahajan A (2013) Mixed-integer nonlinear optimization. *Acta Numerica* 22:1–131.
- Bergholm V, Izaac J, Schuld M, Gogolin C, Ahmed S, Ajith V, Alam MS, Alonso-Linaje G, AkashNarayanan B, Asadi A, Arrazola JM, Azad U, Banning S, Blank C, Bromley TR, Cordier BA, Ceroni J, Delgado A, Di Matteo O, Dusko A, Garg T, Guala D, Hayes A, Hill R, Ijaz A, Isacsson T, Ittah D, Jahangiri S, Jain P, Jiang E, Khandelwal A, Kottmann K, Lang RA, Lee C, Loke T, Lowe A, McKiernan K, Meyer JJ, Montañez-Barrera JA, Moyard R, Niu Z, O’Riordan LJ, Oud S, Panigrahi A, Park CY, Polatajko D, Quesada N, Roberts C, Sá N, Schoch I, Shi B, Shu S, Sim S, Singh A, Strandberg I, Soni

- J, Száva A, Thabet S, Vargas-Hernández RA, Vincent T, Vitucci N, Weber M, Wierichs D, Wiersema R, Willmann M, Wong V, Zhang S, Killoran N (2018) PennyLane: Automatic differentiation of hybrid quantum-classical computations. URL <http://dx.doi.org/10.48550/ARXIV.1811.04968>.
- Bernal Neira DE, Ajagekar A, Harwood SM, Stober ST, Tenev D, You F (2022) Perspectives of quantum computing for chemical engineering. *AIChE Journal* 68(6):e17651, ISSN 1547-5905, URL <http://dx.doi.org/10.1002/aic.17651>.
- Bezanson J, Edelman A, Karpinski S, Shah VB (2017) Julia: A fresh approach to numerical computing. *SIAM review* 59(1):65–98, URL <https://doi.org/10.1137/141000671>.
- Biamonte J, Wittek P, Pancotti N, Rebentrost P, Wiebe N, Lloyd S (2017) Quantum machine learning. *Nature* 549(7671):195–202, URL <http://dx.doi.org/10.1038/nature23474>.
- Bian Z, Chudak F, Israel R, Lackey B, Macready WG, Roy A (2014) Discrete optimization using quantum annealing on sparse ising models. *Frontiers in Physics* 2:56.
- Bodin G, Dias Garcia J, Legat B, Besançon M (2021) Dualization.jl: v0.3.4. URL <http://dx.doi.org/10.5281/zenodo.4718987>, <https://doi.org/10.5281/zenodo.4718987>.
- Boros E, Gruber A (2014) On quadratization of pseudo-boolean functions. *arXiv preprint arXiv:1404.6538*.
- Boros E, Hammer PL (2002) Pseudo-Boolean optimization. *Discrete Applied Mathematics* 123(1-3):155–225, ISSN 0166-218X, URL [http://dx.doi.org/10.1016/s0166-218x\(01\)00341-9](http://dx.doi.org/10.1016/s0166-218x(01)00341-9).
- Breloff T (2023) Plots.jl. URL <http://dx.doi.org/10.5281/zenodo.7994271>.
- Cao Y, Romero J, Olson JP, Degroote M, Johnson PD, Kieferová M, Kivlichan ID, Menke T, Peropadre B, Sawaya NP, et al. (2019) Quantum chemistry in the age of quantum computing. *Chemical reviews* 119(19):10856–10915.
- Chancellor N (2019) Domain wall encoding of discrete variables for quantum annealing and QAOA. *Quantum Science and Technology* 4(4):045004, ISSN 2058-9565, URL <http://dx.doi.org/10.1088/2058-9565/ab33c2>.
- Chen F, Isakov B, King T, Leleu T, McMahon P, Onodera T (2022) cim-optimizer: a simulator of the Coherent Ising Machine. URL <https://github.com/mcmahon-lab/cim-optimizer>.
- Coffrin C (2020) bqpjson. <https://github.com/lanl-ansi/bqpjson>.
- Cplex II (2009) V12. 1: User’s manual for CPLEX.
- D-Wave (2020) D-Wave Ocean plugin for IBM Qiskit. <https://github.com/dwavesystems/dwave-qiskit-plugin>.
- D-Wave (2022) Ocean. <https://github.com/dwavesystems/dwave-ocean-sdk>.
- Dattani N (2019) Quadratization in discrete optimization and quantum mechanics.
- Deshpande A (2022) Assessing the Quantum-Computing Landscape. *Commun. ACM* 65(10):57–65, ISSN 0001-0782, URL <http://dx.doi.org/10.1145/3524109>.

- Developers C (2022) Cirq. URL <http://dx.doi.org/10.5281/zenodo.7465577>, See full list of authors on Github: <https://github.com/quantumlib/Cirq/graphs/contributors>.
- Dias Garcia J (2021) QuadraticToBinary.jl: v0.2.4. URL <http://dx.doi.org/10.5281/zenodo.4718981>, <https://doi.org/10.5281/zenodo.4718981>.
- Dunning I, Gupta S, Silberholz J (2018) What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO. *INFORMS Journal on Computing* 30(3).
- Farhi E, Goldstone J, Gutmann S (2014) A Quantum Approximate Optimization Algorithm. *arXiv preprint arXiv:1411.4028*.
- FICO (2023) FICO Xpress Optimizer Reference Manual. URL <https://www.fico.com/en/products/fico-xpress-solver>.
- Freedman D, Drineas P (2005) Energy minimization via graph cuts: Settling what is possible. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, 939–946 (IEEE).
- Fujitsu (2021) Fujitsu Quantum-Inspired Optimization Services. URL <https://www.fujitsu.com/global/services/business-services/digital-annealer/qios/>.
- Glover F, Kochenberger G, Du Y (2019) Quantum Bridge Analytics I: a tutorial on formulating and using QUBO models. *4OR* 17(4):335–371, ISSN 1619-4500, URL <http://dx.doi.org/10.1007/s10288-019-00424-y>.
- Gurobi Optimization, LLC (2023) Gurobi Optimizer Reference Manual. URL <https://www.gurobi.com>.
- Hadfield S, Wang Z, O’gorman B, Rieffel EG, Venturelli D, Biswas R (2019) From the quantum approximate optimization algorithm to a quantum alternating operator ansatz. *Algorithms* 12(2):34.
- Hen I (2019) Equation planting: A tool for benchmarking ising machines. URL <http://dx.doi.org/0.1103/physrevapplied.12.011003>.
- Herman D, Googin C, Liu X, Sun Y, Galda A, Safro I, Pistoia M, Alexeev Y (2023) Quantum computing for finance. *Nature Reviews Physics* 1–16.
- IBM (2015) IBM® Decision Optimization Modeling for Python. <https://github.com/IBMDDecisionOptimization/docplex-examples>.
- Iosue JT (2022) qubovet. <https://github.com/jtiosue/qubovet>.
- Karimi S, Ronagh P (2019) Practical integer-to-binary mapping for quantum annealers. *Quantum Information Processing* 18(4):94.
- King J, Yarkoni S, Nevisi MM, Hilton JP, McGeoch CC (2015) Benchmarking a quantum annealing processor with the time-to-target metric. *arXiv preprint arXiv:1508.05087*.
- Kolmogorov V, Zabin R (2004) What energy functions can be minimized via graph cuts? *IEEE transactions on pattern analysis and machine intelligence* 26(2):147–159.

- Kowalsky M, Albash T, Hen I, Lidar DA (2022) 3-regular 3-xorsat planted solutions benchmark of classical and quantum heuristic optimizers. *Quantum Science and Technology* URL <http://dx.doi.org/10.1088/2058-9565/ac4d1b>.
- Kronqvist J, Bernal Neira DE, Lundell A, Grossmann IE (2019) A review and comparison of solvers for convex minlp. *Optimization and Engineering* 20:397–455.
- Legat B, Dowson O, Dias Garcia J, Lubin M (2021) MathOptInterface: A Data Structure for Mathematical Optimization Problems. *INFORMS Journal on Computing* ISSN 1091-9856, URL <http://dx.doi.org/10.1287/ijoc.2021.1067>.
- Liberti L (2009) Reformulations in mathematical programming: Definitions and systematics. *RAIRO-Operations Research* 43(1):55–85.
- Liberti L, Marinelli F (2014) Mathematical programming: Turing completeness and applications to software analysis. *Journal of Combinatorial Optimization* 28:82–104.
- Lima PMV, Morveli-Espinoza MMM, França FMG (2007) Logic as Energy: A SAT-Based Approach. Mele F, Ramella G, Santillo S, Ventriglia F, eds., *Advances in Brain, Vision, and Artificial Intelligence*, 458–467 (Berlin, Heidelberg: Springer Berlin Heidelberg), ISBN 978-3-540-75555-5.
- Lubin M, Dowson O, Dias Garcia J, Huchette J, Legat B, Vielma JP (2023) JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation* URL <http://dx.doi.org/10.1007/s12532-023-00239-3>.
- Lucas A (2014) Ising formulations of many NP problems. *Frontiers in Physics* 2:5, URL <http://dx.doi.org/10.3389/fphy.2014.00005>.
- Luo XZ, Liu JG, Zhang P, Wang L (2019) Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design. *arXiv preprint arXiv:1912.10877* .
- Mandra S, Katzgraber HG (2018) A deceptive step towards quantum speedup detection. *Quantum Science and Technology* 3(4):04LT01.
- Matsuda Y (2022) Fixstars Amplify SDK. URL <https://amplify.fixstars.com/en/docs/index.html>.
- Misener R, Floudas CA (2012) Global optimization of mixed-integer models with quadratic and signomial functions: a review. *Applied and Computational Mathematics* .
- Morrell Z, Coffrin C (2022) QuantumAnnealing.jl. URL <https://github.com/lanl-ansi/QuantumAnnealing.jl>.
- Nagarajan H, Lockwood O, Coffrin C (2021) QuantumCircuitOpt: An Open-source Framework for Provably Optimal Quantum Circuit Design. URL <http://dx.doi.org/10.1109/QCS54837.2021.00010>.
- Orús R, Mugel S, Lizaso E (2019) Quantum computing for finance: Overview and prospects. *Reviews in Physics* 4:100028, ISSN 2405-4283, URL <http://dx.doi.org/https://doi.org/10.1016/j.revip.2019.100028>.

- O'Malley D, Vesselinov VV (2016) ToQ.JL: A High-Level Programming Language for D-Wave Machines Based on Julia. URL <http://dx.doi.org/10.1109/hpec.2016.7761616>.
- Pardalos PM, Jha S (1992) Complexity of uniqueness and local search in quadratic 0–1 programming. *Operations research letters* 11(2):119–123.
- Perez HD, Joshi S, Grossmann IE (2023) Disjunctiveprogramming. jl: Generalized disjunctive programming models and algorithms for jump. *arXiv preprint arXiv:2304.10492* .
- Peruzzo A, McClean J, Shadbolt P, Yung MH, Zhou XQ, Love PJ, Aspuru-Guzik A, O'Brien JL (2014) A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5(1):4213, ISSN 2041-1723, URL <http://dx.doi.org/10.1038/ncomms5213>, number: 1 Publisher: Nature Publishing Group.
- Preskill J (2018) Quantum Computing in the NISQ era and beyond. *Quantum* 2:79, URL <http://dx.doi.org/10.22331/q-2018-08-06-79>.
- QuEra (2023) Bloqade.jl: Package for the quantum computation and quantum simulation based on the neutral-atom architecture. URL <https://github.com/QuEraComputing/Bloqade.jl/>.
- Rehfeldt D, Koch T, Shinano Y (2023) Faster exact solution of sparse maxcut and qubo problems. *Mathematical Programming Computation* 1–26.
- Rieffel EG, Hadfield S, Hogg T, Mandrà S, Marshall J, Mossi G, O’Gorman B, Plamadeala E, Tubman NM, Venturelli D, et al. (2019) From ansatzes to z-gates: a NASA view of quantum computing. *Future Trends of HPC in a Disruptive Scenario* 34:133.
- Rowley C (2022) PythonCall.jl: Python and Julia in harmony. URL <https://github.com/cjdoris/PythonCall.jl>.
- Sharma V, Saharan NSB, Chiew SH, Chiacchio EIR, Disilvestro L, Demarie TF, Munro E (2022) OpenQAOA – An SDK for QAOA. URL <http://dx.doi.org/10.48550/ARXIV.2210.08695>.
- Singhal K, Hietala K, Marshall S, Rand R (2022) Q# as a Quantum Algorithmic Language. *Proceedings of the 19th International Conference on Quantum Physics and Logic (QPL), Oxford, U.K., June 27–July 1, 2022*, URL <https://ks.cs.uchicago.edu/publication/q-logic/>.
- Stallman RM, Weinberg Z (1987) The c preprocessor. *Free Software Foundation* 16.
- Tamura K, Shirai T, Katsura H, Tanaka S, Togawa N (2021) Performance Comparison of Typical Binary-Integer Encodings in an Ising Machine. *IEEE Access* 9:81032–81039, ISSN 2169-3536, URL <http://dx.doi.org/10.1109/access.2021.3081685>.
- Tasseff B, Albash T, Morrell Z, Vuffray M, Lokhov AY, Misra S, Coffrin C (2022) On the Emerging Potential of Quantum Annealing Hardware for Combinatorial Optimization. *arXiv preprint arXiv:2210.04291* .
- Treinish M, Gambetta J, Nation P, qiskit bot, Kassebaum P, Rodríguez DM, de la Puente González S, Lishman J, Hu S, Krsulich K, Garrison J, Bello L, Yu J, Marques M, Gacon J, McKay D, Gomez J,

- Capelluto L, Travis-S-IBM, Mitchell A, Panigrahi A, Ierongil, Rahman RI, Wood S, Itoko T, Pozas-Kerstjens A, Wood CJ, Singh D, Risinger D, Arbel E (2022) Qiskit/qiskit: Qiskit 0.39.4. URL <http://dx.doi.org/10.5281/zenodo.7416349>.
- Xavier P, Ripper P, Andrade T, Dias Garcia J, Bernal Neira D (2022) ToQUBO.jl. URL <http://dx.doi.org/10.5281/zenodo.6387592>.
- Xavier P, Ripper P, Andrade T, Dias Garcia J, Bernal Neira DE (2023a) QUBODrivers.jl. URL <http://dx.doi.org/10.5281/zenodo.7826592>.
- Xavier P, Ripper P, Andrade T, Dias Garcia J, Bernal Neira DE (2023b) QUBO.jl. URL <http://dx.doi.org/10.5281/zenodo.7909166>.
- Xavier P, Ripper P, Andrade T, Dias Garcia J, Bernal Neira DE (2023c) QUBOTools.jl. URL <http://dx.doi.org/10.5281/zenodo.7826338>.
- Xavier PM (2023a) CIMOptimizer.jl. URL <http://dx.doi.org/10.5281/zenodo.8021080>.
- Xavier PM (2023b) PySA.jl. URL <http://dx.doi.org/10.5281/zenodo.7951575>.
- Xavier PM, Lima PMV (2022) Satyrus3: v3.0.7. URL <http://dx.doi.org/10.5281/zenodo.6512152>.
- Xavier PM, Ripper P (2022a) DWave.jl. URL <https://github.com/psrenergy/DWave.jl>.
- Xavier PM, Ripper P (2022b) QuantumAnnealingInterface.jl. URL <https://github.com/psrenergy/QuantumAnnealingInterface.jl>.
- Xavier PM, Ripper P (2023a) DWaveNeal.jl. URL <http://dx.doi.org/10.5281/zenodo.7512623>.
- Xavier PM, Ripper P (2023b) MQLib.jl. URL <http://dx.doi.org/10.5281/zenodo.7511330>.
- Xavier PM, Ripper P (2023c) psrenergy/QiskitOpt.jl: v0.1.0. URL <http://dx.doi.org/10.5281/zenodo.7535884>.
- Zaman M, Tanahashi K, Tanaka S (2021) PyQUBO: Python Library for Mapping Combinatorial Optimization Problems to QUBO Form. *arXiv* URL <http://dx.doi.org/10.48550/arxiv.2103.01708>.
- Zapata Computing Inc (2020) Orquestra. URL <https://www.zapatacomputing.com/orquestra-platform/>.