

JAQSI: Just Another Quantum Simulator

Melvin Strobl[†], Maja Franz[§], Lukas Scheller[†], Eileen Kuehn[†], Wolfgang Mauerer^{§¶}, Achim Streit[†],

[†] Karlsruhe Institute of Technology, Germany, {melvin.strobl, lukas.scheller, eileen.kuehn, achim.streit}@kit.edu

[§] Technical University of Applied Science Regensburg, Germany, {maja.franz, wolfgang.mauerer}@othr.de

[¶] Siemens AG, Foundational Technology, Munich, Germany

Abstract—Classical simulation supports the development and validation of quantum algorithms through full-state inspection, controlled noise models and backpropagation, albeit for small quantum systems. Existing frameworks offer gate-level execution, differentiable pulse-level dynamics or optimised hardware execution. Just another quantum simulator (JAQSI) combines gate- and pulse-level simulation in one differentiable circuit model built upon JAX. Circuits are represented as pure functions of numerical inputs which support vectorised batches and JIT compilation. Within a single circuit, matrix-defined operations, gates synthesised from Hamiltonian time evolution and noise channels can be composed freely. For a fast simulation, gate and channel fusion reduces passes over the state, while adjoint differentiation computes expectation-value gradients with a constant number of statevectors. At pulse level, JAQSI reduces evolution under single-term Hamiltonians to a scalar pulse-area integral and reuses cached execution plans and compiled solvers. In a benchmark we compare JAQSI with PennyLane, Qiskit, Qibo, Qulacs, QuTiP and dynamiqs. At gate level, we observe performance either on par or an order of magnitude faster across statevector, density, noise and gradient simulation than the compared simulators up to 12 qubits. Beyond 18 qubits, JAQSI is on par with PennyLane Lightning in both statevector and gradient simulation. In the statevector and noisy density-matrix pulse benchmarks, it is at least 5 times faster than the other simulators, even with its closed-form solve disabled. Disabling the latter increases its runtime by at most a factor of 1.5. We provide JAQSI as open-source software on GitHub and PyPI.

Index Terms—Quantum Computing, Classical Simulation, Quantum Machine Learning, Pulse Level Quantum Computing

I. INTRODUCTION

Quantum hardware remains a scarce resource as access to devices is limited and, in the noisy intermediate-scale quantum (NISQ) era, their results are dominated by noise [29]. Classical simulation is therefore still the primary workbench on which quantum algorithms are developed and validated. It also allows inspection of the full quantum state, controlled use of noise and gradients through the simulated state. This is especially relevant in quantum machine learning (QML), where parameterised quantum circuits (PQCs) are trained by classical optimisers [25, 2, 7]. A single training run may evaluate the circuit thousands of times, often requiring a gradient at each step. What usually requires the parameter-shift rules [38] can instead be done by backpropagation through the state in simulation. However, exact statevector simulation grows exponentially with the number of qubits [41], so simulation throughput constrains the size of the systems examined in studies.

General-purpose frameworks such as Qiskit [17], Qibo [9] and Qulacs [37] focus on fast circuit execution, and PennyLane [3] established the circuit as a differentiable node in a hybrid computation. At the other end, quantum dynamics libraries such as

QuTiP [23, 22] integrate the Schrödinger and master equations under time-dependent Hamiltonians, which is the level at which control pulses act on the physical qubits. Building such tools on JAX [6] is not new either. At gate level, TensorCircuit [40] offers differentiable and vectorised simulation through tensor-network contraction. At pulse level, Qiskit Dynamics [30], dynamiqs [15] and JAXQuantum [18] provide differentiable, hardware-accelerated solvers for quantum dynamics, and the pulse module of PennyLane relies on JAX as well. However, these tools leave room for a single stack that covers both gate and pulse levels while supporting end-to-end differentiation, vectorised batches, cached compilation and a plain Python interface. Tab. I summarises these features for the simulators compared in Sec. III.

TABLE I: Feature coverage of the simulators considered in this work. Gate and Pulse denote simulation levels. Grad, Batch and GPU denote gradient computation, vectorised batch execution and GPU acceleration, respectively. (✓) marks support limited to a backend, add-on or execution mode, such as Qibo’s pulse hardware emulator, QuTiP’s JAX-based gradients or PennyLane’s gate-level parameter broadcasting.

Simulator	Gate	Pulse	Grad	Batch	GPU
PennyLane [3]	✓	✓	✓	(✓)	✓
Qiskit [17]	✓	✗	✗	(✓)	✓
Qibo [9]	✓	(✓)	(✓)	✗	✓
Qulacs [37]	✓	✗	(✓)	✗	✓
QuTiP [22]	(✓)	✓	(✓)	✗	(✓)
dynamiqs [15]	✗	✓	✓	✓	✓
JAQSI	✓	✓	✓	✓	✓

Combining pulse-synthesised gates, ideal gates and noise channels in one circuit allows the pulse parameters of one layer to be trained while the other gates remain ideal and noise acts only on selected qubits. This isolates the effect of pulse-level control within one differentiable model.

The contributions can be summarised as follows: (1) a unified gate- and pulse-level simulator in which pulse-synthesised gates, unitary gates and noise channels compose freely within one circuit, (2) an execution engine with cached, batch-vectorised executables that reduces state passes through fusion and gate structure, keeps statevector storage independent of circuit depth during adjoint differentiation, and reduces pulse solves through closed-form, deduplicated and vectorised integration, and (3) a benchmark comparing JAQSI against PennyLane, Qiskit, Qibo, Qulacs, QuTiP and dynamiqs at both gate and pulse levels.

JAQSI is the stand-alone continuation of the simulation engine behind `qml-essentials` [33], which was developed to evaluate thousands of parameter samples in studies of quantum Fourier models (QFMs) [28, 31, 36, 35]. Up to version 0.2.0 [12], the engine was part of `qml-essentials`. From version 0.3.0 [13] on, `qml-essentials` uses JAQSI as a separate package. Its pulse-level modelling and quantum optimal control (QOC) were utilised in Ref. [10] and carried the experiments of Ref. [34], which show that access to pulse parameters improves the training of QFMs. We provide JAQSI under an MIT licence on GitHub¹ and through PyPI.

II. ARCHITECTURE

JAQSI is organised in four layers: (i) a *front end* that records quantum circuits and dispatches gates, (ii) an *orchestration* layer holding the recording tape, the memory model for batch chunking and the unitary and pulse implementations of every gate, (iii) an *engine* layer with the simulation kernels, the gate library, the noise channels and the Hamiltonian time evolution, all built on a shared Operation base, and (iv) *helper* modules for drawing, QOC and analysis. An overview of the architecture is shown in Fig. 1. The entire stack is implemented in JAX [6], making every simulation a pure function of its numerical inputs and fully differentiable.

A. Usage

From the user’s perspective, a circuit is a plain Python function that applies gates through the Gates entry point, and Script wraps it for execution. Wires are declared per gate, either as a single index or as a list for multi-qubit gates:

```

1  import jaqsi
2  from jaqsi import Gates
3  import jax.numpy as jnp
4
5  def circuit(phi, theta, omega):
6      Gates.Rot(phi, theta, omega, wires=0)
7      Gates.CX(wires=[0, 1])
8      Gates.CX(wires=[1, 2])
9
10     script = jaqsi.Script(circuit, n_qubits=3)
11     res = script.execute(type="expval",
        ↪ obs=[jaqsi.PauliZ(wires=0), jaqsi.PauliZ(wires=2)],
        ↪ args=[jnp.pi, jnp.pi/3, jnp.pi/4])

```

Arguments reach the circuit function through `execute`, while type and the observables in `obs` select the measurement. The following subsections describe what happens behind this call, from recording the circuit to executing it at gate and pulse level.

B. Circuit Recording

A quantum circuit in JAQSI is a plain Python function whose body instantiates an Operation to be registered via a *tape* mechanism. The tape is a thread-local stack of recorded operations that supports nested recordings and is rebuilt on every trace. The interface for the user can be divided into two classes: (i) Gates is the entry point for applying gates: it dispatches each call either to its unitary implementation (UnitaryGates), or, in pulse mode, to its pulse implementation (PulseGates), and attaches noise channels when configured to do so. (ii) Script wraps the circuit function and is the

entry point for everything that consumes a recorded tape, most importantly the execution described in Sec. II-D.

C. Operation Layer

Gates, observables and noise channels derive from the Operation base class. This class stores target wires and parameters, supports symbolic composition (dagger, power, products) and provides the tensor-contraction machinery. The specific gate implementation resides in the gateset module and contains, besides the usual fixed and parameterised gates, a virtual Barrier and a diagonal qubit unitary. Noise is expressed through Kraus channels [27] covering bit and phase flip, depolarising, amplitude and phase damping, thermal relaxation and generic user-defined channels, implemented in the Noise module. The operation layer also defines the Hamiltonian types feeding the pulse engine with ParametrizedHamiltonian representing a sum of time-dependent coefficient functions times Hermitian terms. A Hermitian wraps a static generator or observable. Both expose an `evolve` method returning a gate factory for the pulse-level engine of Sec. II-E.

A design decision central to performance is that an operation only ever stores the $2^k \times 2^k$ matrix acting on its k target qubits. Full $2^n \times 2^n$ matrices are never materialised for gates and channels, and observables are lifted to the full Hilbert space only for measurements outside the fast path for diagonal single-qubit observables of Sec. II-D. In addition, each gate declares static properties of its matrix, namely whether it permutes the computational basis (e.g. X, CX, SWAP), is diagonal (e.g. Z, R_Z, CZ), acts only if its leading control qubits are set (e.g. CR_X), and is unitary. This information is then used in the execution engine instead of inspecting the matrix entries, which are usually traced and thus unknown when the circuit is compiled.

D. Execution Engine

A call to `Script.execute` proceeds in four stages: (1) The circuit function is run once so that each operation registers itself on a fresh tape. (2) The qubit count is inferred from the recorded wires, and the presence of a Kraus channel on the tape selects density-matrix over statevector simulation. (3) The recorded operations are compiled into an execution plan, which one of the stateless simulation kernels applies to the state without any Python dispatch per gate. Whenever a density matrix is requested for a noise-free circuit, the engine simulates the statevector instead and forms $\rho = |\psi\rangle\langle\psi|$ once, reducing the per-gate cost from $O(4^n)$ to $O(2^n)$. (4) The resulting state is reduced to the requested output (state, probs, `expval` or density) and optionally sampled from if shots are required. Diagonal single-qubit observables take a fast path that reduces the probability vector in $O(2^n)$ per observable instead of contracting a lifted $2^n \times 2^n$ observable at $O(4^n)$.

JAQSI natively supports batched execution over parameters, inputs, or initial states. Here, any argument axis can be declared as a batch dimension, in which case the engine vectorises the simulation kernel via `vmap` and caches the resulting execution plan. The batched execution function is wrapped with Equinox’s filtered just-in-time (JIT) [21], which treats Python values as

¹<https://github.com/cirKITers/jaqsi>

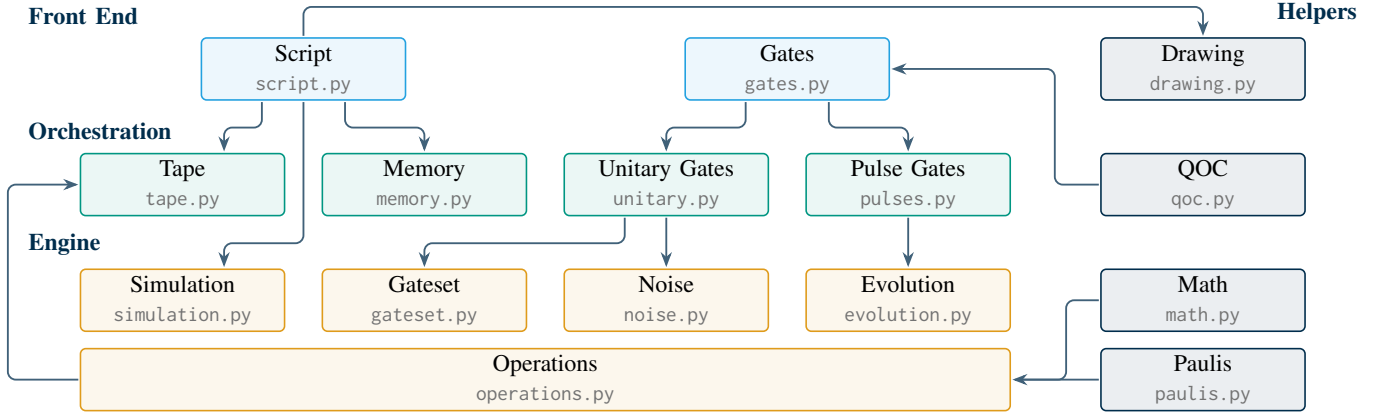


Fig. 1: Architectural overview of JAQSI. The front end records circuits on a tape and routes gates to their unitary or pulse implementations. The engine applies the resulting operations, which share an operation layer. Helper modules provide drawing, QOC and analysis. Arrows denote the principal dependencies between modules.

static and reuses compiled code for matching signatures. Purely array-valued signatures are additionally compiled ahead of time into an Accelerated Linear Algebra (XLA) executable which is then invoked directly. This allows skipping the per-call dispatch overhead usually dominating small circuits in tight loops. Before launching a batch, a memory model estimates the peak requirement from a fixed number of state-sized buffers, independent of circuit depth, and splits the batch into chunks if it exceeds the given budget, which requires that a single circuit alone fits into memory. On CPU, a batch whose working set exceeds the share of the last-level cache per core is further processed in vectorised tiles that fit into it. On CPU, JAX_NUM_CPU_DEVICES sets the number of JAX devices. Sufficiently large batches are sharded when multiple devices are available and the batch size is divisible by their number.

On top of this compiled and vectorised execution, the engine lowers the cost of a circuit in three ways. It reduces passes over the state, bounds gradient memory independently of circuit depth and, at pulse level, reduces the number of solves. The first lever acts in the statevector kernel, which keeps the state in rank- n tensor form, so that every entry of the plan costs one pass over the 2^n amplitudes. To reduce the number of these passes, consecutive gates on the same wires are multiplied into a single matrix as long as no gate in between touches these wires, similar to the gate merging of Qulacs [37] and the gate clustering of Ref. [16]. As these small matrix products remain part of the traced program, gradients flow through them unchanged.

The static gate properties of Sec. II-C determine how each plan entry updates the state. Permutations reindex the amplitudes, diagonal gates scale them elementwise, and controlled gates act only on amplitudes for which their controls are set. Runs of consecutive permutations or diagonal gates on arbitrary wires are also composed into a single precomputed index gather or a single elementwise factor, respectively. For the hardware-efficient ansatz of Fig. 2a on n qubits, this turns the $4n$ gates into n dense single-qubit blocks and one gather for the entire CX ring. Remaining dense blocks on one or two qubits are

applied as weighted sums of state slices and wider ones through einsum.

The same lever applies to the density-matrix kernel, which treats ρ as a rank- $2n$ tensor with separate ket and bra axes. A gate on up to two qubits enters the plan as its superoperator $U \otimes U^*$ and a channel as $\sum_k K_k \otimes K_k^*$ [39], both acting on the ket and bra axes of their wires at once. Gates and channels thereby become the same kind of plan entry and are fused alike, so that each block costs one pass over ρ instead of two per gate and two per Kraus operator. In the noisy variant of the benchmark circuit (Sec. III-A), for instance, the $9n$ gates and channels collapse into $4n$ blocks.

The second lever bounds the memory of gradients. For exact expectation values of circuits whose gates are all unitary, the engine computes gradients with the adjoint method [19], implemented as a custom vector-Jacobian product in JAX. Its forward pass keeps only the final state and the matrices of the plan, and its backward pass walks the plan in reverse, uncomputing the state with $U^\dagger$ while propagating the cotangent. This bounds the memory to a constant number of statevectors regardless of circuit depth, similar to the reversible differentiation in Yao [24]. As the cotangents refer to the fused matrices, JAX chains them into the gate parameters without requiring a generator per gate. All other cases fall back to automatic differentiation through the kernel, which stores the intermediate states. Either way, the simulation remains a pure JAX function of its arguments, so that gradient-based optimisation with standard optimisers applies directly to the circuit of Sec. II-A:

```

1 import jax
2 import optax as otx
3
4 def cost(params):
5     return script.execute(type="expval",
6                           ↪ obs=[jaqsi.PauliZ(wires=0)], args=tuple(params))[0]
7
8 params = jnp.array([0.1, 0.2, 0.3])
9 opt = otx.adam(0.1)
10 opt_state = opt.init(params)
11 for step in range(100):
12     grads = jax.grad(cost)(params)
13     updates, opt_state = opt.update(grads, opt_state, params)
14     params = otx.apply_updates(params, updates)

```

E. Pulse-Level Engine

At pulse level, a gate's drive Hamiltonian determines how its propagator changes during the pulse. JAQSI obtains that propagator by integrating the Schrödinger equation $dU/dt = -i\hat{H}(t)U$ for the gate's parameterised Hamiltonian. PulseGates holds the pulse-level composition where leaf gates (R_X, R_Y, R_Z, CZ) [10] carry their own drive Hamiltonian and envelope, while composite gates decompose into sequences of leaves. Pulse shapes are described by PulseEnvelope (Gaussian, square, cosine, derivative removal by adiabatic gate (DRAG), sech or user-defined), per-gate parameters by PulseParams, and both are unified in PulseInformation for fine-grained control. Each envelope $E(t)$ is centred at the midpoint of its pulse and, except for DRAG, modulates a single quadrature of the carrier, whose phase selects the rotation axis of R_X or R_Y . As in Qiskit², the Gaussian is lifted by its value at the pulse boundaries and rescaled to its peak, so that it vanishes at both ends instead of switching on and off with a step. Following Ref. [26], DRAG adds the derivative $-\beta\dot{E}(t)$ of this Gaussian as a second control on the orthogonal quadrature, where the weight β takes the place of the inverse anharmonicity [14]. As JAQSI models qubits as two-level systems without a leakage level to suppress, and the Gaussian alone already yields the exact rotation under the rotating wave approximation (RWA), the calibrated weight vanishes.

The integration itself lives in the Evolution class, which turns a Hamiltonian into a gate factory whose product is a regular Operation. This makes the execution engine of Sec. II-D agnostic to whether a gate was defined by a matrix or by a pulse. Here, two solver families are available in JAQSI for pulse simulation: (i) adaptive eighth-order Dormand–Prince (DP) integrators via diffrax [20], whose step-size control targets the requested error tolerances for arbitrary envelopes, and (ii) fixed-step commutator-free Magnus integrators of second and fourth order [5], which preserve unitarity by construction and incur a static, JIT-friendly cost per gate. The third lever reduces the cost of integrating pulses whose Hamiltonian has a single time-dependent drive term. A key observation here is that for such a pulse, only the integral of the drive coefficient, or pulse area, needs to be evaluated while the matrix part can be reused. Specifically, if the Hamiltonian is $f(\mathbf{p}, t)\hat{H}$ with a constant matrix $\hat{H} = V\Lambda V^\dagger$, it commutes with itself at all times. The first term of the Magnus expansion is then exact [4]: $U = V \exp(-iFA) V^\dagger$, where $F = \int f(\mathbf{p}, t) dt$ describes the pulse area. In this case, the adaptive solvers integrate only the scalar F , while the eigendecomposition of $\hat{H}$ is computed once when the circuit is traced. This keeps U differentiable in the pulse parameters even at degenerate eigenvalues. Under the RWA, the drives of R_X and R_Y reduce to such a term for single-quadrature envelopes like the Gaussian, so that every leaf gate takes this path. A DRAG pulse, in contrast, changes the ratio of its two quadratures over time, does not commute with itself at different times and is therefore integrated as the matrix equation. Before state evolution, pending pulses with compatible solver settings are grouped into vectorised solves.

²<https://quantum.cloud.ibm.com/docs/api/qiskit/1.4/qiskit.pulse.library.Gaussian>

Identical pulses within a group, such as the fixed rotations within a Hadamard, are evaluated once, and compiled solvers are reused across gates of the same structure. Their propagators then enter the ordinary gate-fusion pass of the execution engine of Sec. II-D. The shipped default pulse parameters are pre-calibrated against the ideal unitaries by the included QOC module, which builds on optax [8] and was presented, together with the drawing capabilities, in Ref. [10].

Because PulseGates builds directly on JAQSI operations, pulse-level gates can be placed in the same circuit as gate-level operations and noise channels, which route the circuit to the density-matrix kernel of Sec. II-D. The underlying Hamiltonian time evolution is exposed as well, such that any Hermitian provides an evolve method returning a time-evolution gate factory $U(t) = \exp(-it\hat{H})$. Multiplying a coefficient callable $f(\mathbf{p}, t)$ with a Hermitian then yields a ParametrizedHamiltonian with the latter being compiled once through evolve. Its parameters enter at call time, allowing the resulting operation to serve as a trainable layer in a pulse-level model. All three mechanisms compose freely in a single circuit:

```
1 from jaqsi import PulseGates
2
3 def coeff(p, t):
4     return p
5
6 def circuit(w, p):
7     PulseGates.RX(w, wires=0)
8     PulseGates.CX(wires=[0, 1])
9     X = jaqsi.PauliX(record=False).matrix
10    ph = coeff * jaqsi.Hermitian(matrix=X, wires=1,
11                                ↪ record=False)
12    ph.evolve()([p], 1.0)
13    jaqsi.DepolarizingChannel(0.1, wires=1)
14
15    script = jaqsi.Script(circuit, n_qubits=2)
16    rho = script.execute(type="density", args=(jnp.pi * 0.5, 0.3))
```

F. Analysis Tools

Three auxiliary modules complete the package. (i) math provides quantum-information utilities such as fidelity, trace distance, the Fubini–Study metric and quantum Fisher information, and partial traces. (ii) paulis implements a symplectic tableau representation of Pauli words [1] with symbolic Clifford conjugation in $O(n)$ per gate and statevector expectations in $O(n2^n)$, backing analytical, non-simulative evaluation of circuit classes built on top of JAQSI [33]. (iii) drawing renders a recorded circuit as ASCII text, a matplotlib figure or quantikz L^AT_EX code [10], and its pulse mode plots the pulse schedule, *i.e.* the sequence of pulses applied on each qubit, where shaded areas denote the pulse envelope and composite gates are automatically decomposed into basis gates.

Note that while these modules are not required for circuit execution, they provide convenient functionality typically seen in related simulators [3, 9].

III. BENCHMARKING

A. Benchmark Design

The benchmark measures the wall-clock time of a batched execution or gradient evaluation against a set of established simulators. At gate level, it uses the hardware-efficient ansatz of Fig. 2a. This circuit follows the Yao benchmark suite [24],

and its input encoding layer mirrors the parameter layout of a QML training step. At pulse level, it uses the H and $\text{CR}_X(\phi)$ ring circuit of Fig. 2b, which the transcribed pulse model covers. All angles are drawn uniformly from $[-\pi, \pi]$ with a fixed seed and shared across all simulators. Inputs differ for each instance in a batch, weights are shared across it, and both are redrawn in each iteration.

Gate-level runs span 2 to 12 qubits in every measured mode. The comparison includes PennyLane [3], Qiskit [17], Qibo [9] and Qulacs [37]. Each gate-level timing averages 20 iterations with a batch size of 10. At pulse level, the comparison includes PennyLane, QuTiP [22] and dynamiqs [15] for 2 to 12 qubits in each measured mode, over 20 iterations with a batch size of 2. Qiskit dropped its pulse support in version 2.0, Qulacs exposes none, and Qibo does so only through its hardware emulator.

The measured modes at both levels are statevector expectation values (expval), noise-free density matrices (density) and noisy density matrices (noise). In noise, a depolarising channel of strength $p = 0.01$ follows each gate on every qubit it acts on: $\rho \mapsto (1-p)\rho + \frac{p}{3}(X\rho X + Y\rho Y + Z\rho Z)$. These channels are part of the circuit, so all simulators apply the same sequence independently of their framework-specific noise models. Unlike in density, none can derive the noisy result from the statevector. At pulse level, the channels act between the pulse sequences of consecutive gates. At gate level, grad measures the gradient of $\sum_i \langle Z_i \rangle$, summed over the batch, with respect to the weights for all simulators except Qiskit and Qibo. The results of each simulator are cross-validated against JAQSI to an absolute tolerance of 10^{-8} at gate level, 10^{-6} for gradients, whose summation order differs between differentiation methods, and 10^{-5} at pulse level, the latter accounting for the accumulated solver error.

The benchmarks ran on a bare-metal node with AVX2 and AVX-512, running NixOS 26.05 (Yarara) at kernel 7.2.6, with 8 Intel Xeon Platinum 8260L CPUs (192 cores, 384 hardware threads) and 5.9 TiB memory. They ran inside a Docker container based on Debian 12 (bookworm) with Python 3.12, pinned to 240 logical CPUs (120 cores with SMT across 5 sockets) and limited to 4 TiB memory without swap. Every simulator is pinned to a single vCPU via the process CPU affinity, and multi-threading in OpenMP, BLAS, Qulacs and XLA is disabled through the respective environment variables and flags. The software stack uses Python 3.12.14, JAQSI 0.1.1 [11], JAX 0.11.1 and diffrax 0.7.2. The competitor versions are PennyLane 0.45.1 with Lightning 0.45.0, Qiskit 2.5.2 with Aer 0.17.2, Qibo 0.3.4 with qibojit 0.1.16, Qulacs 0.6.14, QuTiP 5.3.1 and dynamiqs 0.3.6 at commit 991fd45.³

Circuit and device construction happen outside the timed loop with garbage collection being disabled inside the loop. A single untimed warmup pass triggers JIT compilation. The reported values are the mean and standard deviation of the time per batch over the timed iterations. All simulators run in

their performance-optimised configuration, explained in detail below. PennyLane executes a jitted QNode on `default.qubit`, or on `default.mixed` in the noise mode, which vectorises the batch through parameter broadcasting like JAQSI and is differentiated by backpropagation through JAX. As a second gradient reference, PennyLane Lightning runs the adjoint method [19] on the compiled `lightning.qubit` device, which executes a broadcast batch sample by sample and accepts no channels. Qiskit runs on Aer with the statevector method in `expval` and the density_matrix method otherwise, submitting the batch as a single parameterised circuit whose parameters Aer binds per sample⁴. Qibo runs on its qibojit backend in the measured modes and evaluates each $\langle Z_i \rangle$ through its symbolic observable⁵. As gate fusion requires concrete angles, Qulacs rebuilds and fuses its circuit for every sample inside the timed loop. In these measurements, this is faster than updating the angles of a parametric circuit built once from 10 qubits on in `expval` and from 4 in density. In noise, where its optimiser would discard the channels, Qulacs runs unfused and builds its circuit only once. Qibo and Qulacs loop over the batch in Python, and Qulacs differentiates each sample through the backpropagation of its parametric circuits, whereas Qiskit and Qibo require additional packages for gradients. Qiskit and Qibo parameterise the depolarising channel by the weight λ of the maximally mixed state, so the benchmark sets $\lambda = 4p/3$ [27].

At pulse level, each gate expands into a sequence of time evolutions $dU/dt = -i\hat{H}(t)U$, one per basis gate of its decomposition. A H contributes 3 such pulses and a CX 7, so the CR_X of Fig. 2c accounts for 18 and the full circuit for $21n$. That schedule is transcribed once from JAQSI, using its calibrated Gaussian envelope for driven rotations under the RWA, so that all backends integrate identical pulses instead of their own pulse model. Each pulse is integrated on its own two- or four-dimensional space and the resulting propagator is contracted into the statevector, or applied from both sides to the density matrix in the noise mode, as in the pulse-level engine of Sec. II-E. In density, all backends form the density matrix from the final statevector. In noise, PennyLane applies channels through `default.mixed`, while the benchmark code applies them for QuTiP and dynamiqs. JAQSI and dynamiqs vectorise the batch dimension while PennyLane and QuTiP loop over it in Python. Furthermore, PennyLane integrates each pulse directly with the DP 5(4) solver of `jax.experimental.ode` that its `ParametrizedEvolution` is built on, bypassing the per-operation overhead of that API, QuTiP with its default Adams method, and JAQSI and dynamiqs with DP 8(7). As every pulse Hamiltonian is a single term $f(t)\hat{H}$ that commutes with itself at all times, JAQSI integrates only the scalar pulse area $\int f(t) dt$ and exponentiates $\hat{H}$ in its eigenbasis [4], whereas the other backends integrate the full matrix equation. All backends run at an absolute and relative solver tolerance of 10^{-10} .

The transcription is validated by comparing the infidelity between pulse- and gate-level simulation for each simulator, with JAQSI as the gate-level reference for QuTiP and dynamiqs. Across both pulse configurations, all backends and all qubit

³The code, configurations and raw results to reproduce the benchmarks are available at <https://github.com/cirKITers/jaqsi-data> [32].

⁴https://qiskit.github.io/qiskit-aer/stubs/qiskit_aer.AerSimulator.html

⁵<https://qibo.science/qibo/stable/code-examples/advancedexamples.html>

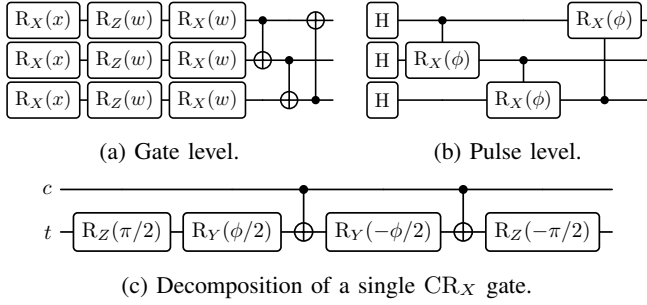


Fig. 2: Benchmark circuits for $n = 3$. (a) Gate-level hardware-efficient ansatz with input angles x and trainable weights w . (b) Pulse-level circuit with a $\text{CR}_X(\phi)$ ring that closes from the last qubit to the first. (c) Decomposition of one CR_X into rotations and two CX gates, with control c and target t . Each CX becomes HCZH on its target, and each H becomes $\text{R}_Z(\pi)$, $\text{R}_Y(\pi/2)$ and a correction phase. The resulting circuit has $21n$ pulses.

counts, the infidelities of the noise-free and noisy density matrices range from 7×10^{-15} to 6×10^{-14} , within a few hundred times double-precision machine epsilon. All four backends reach nearly the same value at each qubit count, consistent with a residual from the shared pulse schedule. Gate- and pulse-level simulation therefore agree to numerical precision.

B. Results

Fig. 3 shows two runtime regimes at gate level: fixed execution overhead dominates small circuits, while passes over the state matter increasingly as the qubit count grows. Tab. II summarises the speedup of JAQSI over the compared simulators at both levels. Within the timed loop, every call of JAQSI hits the cached, ahead-of-time compiled XLA executable and the whole batch is vectorised via vmap. The jitted PennyLane QNode compiles to an equally batched executable, whereas PennyLane Lightning, Qulacs, Qiskit and Qibo handle the batch sample by sample and pay a fixed cost per sample. At few qubits, this per-sample overhead dominates, so JAQSI and PennyLane have similar runtimes and lead the remaining simulators. As the state grows, passes over it account for more of the runtime, and the optimisations of Sec. II-D determine the margins.

In statevector simulation, JAQSI and PennyLane are within about 15 percent through 7 qubits. Beyond that, JAQSI leads by up to a factor of 2.6. Qulacs trails by 1.7 to 2.0 times, while Qiskit and Qibo trail by 4.7 to 7.0 and 6.5 to 13 times, respectively. The lead over PennyLane Lightning peaks at 7 qubits and narrows beyond. The growing gap to PennyLane reflects the first lever: fusion reduces the $4n$ gates of the ansatz to $n + 1$ plan entries, whereas PennyLane applies every gate in a pass of its own.

In noise-free density-matrix simulation, JAQSI is on par with or slightly slower than PennyLane at every measured qubit count, taking at most 28 percent longer. Both form the density matrix from the final statevector, so writing the 4^n output dominates

TABLE II: Speedup of JAQSI over each simulator across the measured qubit counts. Each range gives the minimum and maximum ratio of the other simulator’s mean execution time to that of JAQSI for the indicated mode. Values below one mean the other simulator is faster. Expval, Density, Noise and Grad denote statevector expectation values, noise-free and noisy density matrices, and gradients, respectively. Dashes mark modes without a reported comparison. In the pulse-level matrix-ODE rows, JAQSI integrates the matrix equation instead of using its closed-form pulse solve.

Simulator	Expval	Density	Noise	Grad
<i>Gate level</i>				
PennyLane	0.88–2.6	0.78–0.99	0.97–5.9	0.97–2.5
PennyLane Lightning	4.4–8.4	3.3–12	–	5.5–33
Qiskit	4.7–7.0	4.8–68	1.5–12	–
Qibo	6.5–13	7.4–49	3.2–23	–
Qulacs	1.7–2.0	1.9–43	3.0–13	2.1–3.3
<i>Pulse level</i>				
PennyLane	9.1–12	–	6.1–9.3	–
QuTiP	158–375	–	14–231	–
dynamiqs	7.5–17	–	7.2–13	–
<i>Pulse level (matrix ODE)</i>				
PennyLane	6.9–8.9	–	5.7–8.4	–
QuTiP	110–277	–	13–175	–
dynamiqs	5.5–13	–	5.1–13	–

from 9 qubits on. PennyLane Lightning trails JAQSI by 3.3 to 12 times, while Qulacs, Qiskit and Qibo evolve the full density matrix and fall behind by up to 68 times.

Under noise, JAQSI is the fastest from 4 qubits on, with a consistent lead of 1.4 to 5.9 times over PennyLane. Its lead over Qiskit falls from about 12 times at 4 qubits to 1.5 times at 12, while the lead over Qibo remains at least 3.2 times across the run. Every simulator evolves the full density matrix in this mode, so passes over it set the runtime at larger qubit counts. JAQSI applies $4n$ fused superoperator blocks, whereas PennyLane’s default.mixed applies each of the $9n$ gates and channels separately, which is in line with this lead. The lead over the remaining simulators at few qubits, in turn, reflects their per-sample overhead.

In gradient evaluation, JAQSI is on par with PennyLane up to 5 qubits and pulls ahead beyond. Qulacs trails by 2.1 to 3.3 times across the gate-level run. Qiskit and Qibo require additional packages for gradients and are excluded from this mode. The lead of JAQSI over PennyLane Lightning peaks at 6 qubits and narrows steadily beyond. The margin over PennyLane is the second lever: the adjoint method walks the fused plan backwards once, so that a gradient of JAQSI costs about 3 forward passes at 12 qubits, whereas backpropagation in PennyLane caches the intermediate state of every gate [19]. PennyLane Lightning uses the adjoint method as well, but runs the batch sample by sample through the Python interface of PennyLane, so that its per-sample cost dominates up to 12 qubits while the time of JAQSI already almost doubles per

qubit. The extended run up to 20 qubits in [Sec. A](#) shows how this advantage narrows. PennyLane Lightning overtakes JAQSI in statevector simulation from 19 qubits on and is faster in gradient evaluation only at 19 qubits, while the lead of JAQSI over PennyLane grows to 5.4 times in statevector simulation.

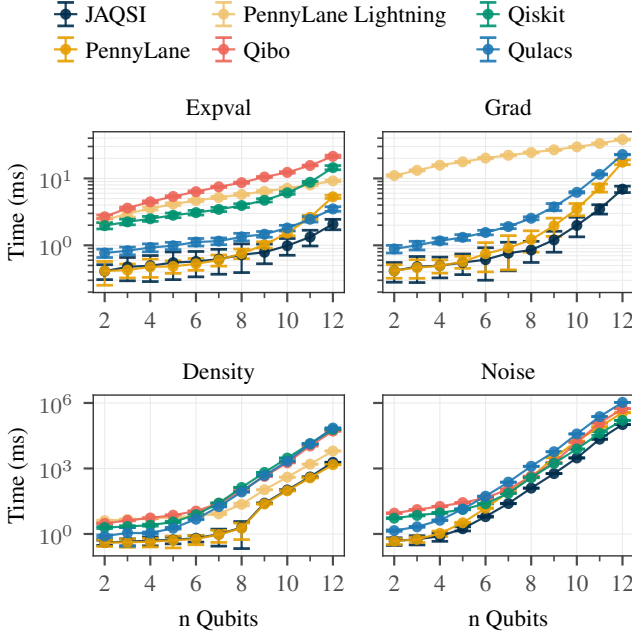


Fig. 3: Gate-level execution time versus qubit count for the single-layer hardware-efficient ansatz, with a logarithmic time axis. Expval denotes statevector expectation values, Grad denotes gradients, and Density and Noise denote noise-free and noisy density matrices. Each point is the mean time per batch of 10 over 20 iterations. Error bars denote the standard deviation. Qiskit and Qibo are absent from Grad, and PennyLane Lightning is absent from Noise.

[Fig. 4](#) shows the pulse-level results. In statevector simulation, JAQSI leads PennyLane by a nearly constant factor, dynamiqs by a factor that grows to 17 at 8 qubits and declines to 13 at 12, and QuTiP by more than two orders of magnitude. Under noise, the leads over PennyLane and dynamiqs persist, whereas the lead over QuTiP drops from above 200 times at 4 qubits to 14 to 19 times from 9 qubits on. In noise-free density-matrix simulation, the leads resemble those in statevector simulation at few qubits. From 10 qubits on, materialising the 4^n density matrix dominates the runtime of JAQSI. Its lead then shrinks to 1.1 to 6.3 times at 12 qubits.

Part of the gap to PennyLane and QuTiP reflects the batching asymmetry noted in [Sec. III-A](#), although a batch of 2 bounds that share to a factor of 2. The larger part stems from the third lever, the pulse-level engine of [Sec. II-E](#). As every pulse of the benchmark schedule is a single-term Hamiltonian, JAQSI integrates the scalar pulse area instead of the matrix equation. It further stacks all pulses of equal shape into one vectorised solve, solves identical pulses only once, *e.g.* 18 distinct solves for the 126 pulses at 6 qubits, and fuses the resulting propagators, *e.g.* the 63 pulses at 3 qubits into 16 passes over the state. PennyLane and dynamiqs, in contrast,

integrate one matrix equation per pulse in sequence, and QuTiP additionally dispatches every solve through Python. The lead over PennyLane stays nearly constant with the qubit count, as JAQSI lowers both the fixed cost per call and the time per additional pulse by about the same factor. Under noise, the density matrix dominates the runtime from about 7 qubits on, where JAQSI applies pulses and channels as fused superoperators, while the per-solve overhead of QuTiP loses weight. The matrix-ODE run in [Sec. B](#) compares performance with the closed-form solve disabled. Disabling it increases the runtime of JAQSI by at most a factor of 1.5 in the plotted statevector and noisy density-matrix modes. JAQSI remains faster than the compared pulse simulators in these modes ([Tab. II](#)).

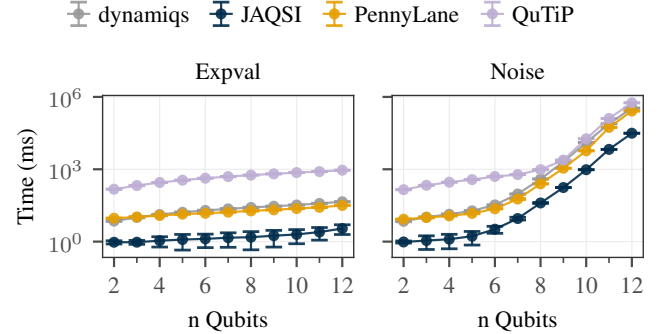


Fig. 4: Pulse-level execution time from 2 to 12 qubits for the H and CR_X ring circuit, with a logarithmic time axis. Expval denotes statevector expectation values, and Noise denotes noisy density matrices. Each execution integrates $21n$ pulse evolutions for n qubits. Each point is the mean time per batch of 2 over 20 iterations. Error bars denote the standard deviation.

IV. DISCUSSION & CONCLUSION

JAQSI combines gate- and pulse-level simulation in an end-to-end differentiable, batched and compiled stack while keeping circuits as plain Python functions. In the main single-threaded gate-level benchmark up to 12 qubits, it is faster than Qulacs, Qiskit and Qibo in every compared mode at every measured qubit count. It keeps pace with or outperforms the compiled PennyLane path in statevector simulation and gradient evaluation, and it is the fastest of the compared pulse-level simulators in the plotted statevector and noisy density-matrix modes. In noise-free density-matrix simulation, JAQSI is on par with or slightly slower than PennyLane at every measured qubit count. PennyLane Lightning illustrates how the gate-level margins change with circuit size: its per-sample overhead favours JAQSI at smaller qubit counts, while the extended run of [Sec. A](#) shows it overtaking JAQSI in statevector simulation from 19 qubits on and in gradient evaluation at 19 qubits. Qulacs also closes in on JAQSI in statevector simulation and is 8 percent faster at 20 qubits, while remaining slower in gradient evaluation.

Two features of the benchmark design qualify these comparisons. In the noise-free density mode, JAQSI and PennyLane form the result from a statevector, whereas Qiskit, Qibo and Qulacs evolve the full $O(4^n)$ density matrix. The margin over the latter therefore partly reflects the cheaper computation.

In the noise mode, every simulator evolves the full density matrix. Here the lead of JAQSI reflects vectorised batches and fused superoperators.

At pulse level, every pulse of the benchmark schedule has a single-term Hamiltonian, which JAQSI solves through its pulse area while the other backends integrate the matrix equation. This shortcut only applies to single-quadrature envelopes such as the Gaussian used in the benchmark. The DRAG envelope of JAQSI adds its derivative correction as a second control on the orthogonal quadrature [26], so that its pulses, as well as drives beyond the RWA, do not commute with themselves at different times and fall back to the matrix equation. Solving that equation for the Gaussian benchmark schedule makes JAQSI at most 1.5 times slower (Fig. 6). This comparison suggests that the shortcut accounts for only a small part of its lead in the plotted modes.

The remaining limits concern memory, differentiation and benchmark scope. Chunking, tiling and sharding act only on the batch axis, so a single state that exceeds the memory budget cannot be executed regardless of batch size. Representations that exploit limited entanglement, such as tensor networks, could extend this range beyond the statevector approach. The adjoint gradient uses a custom vector-Jacobian product and therefore supports reverse-mode differentiation only. Forward-mode differentiation, as used for the quantum Fisher information of the math module, falls back to automatic differentiation through the simulation kernel. The benchmarks use 1 thread, batches of 10 at gate level and 2 at pulse level, and a single-layer ansatz. Larger batches, deeper circuits and multi-threaded or GPU execution may shift the margins, so those configurations need separate measurement. Future work may investigate tensor-network representations, benchmark these other configurations and add an interface through which JAQSI can serve as a backend for other simulators.

Acknowledgements Implementation of numerical experiments as well as grammar and wording checks were partially supported by Anthropic Claude Opus series. MS, EK and AS acknowledge support by the state of Baden-Württemberg through bwHPC. LS acknowledges the support by the Doctoral School Karlsruhe School of Elementary and Astroparticle Physics: Science and Technology. MF and WM acknowledge partial support by the German Federal Ministry of Research, Technology and Space (BMFTR), funding programme “Research Program Quantum Systems”, grant number 13N17387, and by the German Research Foundation, grant 563122858 (MA 9739/1-1). WM acknowledges support by the High-Tech Agenda of the Free State of Bavaria.

REFERENCES

- [1] Scott Aaronson and Daniel Gottesman. “Improved Simulation of Stabilizer Circuits”. In: *Physical Review A* 70.5 (2004), p. 052328. DOI: [10.1103/PhysRevA.70.052328](https://doi.org/10.1103/PhysRevA.70.052328).
- [2] Marcello Benedetti, Erika Lloyd, Stefan Sack, et al. “Parameterized quantum circuits as machine learning models”. In: *Quantum science and technology* 4.4 (2019), p. 043001. DOI: [10.1088/2058-9565/ab4eb5](https://doi.org/10.1088/2058-9565/ab4eb5).
- [3] Ville Bergholm, Josh Izaac, Maria Schuld, et al. *PennyLane: Automatic Differentiation of Hybrid Quantum-Classical Computations*. 2018. arXiv: [1811.04968 \[quant-ph\]](https://arxiv.org/abs/1811.04968).
- [4] Sergio Blanes, Fernando Casas, José Angel Oteo, et al. “The Magnus Expansion and Some of Its Applications”. In: *Physics Reports* 470.5–6 (2009), pp. 151–238. DOI: [10.1016/j.physrep.2008.11.001](https://doi.org/10.1016/j.physrep.2008.11.001). eprint: [0810.5488](https://arxiv.org/abs/0810.5488).
- [5] Sergio Blanes and Per Christian Moan. “Fourth- and Sixth-Order Commutator-Free Magnus Integrators for Linear and Non-linear Dynamical Systems”. In: *Applied Numerical Mathematics* 56.12 (2006), pp. 1519–1537. DOI: [10.1016/j.apnum.2005.11.004](https://doi.org/10.1016/j.apnum.2005.11.004).
- [6] James Bradbury, Roy Frostig, Peter Hawkins, et al. *JAX: Composable Transformations of Python+NumPy Programs*. 2018. URL: <http://github.com/google/jax>.
- [7] Marco Cerezo, Guillaume Verdon, Hsin-Yuan Huang, et al. “Challenges and opportunities in quantum machine learning”. In: *Nature computational science* 2.9 (2022), pp. 567–576. DOI: [10.1038/s43588-022-00311-3](https://doi.org/10.1038/s43588-022-00311-3).
- [8] DeepMind, Igor Babuschkin, Kate Baumli, et al. *The DeepMind JAX Ecosystem*. 2020. URL: <http://github.com/google-deepmind>.
- [9] Stavros Efthymiou, Sergi Ramos-Calderer, Carlos Bravo-Prieto, et al. “Qibo: A Framework for Quantum Simulation with Hardware Acceleration”. In: *Quantum Science and Technology* 7.1 (Jan. 2022), p. 015018. DOI: [10.1088/2058-9565/ac39f5](https://doi.org/10.1088/2058-9565/ac39f5). arXiv: [2009.01845 \[quant-ph\]](https://arxiv.org/abs/2009.01845).
- [10] Maja Franz, Melvin Strobl, Jonathan Hunz, et al. “Software Between Quantum and Machine Learning - and Down to Pulses”. In: *2026 IEEE International Conference on Quantum Software (QSW)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2026, pp. 66–78. DOI: [10.1109/QSW72780.2026.00017](https://doi.org/10.1109/QSW72780.2026.00017).
- [11] Maja Franz, Melvin Strobl, and Eileen Kuehn. *JAQSI*. Version 0.1.1. Oct. 2026. DOI: [10.5281/zenodo.23087368](https://doi.org/10.5281/zenodo.23087368).
- [12] Maja Franz, Melvin Strobl, Eileen Kuehn, Lucas van der Horst, Paul Schillinger, Jonathan Hunz, and Johannes Wahl. *QML Essentials*. Version 0.2.0. May 2026. DOI: [10.5281/zenodo.20067335](https://doi.org/10.5281/zenodo.20067335).
- [13] Maja Franz, Melvin Strobl, Eileen Kuehn, Lucas van der Horst, Paul Schillinger, Jonathan Hunz, and Johannes Wahl. *QML Essentials*. Version 0.3.0. Sept. 2026. DOI: [10.5281/zenodo.22256367](https://doi.org/10.5281/zenodo.22256367).
- [14] Jay M. Gambetta, Felix Motzoi, Seth T. Merkel, et al. “Analytic Control Methods for High-Fidelity Unitary Operations in a Weakly Nonlinear Oscillator”. In: *Phys-*

- ical Review A* 83.1 (2011), p. 012308. DOI: [10.1103/PhysRevA.83.012308](https://doi.org/10.1103/PhysRevA.83.012308). arXiv: [1011.1949](https://arxiv.org/abs/1011.1949).
- [15] Pierre Guilmin, Adrien Bocquet, Élie Genois, et al. “Dynamiqs: an open-source Python library for GPU-accelerated and differentiable simulation of quantum systems”. 2025. URL: <https://github.com/dynamiqs/dynamiqs>.
- [16] Thomas Häner and Damian S. Steiger. “0.5 Petabyte Simulation of a 45-Qubit Quantum Circuit”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2017. DOI: [10.1145/3126908.3126947](https://doi.org/10.1145/3126908.3126947). arXiv: [1704.01127](https://arxiv.org/abs/1704.01127).
- [17] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, et al. *Quantum computing with Qiskit*. 2024. arXiv: [2405.08810](https://arxiv.org/abs/2405.08810) [quant-ph].
- [18] Shantanu R. Jha, Shoumik Chowdhury, Gabriele Roller, et al. *JAXQuantum: An auto-differentiable and hardware-accelerated toolkit for quantum hardware design, simulation, and control*. Version 0.3.0. 2024. URL: <https://jaxquantum.org>.
- [19] Tyson Jones and Julien Gacon. *Efficient Calculation of Gradients in Classical Simulations of Variational Quantum Algorithms*. 2020. arXiv: [2009.02823](https://arxiv.org/abs/2009.02823) [quant-ph].
- [20] Patrick Kidger. “On Neural Differential Equations”. PhD thesis. University of Oxford, 2021. eprint: [2202.02435](https://arxiv.org/abs/2202.02435).
- [21] Patrick Kidger and Cristian Garcia. *Equinox: Neural Networks in JAX via Callable PyTrees and Filtered Transformations*. Differentiable Programming workshop at Neural Information Processing Systems 2021. 2021. arXiv: [2111.00254](https://arxiv.org/abs/2111.00254) [cs.LG].
- [22] Neill Lambert, Eric Giguère, Paul Mencia, et al. “QuTiP 5: The Quantum Toolbox in Python”. In: *Physics Reports* 1153 (2026), pp. 1–62. DOI: [10.1016/j.physrep.2025.10.001](https://doi.org/10.1016/j.physrep.2025.10.001).
- [23] Boxi Li, Shah Nawaz Ahmed, Sidhant Saraogi, et al. “Pulse-Level Noisy Quantum Circuits with QuTiP”. In: *Quantum* 6 (Jan. 2022), p. 630. DOI: [10.22331/q-2022-01-24-630](https://doi.org/10.22331/q-2022-01-24-630). arXiv: [2105.09902](https://arxiv.org/abs/2105.09902) [quant-ph].
- [24] Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, et al. “Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design”. In: *Quantum* 4 (2020), p. 341. DOI: [10.22331/q-2020-10-11-341](https://doi.org/10.22331/q-2020-10-11-341).
- [25] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, et al. “Quantum circuit learning”. In: *Physical Review A* 98.3 (2018), p. 032309. DOI: [10.1103/PhysRevA.98.032309](https://doi.org/10.1103/PhysRevA.98.032309).
- [26] Felix Motzoi, Jay M. Gambetta, Patrick Rebentrost, et al. “Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits”. In: *Physical Review Letters* 103.11 (2009), p. 110501. DOI: [10.1103/PhysRevLett.103.110501](https://doi.org/10.1103/PhysRevLett.103.110501). arXiv: [0901.0534](https://arxiv.org/abs/0901.0534).
- [27] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge: Cambridge University Press, 2010. DOI: [10.1017/CBO9780511976667](https://doi.org/10.1017/CBO9780511976667).
- [28] Adrián Pérez-Salinas, Alba Cervera-Lierta, Elies Gil-Fuster, et al. “Data re-uploading for a universal quantum classifier”. In: *Quantum* 4 (Feb. 2020), p. 226. DOI: [10.22331/q-2020-02-06-226](https://doi.org/10.22331/q-2020-02-06-226). arXiv: [1907.02085](https://arxiv.org/abs/1907.02085).
- [29] John Preskill. “Quantum Computing in the NISQ Era and Beyond”. In: *Quantum* 2 (2018), p. 79. DOI: [10.22331/q-2018-08-06-79](https://doi.org/10.22331/q-2018-08-06-79). arXiv: [1801.00862](https://arxiv.org/abs/1801.00862) [quant-ph].
- [30] Daniel Pizzuoli, Christopher J. Wood, Daniel J. Egger, et al. “Qiskit Dynamics: A Python Package for Simulating the Time Dynamics of Quantum Systems”. In: *Journal of Open Source Software* 8.90 (2023), p. 5853. DOI: [10.21105/joss.05853](https://doi.org/10.21105/joss.05853).
- [31] Maria Schuld, Ryan Sweke, and Johannes Jakob Meyer. “The effect of data encoding on the expressive power of variational quantum machine learning models”. In: *Physical Review A* 103.3 (Mar. 2021), p. 032430. DOI: [10.1103/PhysRevA.103.032430](https://doi.org/10.1103/PhysRevA.103.032430). arXiv: [2008.08605](https://arxiv.org/abs/2008.08605).
- [32] Melvin Strobl and Maja Franz. *JAQSI-data*. Version 0.1.1. Oct. 2026. DOI: [10.5281/zenodo.23122633](https://doi.org/10.5281/zenodo.23122633).
- [33] Melvin Strobl, Maja Franz, Eileen Kuehn, et al. “QML-Essentials – A Framework for Working with Quantum Fourier Models”. In: *2025 IEEE International Conference on Quantum Software (QSW)*. 2025, pp. 238–243. DOI: [10.1109/QSW67625.2025.00035](https://doi.org/10.1109/QSW67625.2025.00035).
- [34] Melvin Strobl, Maja Franz, Lukas Scheller, et al. “Beyond Gates: Pulse Level Quantum Fourier Models”. In: *Proceedings of the 2026 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2026. arXiv: [2605.04945](https://arxiv.org/abs/2605.04945) [quant-ph].
- [35] Melvin Strobl, Gabriel Mejia, Eileen Kuehn, et al. *Unflattening by Flattening – How Input Distributions Shape Output Variance in Angle-Encoded Circuits*. 2026. arXiv: [2610.01446](https://arxiv.org/abs/2610.01446) [quant-ph].
- [36] Melvin Strobl, M. Emre Sahin, Lucas van der Horst, et al. “Fourier fingerprints of ansatzes in quantum machine learning”. In: *Quantum Science and Technology* 11.4 (Aug. 2026), p. 045008. DOI: [10.1088/2058-9565/ae8df8](https://doi.org/10.1088/2058-9565/ae8df8).
- [37] Yasunari Suzuki, Yoshiaki Kawase, Yuya Masumura, et al. “Qulacs: a fast and versatile quantum circuit simulator for research purpose”. In: *Quantum* 5 (Oct. 2021), p. 559. DOI: [10.22331/q-2021-10-06-559](https://doi.org/10.22331/q-2021-10-06-559).
- [38] David Wierichs, Josh Izaac, Cody Wang, et al. “General Parameter-Shift Rules for Quantum Gradients”. In: *Quantum* 6 (Mar. 2022), p. 677. DOI: [10.22331/q-2022-03-30-677](https://doi.org/10.22331/q-2022-03-30-677). arXiv: [2107.12390](https://arxiv.org/abs/2107.12390).
- [39] Christopher J. Wood, Jacob D. Biamonte, and David G. Cory. “Tensor Networks and Graphical Calculus for Open Quantum Systems”. In: *Quantum Information and Computation* 15.9–10 (2015), pp. 759–811. arXiv: [1111.6950](https://arxiv.org/abs/1111.6950).
- [40] Shi-Xin Zhang, Jonathan Allcock, Zhou-Quan Wan, et al. “TensorCircuit: A Quantum Software Framework for the NISQ Era”. In: *Quantum* 7 (2023), p. 912. DOI: [10.22331/q-2023-02-02-912](https://doi.org/10.22331/q-2023-02-02-912). arXiv: [2205.10091](https://arxiv.org/abs/2205.10091).
- [41] Yiqing Zhou, E. Miles Stoudenmire, and Xavier Waintal. “What Limits the Simulation of Quantum Computers?” In: *Physical Review X* 10.4 (2020), p. 041038. DOI: [10.1103/PhysRevX.10.041038](https://doi.org/10.1103/PhysRevX.10.041038). arXiv: [2002.07730](https://arxiv.org/abs/2002.07730) [quant-ph].

APPENDIX A GATE-LEVEL SCALING BENCHMARK

To examine gate-level performance beyond 12 qubits, the expval and grad modes of [Sec. III-A](#) are repeated for the single-layer hardware-efficient ansatz from 8 to 20 qubits. Each timing averages 50 iterations with a batch size of 10. This run includes PennyLane, PennyLane Lightning and Qulacs alongside JAQSI. While the runtimes of JAQSI and PennyLane already grow by about a factor of 2 per qubit at 12 qubits, that of PennyLane Lightning only does so once the state outweighs its fixed cost per sample, so that a crossover is expected at larger qubit counts.

[Fig. 5](#) confirms this expectation. In statevector simulation, the lead of JAQSI over PennyLane Lightning narrows from 11 times at 8 qubits to 3 percent at 18 qubits. PennyLane Lightning is 1.1 times faster at 19 qubits and 1.3 times faster at 20 qubits. Meanwhile, the lead over PennyLane grows from 1.3 times at 8 qubits to 4.6 to 5.4 times from 17 qubits on. Qulacs closes its gap to JAQSI from 2.4 times at 8 qubits to 6 percent at 19 qubits and is 8 percent faster at 20 qubits. From 18 qubits on, the time of JAQSI grows by 2.3 to 2.6 times per qubit, whereas that of PennyLane Lightning about doubles, so PennyLane Lightning is expected to extend its lead beyond 20 qubits in this mode. In gradient evaluation, the lead of JAQSI over PennyLane Lightning narrows from 31 times at 8 qubits to 1.2 times at 18 qubits. PennyLane Lightning is 1.1 times faster at 19 qubits, but its runtime grows about fourfold from 19 to 20 qubits, so that JAQSI leads again by 1.3 times at 20 qubits. JAQSI remains 1.5 to 4.8 times faster than PennyLane. It also remains ahead of Qulacs in gradients, although that lead narrows from 3.2 times at 8 qubits to 1.8 to 1.9 times at 19 and 20 qubits. Here, the cost of a gradient of JAQSI keeps rising with the qubit count and reaches about 7 forward passes at 20 qubits.

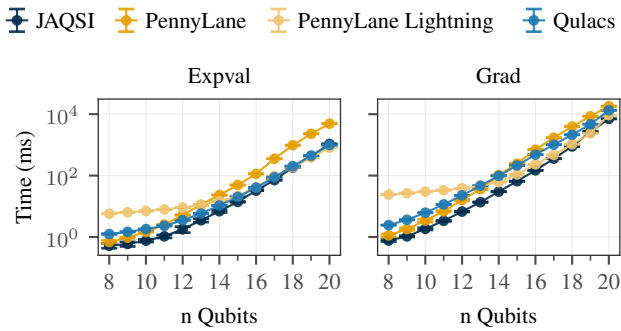


Fig. 5: Gate-level execution time of JAQSI, PennyLane, PennyLane Lightning and Qulacs from 8 to 20 qubits for statevector expectation values (Expval) and gradients (Grad) of the single-layer hardware-efficient ansatz, with a logarithmic time axis. Each point is the mean time per batch of 10 over 50 iterations. Error bars denote the standard deviation.

APPENDIX B PULSE-LEVEL MATRIX-ODE BENCHMARK

To assess the role of its closed-form pulse solve, JAQSI is run again on the pulse schedule of [Sec. III-A](#) with that solve disabled. The run spans 2 to 12 qubits with batches of 2 over 20 iterations. It integrates the full matrix equation for each pulse, as PennyLane, QuTiP and dynamiqs do in both runs. The other backends and the absolute and relative solver tolerances of 10^{-10} remain the same. Both configurations use the default Gaussian envelope of JAQSI for driven rotations. The second quadrature of derivative removal by adiabatic gate (DRAG) would require the matrix equation for every R_Y in either configuration, so the Gaussian keeps the pulse schedule otherwise unchanged.

[Fig. 6](#) shows that the matrix-ODE configuration is at most 1.5 times slower than the closed-form baseline of [Fig. 4](#). In the plotted modes, JAQSI remains 5.7 to 8.9 times faster than PennyLane, 13 to 277 times faster than QuTiP and 5.1 to 13 times faster than dynamiqs ([Tab. II](#)). Under noise, the two JAQSI configurations have similar runtimes at larger qubit counts, where density-matrix operations dominate. The fixed-step commutator-free Magnus integrators of [Sec. II-E](#) are not exercised in this benchmark.

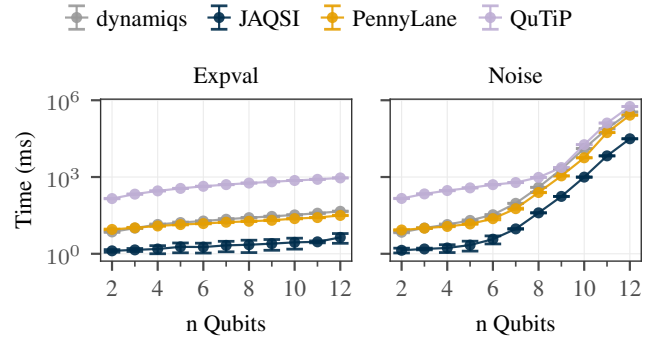


Fig. 6: Pulse-level execution time from 2 to 12 qubits for the H and CR_X ring circuit, with a logarithmic time axis. Expval denotes statevector expectation values, and Noise denotes noisy density matrices. JAQSI integrates the matrix equation for all $21n$ pulse evolutions instead of using its closed-form pulse solve. Each point is the mean time per batch of 2 over 20 iterations. Error bars denote the standard deviation.