Ket Programming Guide

Evandro C. R. da Rosa[1], Jerusa Marchi[1], Rafael de Santiago[1]

[1]Grupo de Computação Quântica – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

evandro.crr@posgrad.ufsc.br, {jerusa.marchi,r.santiago}@ufsc.br

Abstract. *The development of quantum computing necessitates specialized software tools. Ket is an open-source quantum programming platform that enables gate-based quantum development in Python, supported by a Rust-based runtime library and simulator. This paper presents a programming guide for the Ket platform. It covers the core concepts required to develop quantum applications, including quantum process management, gate operations, measurement strategies, and Hamiltonian construction. Designed for readers with prior knowledge of quantum computing and Python, this guide bridges the gap between theoretical algorithms and practical software implementation. To consolidate these concepts, the paper concludes with a practical tutorial applying QAOA, VQE, and FALQON to the Max-Cut problem, ultimately equipping researchers and developers to fully utilize the Ket platform.*

1. Introduction

It has long been known theoretically that quantum computers have the potential to solve certain problems faster than classical computers, and this reality is steadily moving closer to practical application. While it remains unclear exactly when today's quantum computers will achieve a definitive advantage in solving real-world problems, the path toward this future is well-established [Acharya et al. 2025].

Quantum computing is an emerging paradigm for data processing that relies on the fundamental principles of quantum mechanics. This new paradigm introduces superposition and entanglement, powerful new tools for software development, while also imposing strict physical constraints, such as the impossibility of copying an unknown quantum state and the destructive nature of quantum measurement. These unique characteristics drive the need for specialized quantum programming tools.

Ket [Rosa et al. 2026] is an open-source quantum programming platform that provides the necessary functions and types to enable quantum programming natively in Python. The platform is built around the gate-based quantum computing model, where data is stored in qubits and the program is expressed as a quantum circuit.

This article is presented as a *Tutorial Paper*, serving as a comprehensive programming guide for the Ket platform, demonstrating how to develop quantum applications and explore the nuances of quantum software development. A priori knowledge of quantum computing and Python programming is expected from the reader. Exhaustive details, such as comprehensive lists of available gates and optional parameters, can be found in the Ket API documentation[1]; we recommend consulting the official documentation as supplementary material.

[1]https://quantumket.org

The remainder of this paper is organized as follows. Section 2 outlines the architecture of the Ket platform. Section 3 explains the management of the quantum process. Section 4 details the application of quantum gates, encompassing controlled operations, higher-order gate composition, and inverse operations. Section 5 discusses measurement strategies, expectation value calculations, and state extraction for retrieving classical information from quantum circuits. Section 6 provides a comprehensive guide to Hamiltonian construction. Section 7 consolidates the presented concepts through a practical tutorial implementing QAOA, VQE, and FALQON for the Max-Cut problem. Finally, Section 8 concludes the guide with final remarks and an outlook on future developments.

2. The Platform

The Ket project is structurally organized into three highly integrated components:

Ket The Python module serving as the frontend interface. It provides an expressive, user-facing syntax that seamlessly embeds quantum instructions within Python code.

Libket The platform's core runtime library. Developed in Rust for maximum performance, it is responsible for orchestrating the quantum process, managing opaque qubit references, and dynamically compiling the quantum circuits.

KBW The platform's default backend simulator. It features both sparse and dense simulation modes, allowing programmers to optimize memory usage and execution speed based on the circuit's structure. The sparse mode stores only non-zero probability amplitudes, making it highly efficient for algorithms that explore a limited subset of the Hilbert space, such as the QAOA ansatz [Hadfield et al. 2019], though it has limited multithreading capabilities. Conversely, the dense mode acts as a multicore state-vector simulator that computes the entire amplitude vector, maximizing parallel processing.

Both Libket and KBW are written in Rust and are exposed to the Python package via a C foreign language interface (FFI). Because the platform is under active development, neither the Rust nor the C APIs are strictly stable, and they may undergo changes between minor releases. However, the frontend Python module is designed to provide a stable, expressive, and user-friendly interface for the programmer.

This paper is based on Ket version 0.9.2.3. Ket can be easily installed using the Python package manager via `pip install ket-lang`. The platform supports Linux, macOS, and Windows, offering compiled binaries for x86_64 and Apple Silicon.

3. Quantum Process

Every operation in Ket is tied to a quantum process. The process holds all the necessary information about the quantum circuit and the target quantum execution backend. In Ket, a process is instantiated using the `Process` class. By default, the process sends the quantum execution to KBW using a sparse simulator limited to 32 qubits. Optional arguments can be passed to configure KBW or to designate other execution target.

From the programmer's perspective, the primary role of the process is to allocate qubits for execution using the `.alloc(n)` method, which allocates `n` qubits. These qubits are exposed to the programmer encapsulated within a `Quant` object, which acts as a list of opaque references to the underlying qubits. Many standard Python list operations

are available for `Quant` objects, including indexing, slicing, concatenation, and iteration. Even when a qubit is individually indexed, it remains encapsulated within a `Quant`.

```
process = Process()        # Creating a process with default arguments.
qubits = process.alloc(n) # Allocating n qubits.
```

The use of opaque references means that each qubit variable in Ket does not hold the raw quantum information itself, but merely points to it. This design mitigates the need for the programmer or the platform to handle the no-cloning theorem; copying a qubit variable simply copies the classical reference, not the underlying quantum state.

Ket is intrinsically designed to support two distinct quantum execution modes. However, the specific mode utilized during a program's runtime depends entirely on the capabilities and requirements of the target quantum execution backend. The modes are:

Batch In this paradigm, the target quantum device operates similarly to a traditional batch-job scheduler. The complete quantum circuit is constructed by the classical Python host and submitted to the backend as a single, indivisible job. Consequently, the quantum execution cannot be preempted or paused, meaning that classical control flow (such as `if`/`else` branching) cannot dynamically depend on intermediate quantum measurement results.

Live This paradigm enables real-time, dynamic interaction between the classical host and the quantum execution backend. The quantum state coherence is maintained while intermediate measurement results are returned to the classical program. This facilitates dynamic, on-the-fly circuit generation and algorithms that strictly require mid-circuit measurements and classical feed-forward operations. While quantum hardware supporting the live mode currently remains limited, this mode provides an essential simulation environment for researching quantum error correction and dynamic quantum circuits

When instantiating the `Process` class, the programmer can configure the underlying backend. The default KBW simulator natively supports both execution modes and accepts optional keyword arguments to fine-tune the simulation environment:

`execution` Explicitly dictates the execution mode for the simulator, accepting either `"live"` (the default) or `"batch"`.

`simulator` Specifies the internal state-vector simulation algorithm. It accepts either `"sparse"` (the default) or `"dense"`.

`num_qubits` Defines the maximum capacity of qubits available for allocation. If omitted, this defaults to 32 qubits for the sparse and 12 qubits for the dense simulator.

`gradient` A boolean flag (defaulting to `False`) that instructs the runtime library to enable automatic gradient computation for parameterized quantum circuits.

4. Quantum Gates

Ket provides a core set of built-in quantum gates, namely the Pauli gates, Hadamard, Phase, and the Rotation gates. These are implemented as functions that take a `Quant` as input, apply the desired unitary operation, and return the same `Quant`. Furthermore, any function that directly or indirectly calls these gates, provided it does not allocate or measure a qubit, is also considered a quantum gate. This loose definition allows for a highly expressive approach to creating custom quantum gates.

```
H(qubits) # Applying a Hadamard gate to every qubit in the Quant
```

4.1. Controlled Gates

It is important to note that the built-in standard gates are single-qubit operations. To achieve universal quantum computation, Ket allows any quantum gate to be conditioned on control qubits. This is performed using either the `ctrl` function or the `with control` block construct.

The `ctrl` function takes two positional arguments, the control qubits and the target gate, and returns a controlled version of that gate. For example, to apply a CNOT gate with qubit `a` as the control and `b` as the target:

```
ctrl(a, X)(b) # Controlled Pauli-X gate (CNOT)
```

Note that while CNOT, SWAP, QFT, and other commonly used multi-qubit gates are readily available within the platform, they can also be constructed manually from foundational built-in gates.

Alternatively, the `with control` instruction allows controlling entire scopes of quantum operations. For instance, to apply a Fredkin (Controlled-SWAP) gate, swapping qubits `a` and `b` based on the state of control qubit `c`:

```
with control(c):
    SWAP(a, b)
```

4.2. Gate Concatenation and Tensor Products

Programmers can construct custom quantum gates by combining existing ones using higher-order functions: `cat` for sequential concatenation and `kron` for parallel tensor products. The resulting composite gate accepts a specific number of `Quant` arguments, which is determined by its constituent operations.

Additionally, Ket natively supports operation broadcasting. Whenever a multi-qubit `Quant` array is passed to a gate, the operation is automatically applied qubit-wise. For example, while `CNOT(a, b)` is fundamentally a two-argument gate, passing arrays of equal length for `a` and `b` will apply the CNOT operation pairwise across their qubits.

The `cat` function takes a variable number of gates as arguments and returns a new gate representing their sequential application. Conversely, the `kron` function takes a list of n input gates and returns a new combined gate requiring n arguments (one for each gate's respective target). These constructs are useful when working with functions that accept gates as arguments, similar to the use of anonymous `lambda` functions.

To illustrate how these higher-order functions can be combined, consider the construction of a custom gate that prepares two qubits in the Bell state:

```
bell = cat(kron(H, I), CNOT) # Create a Bell gate
bell(a, b)                   # Prepare qubits a and b in the Bell state
```

4.3. Inverse Gates

Because unitary quantum operations are inherently reversible, Ket allows programmers to easily call the adjoint (inverse) of any gate using the `adj` function. This function takes a gate U as input and returns its mathematical inverse $U^\dagger$. For example, using the available Quantum Fourier Transform (`QFT`) gate, we can dynamically generate the inverse QFT:

```
iqft = adj(QFT)  # Create the inverse gate
iqft(qubits)     # Call the inverse gate
```

Since every operation is tied to a specific process context, the inverse operation must also be correctly linked to that process. This linkage is achieved automatically via the arguments passed to the inverse gate call; at least one of the arguments must be a `Quant` so the underlying process can be identified.

Another construct that relies on inverse gates is the `with around` block, which encapsulates the common circuit pattern $ABA^\dagger$, where A and B are quantum operations. For example, to implement an $R_{XX}(\theta)$ gate acting on qubits `a` and `b`:

```
def rxx(theta: float, a: Quant, b: Quant):
    with around(cat(kron(H, H), CNOT), a, b):
        RZ(theta, b)
```

Here, the composite operation `cat(kron(H, H), CNOT)` acts as operation A and is applied to both qubits. This is followed by the `RZ` gate, which acts as operation B. At the end of the `with around` block's scope, the inverse operation ($A^\dagger$) is automatically applied to complete the sequence. The quantum circuit representing this `rxx` gate is illustrated in Figure 1.

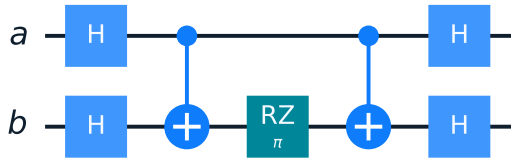

Figure 1. Circuit the `rxx` function with the parameter `theta` $= \pi$. Visualization produced using Ket's drawing utility: `qulib.draw(rxx, (1, 1), pi)`.

4.4. Global Phase

It is important to note that not every single-qubit gate can be naively implemented using just the basic built-in rotation gates, particularly when these operations are subjected to control qubits. This limitation arises due to the concept of *global phase*.

Consider the gate $\sqrt{X}$ (also known as the SX gate). For a single, isolated qubit, this is equivalent to the rotation gate $R_X(\pi/2)$. Both operations alter the quantum state identically, differing only by a global phase factor:

$$\underbrace{\frac{1}{\sqrt{2}}\begin{bmatrix}1 & -i\\ -i & 1\end{bmatrix}}_{R_X(\pi/2)} \times \underbrace{e^{i\frac{\pi}{4}}}_{\text{Global Phase}} = \frac{1}{2}\begin{bmatrix}1+i & 1-i\\ 1-i & 1+i\end{bmatrix} = \sqrt{X}. \tag{1}$$

However, a crucial difference emerges when these operations are executed in a controlled manner. A global phase, which is physically unobservable on an isolated qubit, becomes an observable *relative phase* when conditioned on control qubits, altering the quantum circuit's behavior. If we naively construct a controlled-$R_X(\pi/2)$ gate and simply append the global phase, it fails to produce the correct controlled-$\sqrt{X}$ ($C\sqrt{X}$) operation:

$$\underbrace{\frac{1}{\sqrt{2}}\begin{bmatrix}\sqrt{2} & 0 & 0 & 0\\ 0 & \sqrt{2} & 0 & 0\\ 0 & 0 & 1 & -i\\ 0 & 0 & -i & 1\end{bmatrix}}_{CR_X(\pi/2)} \times \underbrace{e^{i\frac{\pi}{4}}}_{\text{Global Phase}} \neq \frac{1}{2}\begin{bmatrix}2 & 0 & 0 & 0\\ 0 & 2 & 0 & 0\\ 0 & 0 & 1+i & 1-i\\ 0 & 0 & 1-i & 1+i\end{bmatrix} = C\sqrt{X}. \quad (2)$$

Therefore, to implement the gate $C\sqrt{X}$ correctly, the global phase from Eq. (1) must be embedded directly into the gate definition. Ket supports the injection of global phases into custom quantum gates using the `@global_phase` decorator. This allows programmers to construct gates that maintain their behavior under control conditions.

```
@global_phase(pi / 4)
def my_sx(qubit):
    RX(pi / 2, qubit)
```

5. Measurement and State Extraction

Built-in quantum gates have no side effects on classical execution; the only way to extract information from a quantum state is through measurement or simulation state extraction. Ket offers three primary methods for evaluating a quantum state: the `measure` function for single-shot measurements, the `sample` function for multi-shot measurements, and the `exp_value` function to compute the expectation value of a Hamiltonian. Additionally, Ket provides the `dump` function, which returns the exact state vector from a simulator.

5.1. Single-Shot Measurement

The `measure` function takes a `Quant` as input and returns a `Measurement` object. This object reference the result, acting similarly to an asynchronous programming Future.

In live execution mode, the measurement result is evaluated immediately and can be retrieved using the `.get()` method, which returns an integer representing the measured state. In batch mode, calling `.get()` triggers the quantum execution. Note that once execution starts in batch mode, all `Measurement` objects associated with the process are resolved, but the quantum process is terminated, preventing further operations. After the result becomes available, the `.get()` method can be called as often as needed.

```
process = Process()
a, b = process.alloc(2)
CNOT(H(a), b)              # Prepare a Bell state
m = measure(a + b).get()  # The result will be either 0 or 3
```

Whether a qubit can be manipulated after a measurement depends on the target execution backend. Typically, on physical quantum hardware, the qubit is invalidated

after measurement. However, in the KBW simulator, the qubit collapses to the measured state and remains available for subsequent operations. Single-shot measurement is particularly useful for implementing fault-tolerant algorithms, quantum communication protocols, and quantum error correction routines.

5.2. Multi-Shot Sampling

Because many quantum algorithms output a probability distribution rather than a single deterministic state, the `sample` function is often more appropriate. The `sample` function returns a `Samples` object, which operates similarly to the `Measurement` class.

Calling the `.get()` method triggers the quantum execution and returns a dictionary where the measurement result (as an `int`) is mapped to its exact observation count. The `.histogram()` method generates a plot of the measurement distribution.

```
process = Process()
a, b = process.alloc(2)
CNOT(H(a), b)                        # Prepare a Bell state
sample(a + b, 100).histogram("bin") # Histogram of 100 shots
```

On quantum hardware, sampling generally terminates the quantum process. The KBW simulator allows execution to continue after a sample; however, each sample represents an independent run. Consequently, entanglement correlations cannot be detected between two different samples, only within a single sample itself.

5.3. Expectation Values and Gradients

Calculating expectation values is crucial for near-term quantum algorithms, such as Variational Quantum Algorithms and Quantum Machine Learning [Amaral et al. 2026]. To calculate an expectation value, an observable is required. The `exp_value` function takes a `Hamiltonian` object as input and returns an `ExpValue` object. Calling the `.get()` method triggers execution and returns the calculated expectation value as a float.

```
# chsh_obs is an instance of a Hamiltonian object
exp_value(chsh_obs).get() # e.g., 2.828427125
```

Hamiltonian construction is detailed in Section 6. Note that the `exp_value` function requires only the observable as input; the target qubit references are inherently encapsulated within the Hamiltonian object itself.

A key feature of expectation values in Ket is the ability to calculate parameter gradients, which is essential for gradient-based optimization in Variational Quantum Algorithms. Gradient calculation is handled by the target backend via Native Simulation (*e.g.*, tensor networks) or automatically via the Parameter-Shift Rule [Mitarai et al. 2018] applied by Libket. To enable gradient calculation with the KBW simulator, the keyword argument `gradient=True` must be provided when instantiating the process.

```
process = Process(gradient=True)
theta = process.param(0.5) # Define a variational parameter
# ... perform expectation value calculation circuit ...
grad = theta.grad          # Retrieve the calculated gradient
```

5.4. State Vector Inspection

Ket provides the `dump` function to retrieve the full state vector from a simulator. Naturally, this is physically impossible on actual quantum hardware. The `dump` function is not intended to be a substitute for measurement. It does not collapse the quantum state, and extracting a subset of entangled qubits via a dump will not yield the expected partial trace of the system. Nevertheless, it is an invaluable tool for debugging. It returns a `QuantumState` object equipped with useful methods such as `.get()` to view complex probability amplitudes, `.sphere()` to plot a Bloch sphere for single qubits, and `.show()` to output LaTeX representations of the state.

6. Hamiltonian Construction

A core design philosophy of Ket is to make the programmatic construction of observables, including Hamiltonians, mirror their formal mathematical definitions as closely as possible. By utilizing linear combinations and tensor products of the Pauli matrices I, X, Y, and Z, Ket provides a complete basis for defining any n-qubit observable.

To translate these mathematical expressions into native Python syntax, Ket introduces the `with obs()` context manager. Within this execution block, the standard Pauli gate functions (`I`, `X`, `Y`, and `Z`) are temporarily redefined. Rather than applying unitary operations to a quantum circuit, they return observable objects, allowing programmers to construct operators using a syntax that directly reflects the underlying equations.

Scalar multiplication and division are fully supported. Complex scalars are also permitted, which is particularly useful for defining non-Hermitian operators such as the fermionic creation operator

$$\hat{a}_q^\dagger = \frac{X_q - iY_q}{2} = |1\rangle\langle 0|_q \,. \tag{3}$$

In Ket, this is implemented as:

```
with obs():
    a_dg = (X(q) - 1j * Y(q)) / 2
```

The use of complex scalars further facilitates the description of observables involving matrix multiplication, which is supported natively via the Python `@` operator. A practical example is the observable A utilized in the FALQON algorithm [Magann et al. 2022], defined by the commutator:

$$A = i[H_m, H_c]. \tag{4}$$

In Ket, this can be implemented using the `commutator` function or `matmul`:

```
A = 1j * commutator(Hm, Hc)
# Equivalently, using the matmul @ operator:
A = 1j * (Hm @ Hc - Hc @ Hm)
```

Note that because `Hm` and `Hc` are already instantiated as `Hamiltonian` objects, the `with obs()` context manager is not required for these operations.

It is crucial to note that for expectation value calculations (as discussed in Section 5.3), the final observable must be Hermitian. While intermediate steps or specific operators (such as $\hat{a}_q^\dagger$) may rely on complex coefficients, any `Hamiltonian` object passed directly to the `exp_value` function must ultimately resolve to real (`float`) coefficients, as physical observables strictly correspond to real eigenvalues.

Quadratic Unconstrained Binary Optimization (QUBO) formulations are frequently used in quantum optimization algorithms. QUBO variables take binary values $x_q \in \{0, 1\}$, whereas the Pauli-Z observable has eigenvalues $\{-1, +1\}$. To map between these spaces, Ket provides a mapping function `B` the transformation:

$$\texttt{B}(q) = \frac{1 - Z_q}{2} \tag{5}$$

By utilizing the function `B`, arbitrary QUBO Hamiltonians can be implemented directly within Ket. As a practical example, the QUBO Hamiltonian Knapsack problem:

$$H = -\overbrace{\sum_i v_i x_i}^{\texttt{obj}} + A \underbrace{\left(\sum_i w_i x_i - C \right)^2}_{\texttt{rest}} \tag{6}$$

This mathematical formulation translates intuitively into the following Ket code:

```
def knapsack(w: list[float], v: list[float],
             C: float, qubits: Quant, A: float) -> Hamiltonian:
    obj = sum(v_i * B(q) for v_i, q in zip(v, qubits))
    rest = (sum(w_i * B(q) for w_i, q in zip(w, qubits)) - C)**2
    return -obj + A * rest
```

Hamiltonians are also used to define the time evolution of a quantum system via the unitary operator e^{-iH}. Ket provides streamlined support for this through the `evolve` function. Currently, this function natively evolves Hamiltonians composed of 1-local terms and 2-local terms of the form XX, YY, and ZZ.

A critical requirement of the `evolve` function is that it assumes all individual terms within the provided Hamiltonian mutually commute. If the Hamiltonian contains non-commuting terms, the function will not automatically approximate the evolution. Despite this restriction, the `evolve` function accommodates the mixing and cost layers of algorithms like QAOA and FALQON.

7. Examples: MaxCut

In this section, we present three implementations to solve the Max-Cut problem using quantum computing: QAOA [Farhi et al. 2014], FALQON [Magann et al. 2022], and VQE [Peruzzo et al. 2014]. Here, our strict focus is on their implementation in Ket; for detailed algorithmic presentations, we refer the reader to the original works. The Max-Cut problem seeks to partition the vertices of a graph into two disjoint sets such that the number of edges spanning the two sets is maximized.

7.1. Defining the Hamiltonians

For QAOA, FALQON, and VQE, the objective of Max-Cut can be encoded into a cost Hamiltonian H_c. For a graph with edges $\mathcal{E}$, the cost Hamiltonian assigns a lower energy to valid bitstrings representing the cuts. Additionally, for QAOA and FALQON, we require a mixer Hamiltonian H_m to drive transitions between states.

$$H_c = -\frac{1}{2} \sum_{a,\, b \,\in\, \mathcal{E}} (1 - Z_a Z_b) \qquad (7)$$

$$H_m = \sum_q X_q \qquad (8)$$

```
def cost_h(edges, q: Quant):
  with obs():
    hc = sum(1 - Z(q[a]) * Z(q[b])
             for a, b in edges)
  return -hc / 2
```

```
def mixer_h(nodes: Quant):
  with obs():
    Hm = sum(X(q) for q in nodes)
  return Hm
```

7.2. QAOA

The QAOA ansatz consists of alternating applications of the time-evolution operators $e^{-i\gamma H_c}$ and $e^{-i\beta H_m}$. Since the terms within H_c and H_m mutually commute, we can pass these Hamiltonians directly to Ket's `evolve` function.

```
def qaoa_layer(edges, qubits, gamma, beta):
    evolve(gamma * cost_h(edges, qubits))
    evolve(beta * mixer_h(qubits))

def qaoa_ansatz(edges, qubits, gamma, beta):
    for g, b in zip(gamma, beta):
        qaoa_layer(edges, qubits, g, b)
```

A classical optimizer is needed to execute this algorithm. To use SciPy's `minimize`, we must create a cost function that encapsulates the quantum process:

```
def qaoa_objective(graph, n, parameters):
    process = Process(num_qubits=n, simulator="dense",
                      execution="batch")

    p = len(parameters) // 2 # Number of layers
    gamma = parameters[p:]
    beta = parameters[:p]

    qubits = process.alloc(n)
    H(qubits) # Prepare the initial uniform superposition
    qaoa_ansatz(graph, qubits, gamma, beta)

    return exp_value(cost_h(graph, qubits)).get()

res = minimize(
    partial(qaoa_objective, graph, n),
    parameters,      # Initial parameters
    method="COBYLA", # Gradient-free optimizer
)
```

7.3. VQE

Unlike QAOA, VQE does not prescribe a rigidly structured ansatz. A possible, hardware-efficient implementation for the Max-Cut problem is as follows:

```
def vqe_ansatz(graph, qubits, parameters):
    for p in parameters:
        for a, b in graph:
            CZ(qubits[a], qubits[b])
        RY(p, qubits)
```

As with QAOA, we need an objective function to utilize the SciPy minimizer. It is common to use gradient-based optimizers for VQE; here, we implement the objective function so that the gradient is calculated and returned alongside the cost value:

```
def vqe_objective(graph, n, parameters):
    process = Process(num_qubits=n, simulator="dense",
                      execution="batch", gradient=True)
    # Register parameters for gradient calculation
    parameters = process.param(*parameters)
    qubits = process.alloc(n)
    vqe_ansatz(graph, qubits, parameters)

    return (
        exp_value(cost_h(graph, qubits)).get(),
        [p.grad for p in parameters],
    )

res = minimize(
    partial(vqe_objective, graph, n),
    parameters,         # Initial parameters
    method="L-BFGS-B", # Gradient-based optimizer
    jac=True,
)
```

7.4. FALQON

FALQON utilizes a different approach from QAOA and VQE; it relies on feedback-based measurements to iteratively optimize the quantum state over several layers.

```
def falqon_layer(graph, qubits, beta, delta_t):
    evolve(delta_t * cost_h(graph, qubits))
    evolve(beta * delta_t * mixer_h(qubits))

def beta_h(graph, qubits):
    return -1j * commutator(mixer_h(qubits), cost_h(graph, qubits))
```

In Ket, we can use the KBW simulator and live execution to calculate the feedback parameter β and the cost expectation $\langle H_c \rangle$ interactively for each layer.

```
beta, cost = [0.0], []
process = Process()
qubits = H(process.alloc(n))

```

```
for _ in range(num_layers):
    falqon_layer(graph, qubits, beta[-1], delta_t)
    cost.append(exp_value(cost_h(graph, qubits)).get())
    beta.append(exp_value(beta_h(graph, qubits)).get())
```

8. Final Remarks

The Ket platform is developed within the Quantum Computing Group at the Federal University of Santa Catarina (GCQ-UFSC). Highlighting its impact, in 2025, Ket was recognized by the Brazilian Computer Society (SBC) with the Innovation Seal (*Selo de Inovação*) for its contribution to the national development of quantum computing.

More than just an introduction to the platform, this article contributes a comprehensive, structured programming guide. It establishes a clear methodology for quantum software development, ultimately lowering the barrier to entry and driving further research within the quantum ecosystem.

Ket remains a project in constant development. While the platform is already well-equipped for state-of-the-art research, its future roadmap prioritizes the development of domain-specific libraries and the integration of pulse-level control for error mitigation. Furthermore, to prepare for the upcoming paradigm of fault-tolerant quantum computing, the strict implementation of quantum error correction remains a core focus.

Acknowledgment

The authors acknowledge the financial support of FAPESC (No. 2024TR002672) and INCT-CQA (CNPq, No. 408884/2024-0).

References

Acharya, R., Abanin, D. A., Aghababaie-Beni, L., et al. (2025). Quantum error correction below the surface code threshold. *Nature*, 638(8052):920–926.

Amaral, C. A., Oliveira, V. L., Salazar, J. P. L. C., and Duzzioni, E. I. (2026). A Review of Quantum Machine Learning and Quantum-inspired Applied Methods to Computational Fluid Dynamics. *Braz J Phys*, 56(1):39.

Farhi, E., Goldstone, J., and Gutmann, S. (2014). A Quantum Approximate Optimization Algorithm.

Hadfield, S., Wang, Z., O'Gorman, B., Rieffel, E. G., Venturelli, D., and Biswas, R. (2019). From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz. *Algorithms*, 12(2):34.

Magann, A. B., Rudinger, K. M., Grace, M. D., and Sarovar, M. (2022). Feedback-Based Quantum Optimization. *Phys. Rev. Lett.*, 129(25):250502.

Mitarai, K., Negoro, M., Kitagawa, M., and Fujii, K. (2018). Quantum circuit learning. *Phys. Rev. A*, 98(3):032309.

Peruzzo, A., McClean, J., Shadbolt, P., Yung, M.-H., Zhou, X.-Q., Love, P. J., Aspuru-Guzik, A., and O'Brien, J. L. (2014). A variational eigenvalue solver on a photonic quantum processor. *Nat Commun*, 5(1):4213.

Rosa, E., Lussi, E., Marchi, J., De Santiago, R., and Duzzioni, E. (2026). Full Quantum Stack: Ket Platform. *Braz J Phys*, 56(1):45.