Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design

Xiu-Zhe Luo[1,2,3,4], Jin-Guo Liu[1], Pan Zhang[2], and Lei Wang[1,5]

[1]Institute of Physics, Chinese Academy of Sciences, Beijing 100190, China
[2]Institute of Theoretical Physics, Chinese Academy of Sciences, Beijing 100190, China
[3]Department of Physics and Astronomy, University of Waterloo, Waterloo N2L 3G1, Canada
[4]Perimeter Institute for Theoretical Physics, Waterloo, Ontario N2L 2Y5, Canada
[5]Songshan Lake Materials Laboratory, Dongguan, Guangdong 523808, China

December 24, 2019

Figure 1: The logo of Yao. It contains a Chinese character pronounced as yāo, which stands for being unitary.

We introduce Yao, an extensible, efficient open-source framework for quantum algorithm design. Yao features generic and differentiable programming of quantum circuits. It achieves state-of-the-art performance in simulating small to intermediate-sized quantum circuits that are relevant to near-term applications. We introduce the design principles and critical techniques behind Yao. These include the quantum block intermediate representation of quantum circuits, a builtin automatic differentiation engine optimized for reversible computing, and batched quantum registers with GPU acceleration. The extensibility and efficiency of Yao help boost innovation in quantum algorithm design.

1 Introduction

Yao is a software for solving practical problems in quantum computation research. Given the limitations of near-term noisy intermediate-scale quantum circuits [1], it is advantageous to treat quantum devices as co-processors and compliment their abilities with classical computing resources. Variational quantum algorithms have emerged as a promising research direction in particular. These algorithms typically involve a quantum circuit with adjustable gate parameters and a classical optimizer. Many of these quantum algorithms, including the variational quantum eigensolver [2–4], quantum approximate optimization algorithm [5], quantum circuit learning [6, 7], and quantum circuit Born machine for generative modeling [8, 9] have had small scale demonstrations in experiments [10–14]. There are still fundamental issues in this field that call for better quantum software alongside hardware advances. For example, variational optimization of random circuits may encounter exponentially vanishing gradients [15] as the qubit number increases. Efficient quantum software is crucial for designing and verifying quantum algorithms in these challenging regimes. Other research demands also call for quantum software that features a small overhead for repeated feedback control, convenient circuit structure manipulations, and efficient gradient calculation besides simply pushing up the number of qubits in experiments.

On the other hand, machine learning and its extension *differentiable programming* offer great inspiration and techniques for programming quantum computers. Differentiable programming [18] composes many differentiable components to a learnable architecture and then learns the whole program by optimizing an objective function. The components are typically, but not limited to, neural networks. The word "differentiable" originates from the usual requirement of a gradient-based optimization scheme, which is

Xiu-Zhe Luo: rogerluo.rl18@gmail.com
Jin-Guo Liu: cacate0129@iphy.ac.cn

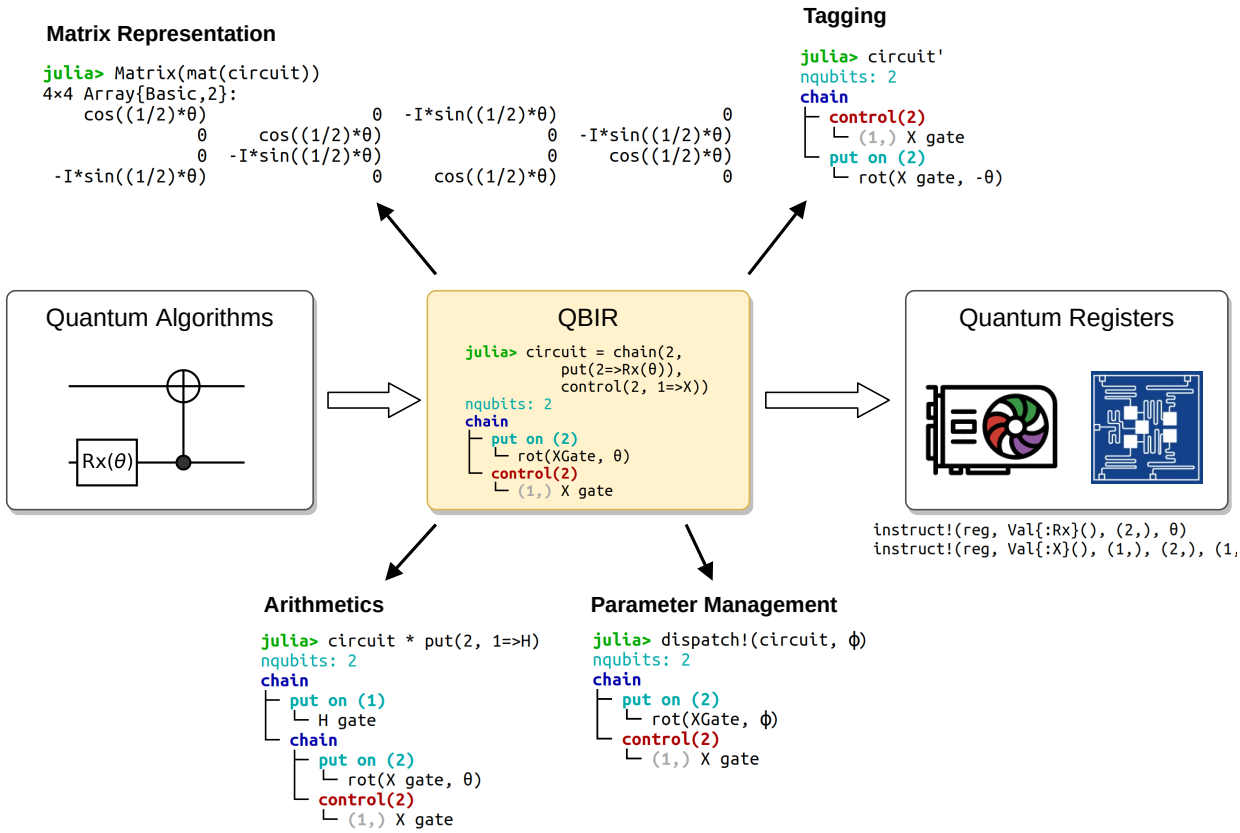


Figure 2: Quantum block intermediate representation plays a central role in Yao. The images of GPU and quantum circuits are taken from JuliaGPU [16] and IBM q-experience [17].

crucial for scaling up to high dimensional parameter spaces. Differentiable programming removes laborious human efforts and sometimes produces even better programs than humans can produce themselves [19].

Differentiable programming is a suitable paradigm for variational quantum algorithms, where parameters of quantum circuits are modified within a particular parameter space, to optimize a loss function. In this regard, programming quantum circuits in the differentiable paradigms address a much long term issue rather than the short-term considerations of compensating low-depth noisy quantum circuits with hybrid quantum-classical algorithms. Designing innovative and profitable quantum algorithms is, in general, nontrivial due to the lack of quantum intuitions. Fortunately, differentiable programming offers a new paradigm for devising novel quantum algorithms, much like what has already happened to the classical software landscape [19].

The algorithmic advances in differentiable programming hugely benefit from rapid development in software frameworks [20–25], among which the automatic differentiation (AD) of the computational graph is the key technique behind the scene. In order to evaluate gradients through the automatic differentiation process, these packages [20–25] create computational graphs in different ways.

It is instructive to view quantum circuits from the perspective of computational graphs with additional properties such as reversibility. In this regard, contextual analysis of the quantum computational graphs can be even more profitable than neural networks. For example, uncomputing (also known as adjoint or dagger), a sub-program plays a central role in reversible computing [26] since it returns qubit resources to the pool. While in differentiable programming of quantum circuits, exploiting the reversibility of the computational graph allows differentiating through the circuit with constant memory independent to its depth.

Inspired by differentiable programming software, we design `Yao` to be around the domain-specific computational graph, the quantum block

intermediate representation (QBIR). A block refers to a tensor representation of quantum operations, which can be quantum circuits and quantum operators of various granularities (quantum gates, Hamiltonian, or the whole program). As shown in Fig. 2, QBIR offers a hardware-agnostic abstraction of quantum circuits. It is called an intermediate representation due to its stage in the quantum compilation, which bridges the high-level quantum algorithms and low-level device-specific instructions. `Yao` provides rich functionalities to construct, inspect, manipulate, and differentiate quantum circuits in terms of QBIR.

What can `Yao` do ?

- Optimize a variational circuit with $10,000$ layers on a laptop, see Listing 9.
- Construct sparse matrix representation of 20 site Heisenberg Hamiltonian in approximately 5 seconds, see Listing 16.
- Play with Shor's 9 qubit error correction code symbolically, see Appendix C.
- Send your circuit or Hamiltonian to a remote host in the form of `YaoScript`, see Appendix D.
- Compile an arbitrary two-qubit unitary to a target circuit structure via gradient optimization, see Appendix H and `Yao`'s `QuAlgorithmZoo`.
- Solve ground state of a 6×6 lattice spin model with a tensor network inspired quantum circuit on GPU [27].

A distinct feature of `Yao` is its builtin automatic differentiation engine. Instead of building upon existing machine learning frameworks [20–25], we design `Yao`'s automatic differentiation engine to exploit reversibility in quantum computing, where the QBIR serves as a reversible computational graph. This implementation features speed and constant memory cost to circuit depth.

`Yao` dispatches QBIR to low-level instructions of quantum registers of various types (CPU, GPU, and QPU in the future). Extensions of `Yao` can be done straightforwardly by defining new QBIR nodes or quantum register types. As a bonus of the generic design, symbolic manipulation of quantum circuits in `Yao` follows almost for free. `Yao` achieves all these great flexibility and extensibility without sacrificing performance. `Yao` achieves one of the best performance for simulating quantum circuits of small to intermediate sizes, which are arguably the most relevant to quantum algorithms design for near-term devices.

`Yao` adds a unique solution to the landscape of open source quantum computing software includes `Quipper` [28], `ProjectQ` [29], `Q#` [30], `Cirq` [31], `qulacs` [32], `PennyLane` [33], `qiskit` [34], and `QuEST` [35]. References [36–38] contain more complete surveys of quantum software. Most software represents quantum circuits as a sequence of instructions. Thus, users need to define their abstraction for circuits with rich structures. `Yao` offers QBIR and related utilities to compose and manipulate complex quantum circuits. `Yao`'s QBIR is nothing but an abstract syntax tree, which is a commonly used data structure in modern programming languages thanks to its strong expressibility for control flows and hierarchical structures. `Quipper` [28] has adopted a similar strategy for the functional programming of quantum computing. `Yao` additionally introduces `Subroutine` to manage the scope of active and ancilla qubits. Besides these basic features, `Yao` puts a strong focus on differentiable programming of quantum circuits. In this regards, `Yao`'s batched quantum register with GPU acceleration and built-in AD engine offers significant speedup and convenience compared to `PennyLane` [33] and `qulacs` [32].

An overview for the ecosystems of `Yao`, ranging from the low level customized bit operations and linear algebra to high-level quantum algorithms, is provided in Figure 3. In Sec. 2 we introduce the quantum block intermediate representation; In Sec. 3 we explain the mechanism of the reversible computing and automatic differentiation in `Yao`; and the quantum register which stores hardware specific information about the quantum states in `Yao` are explained in Sec. 4. In Sec. 5, we compare performance of `Yao` against other frameworks to illustrate the excellent efficiency of `Yao`. The more important feature of `Yao`, its flexibility and extensibility of `Yao` are emphasized in Sec. 6. The applications of `Yao` and future directions for developing `Yao` are put in Sec. 7 and Sec. 8 respectively. Finally we summarize in Sec. 9. The Appendices (A-I) show various aspects and ver-

satile applications of `Yao`.

Why Julia ?

Julia is fast! Generic programming in Julia [39] helps `Yao` reach optimized performance while still keeping the code base general and concise. Benchmarks in Sec. 5 show that `Yao` reaches one of the best performances with generic codes written purely in Julia.

Julia codes can be highly extensible thanks to its type system and multiple dispatch mechanics. `Yao` builds its customized type system and dispatches to the quantum registers and circuits with a general interface. Moreover, Julia's meta-programming ability makes developing customized syntax and device-specific programs simple. Julia's dynamic and generic approach for GPU programming [16] powers `Yao`'s CUDA extension.

Julia plays well with other programming languages. It is generally straightforward to use external libraries written in other languages in `Yao`. For example, the symbolic backend of `Yao` builds on `SymEngine` written in C++. In Appendix A, we show an example of using the Python package `OpenFermion` [40] within `Yao`.

2 Quantum Block Intermediate Representation

The QBIR is a domain-specific abstract syntax tree for quantum operators, including circuits and observables. In this section, we introduce QBIR and its central role in `Yao` via concrete examples.

2.1 Representing Quantum Circuits

Figure 4 shows the quantum Fourier transformation circuit [41–43] which contains the `hcphases` blocks (marked in red) of different sizes. Each block itself is also a composition of Hadamard gates and `cphase` blocks (marked in blue) on various locations. In `Yao`, it takes three lines of code to construct the QBIR of the QFT circuit.

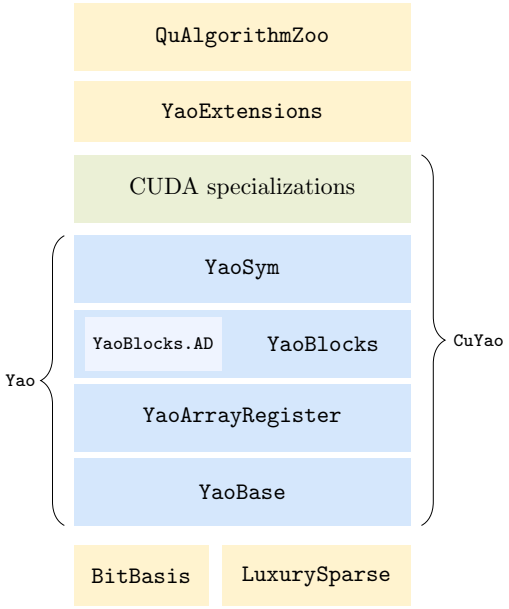


Figure 3: The packages in `Yao`'s ecosystem.

- **BitBasis** provides bitwise operations,
- **LuxurySparse** is an extension of Julia's builtin **SparseArrays**. It defines customized sparse matrix types and implements efficient operations relevant to quantum computation.
- **YaoBase** is an abstract package that defines basic abstract type for quantum registers and operations.
- **YaoArrayRegister** defines a register type as well as instruction sets,
- **YaoBlocks** defines utilities to construct and manipulate quantum circuits. It contains a builtin AD engine **YaoBlocks.AD** (Note: it is reexported in **Yao**, hence referred as **Yao.AD** in the main text).
- **YaoSym** provides symbolic computation supports.
- **Yao** is a meta package that re-export **YaoBase**, **YaoArrayRegister**, **YaoBlocks** and **YaoSym**.
- **CuYao** is a meta-package which contains **Yao** and provides specializations on CUDA devices.
- **YaoExtensions** provides utilities for constructing circuits and hamiltonians, faithful gradient estimator of quantum circuit, and some experimental features.
- **QuAlgorithmZoo** contains examples of quantum algorithms and applications.

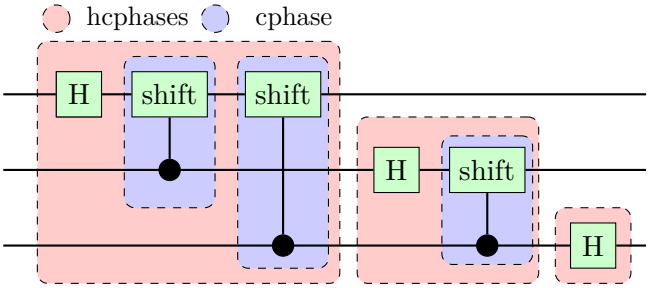


Figure 4: Quantum Fourier transformation circuit. The red and blue dashed blocks are built by the **hcphases** and **cphase** functions in the Listing 1.

Listing 1: quantum Fourier transform

```
julia> using Yao

julia> cphase(i, j) = control(i, j=> shift(
    2π/(2^(i-j+1))));

julia> hcphases(n, i) = chain(n, i==j ?
   put(i=>H) : cphase(j, i) for j in i:n);

julia> qft(n) = chain(hcphases(n, i)
                for i in 1:n)

julia> qft(3)
nqubits: 3
chain
├─ chain
│  ├─ put on (1)
│  │  └─ H gate
│  ├─ control(2)
│  │  └─ (1,) shift(1.5707963267948966)
│  └─ control(3)
│     └─ (1,) shift(0.7853981633974483)
├─ chain
│  ├─ put on (2)
│  │  └─ H gate
│  └─ control(3)
│     └─ (2,) shift(1.5707963267948966)
└─ chain
   └─ put on (3)
      └─ H gate
```

The function `cphase` defines a control phase shift gate with the `control` and `shift` functions. The function `hcphases` defines the recursive pattern in the QFT circuit, which puts a Hadamard gate in the first qubit of the subblock and then chains it with several control shift gates. Finally, one composes the QFT circuit of a given size by chaining the `hcphases` blocks. Overall, these codes construct a tree representation of the circuit shown in Fig. 5. The subtrees are composite blocks (`ChainBlock`, `ControlBlock`, and `PutBlock`) with different composition relations indicated in their roots. The leaves of the tree are primitive blocks. Appendix B shows the builtin block types of `Yao`, which are open to extension as shown in Appendix F.

In `Yao`, to execute a quantum circuit, one can simply feed a quantum state into the QBIR.

Listing 2: apply! and pipe

```
julia> rand_state(3) |> qft(3);
     # same as apply!(rand_state(3), qft(3))
```

Here, we define a random state on 3 qubits and pass it through the QFT circuit. The pipe operator `|>` is overloaded to call the `apply!` function which applies the quantum circuit block to the register and modifies the register **inplace**.

The generic implementation of QBIR in `Yao` allows supporting both numeric and symbolic data types. For example, one can inspect the matrix representation of quantum gates defined in `Yao` with symbolic variables.

Listing 3: inspecting gates

```
julia> @vars θ
(θ,)

julia> shift(θ) |> mat
2×2 LinearAlgebra.Diagonal
{Basic,Array{Basic,1}}:
 1          ⋅
 ⋅   exp(I*θ)

julia> control(2,1,2=>shift(θ)) |> mat
4×4 LinearAlgebra.Diagonal{Basic,
                    Array{Basic,1}}:
 1  ⋅  ⋅          ⋅
 ⋅  1  ⋅          ⋅
 ⋅  ⋅  1          ⋅
 ⋅  ⋅  ⋅   exp(I*θ)
```

Here, the `@vars` macro declares the symbolic variable θ. The `mat` function constructs the matrix representation of a quantum block. Appendix C shows another example of demonstrating Shor's 9 qubits code for quantum error correction with symbolic computation.

2.2 Manipulating Quantum Circuits

In essence, QBIR represents the algebraic operations of a quantum circuit as types. Being an

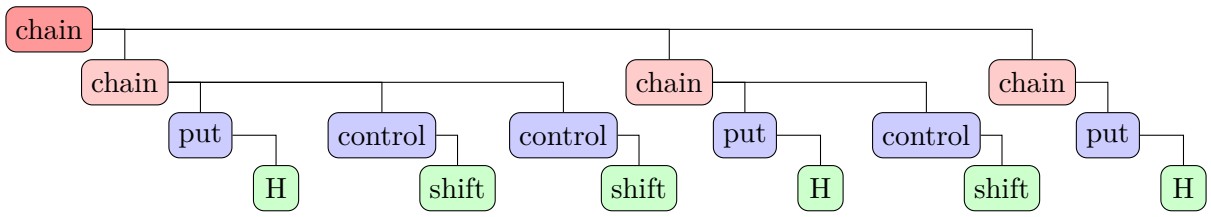


Figure 5: Quantum Fourier transformation circuit as a QBIR. The red nodes are roots of the composite **ChainBlock**. The blue nodes indicate the composite **ControlBlock** and **PutBlock**. Green nodes are primitive blocks.

algebraic data type system, QBIR automatically allows pattern matching with Julia's multiple dispatch mechanics. Pattern matching enables manipulating quantum circuits in their QBIR form in a straightforward manner.

Listing 4: gate decomposition

```
julia> decompose(x::HGate) =
    Rz(0.5π)*Rx(0.5π)*Rz(0.5π);

julia> decompose(x::AbstractBlock) =
    chsubblocks(x, decompose.(subblocks(x)));

julia> qft(3) |> decompose
nqubits: 3
chain
├─ chain
│  ├─ put on (1)
│  │  └─ chain
│  │     ├─ rot(ZGate, 1.5707963267948966)
│  │     ├─ rot(XGate, 1.5707963267948966)
│  │     └─ rot(ZGate, 1.5707963267948966)
│  ├─ control(2)
│  │  └─ (1,) shift(1.5707963267948966)
│  └─ control(3)
│     └─ (1,) shift(0.7853981633974483)
├─ chain
│  ├─ put on (2)
│  │  └─ chain
│  │     ├─ rot(ZGate, 1.5707963267948966)
│  │     ├─ rot(XGate, 1.5707963267948966)
│  │     └─ rot(ZGate, 1.5707963267948966)
│  └─ control(3)
│     └─ (2,) shift(1.5707963267948966)
└─ chain
   └─ put on (3)
      └─ chain
         ├─ rot(ZGate, 1.5707963267948966)
         ├─ rot(XGate, 1.5707963267948966)
         └─ rot(ZGate, 1.5707963267948966)
```

For example, consider a practical situation where one needs to decompose the Hadamard gate into three rotation gates. The codes in Listing 4 define compilation passes by dispatching the `decompose` function on different quantum block types. For the generic `AbstractBlock`, we apply `decompose` recursively to all its sub-blocks and use the function `chsubblocks` defined in `Yao` to substitute the blocks. The recursion terminates on primitive blocks where `subblocks` returns an empty set. Due to the specialization of `decompose` method on Hadamard gates, a chain of three rotation gates are returned as a subblock instead.

Listing 5: inverse QFT

```
julia> iqft(n) = qft(n)';

julia> iqft(3)
nqubits: 3
chain
├─ chain
│  └─ put on (3)
│     └─ H gate
├─ chain
│  ├─ control(3)
│  │  └─ (2,) shift(-1.5707963267948966)
│  └─ put on (2)
│     └─ H gate
└─ chain
   ├─ control(3)
   │  └─ (1,) shift(-0.7853981633974483)
   ├─ control(2)
   │  └─ (1,) shift(-1.5707963267948966)
   └─ put on (1)
      └─ H gate
```

Besides replacing gates, one can also modify a block by applying tags to it. For example, the `Daggered` tag takes the hermitian conjugate of the block. We use the `'` operator to apply the `Daggered` tag. Similar to the implementation of `Transpose` on matrices in Julia, the dagger operator in `Yao` is "lazy" in the sense that one simply marks the block as `Daggered` unless there are specific daggered rules defined for the block. For example, the hermitian conjugate of a `ChainBlock` reverses the order of its child nodes and propagate the `Daggered` tag to each subblock. Finally,

we have the following rules for primitive blocks,

- Hermitian gates are unchanged under dagger operation
- The hermitian conjugate of a rotational gate $R_\sigma(\theta) \to R_\sigma(-\theta)$
- Time evolution block $e^{-iHt} \to e^{-iH(-t^*)}$
- Some special constant gates are hermitian conjugate to each other, e.g. `T` and `Tdag`.

With these rules, we can define the inverse QFT circuit directly in Listing 5.

2.3 Matrix Representation

Quantum blocks have a matrix representations of different types for optimized performance. By default, the `apply!` method act quantum blocks to quantum registers using their matrix representations. The matrix representation is also useful for determining operator properties such as hermicity, unitarity, reflexivity, and commutativity. Lastly, one can also use `Yao`'s sparse matrix representation for quantum many-body computations such as exact diagonalization and (real and imaginary) time evolution.

For example, one can construct the Heisenberg Hamiltonian and obtain its ground state using the Krylov space solver in Listing 6. The arithmetic operations `*` and `sum` return `ChainBlock` and `Add` blocks respectively. It is worth noticing the differences between the QBIR arithmetic operations of the quantum circuits and those of Hamiltonians. Since the Hamiltonians are generators of quantum unitaries (i.e., $U = e^{-iHt}$), it is natural to perform additions for Hamiltonians (and other observables) and multiplications for unitaries. `YaoExtensions` provides some convenience functions for creating Hamiltonians on various lattices and variational quantum circuits.

Listing 6: Heisenberg Hamiltonian

```
julia> using KrylovKit: eigsolve

julia> bond(n, i) = sum([put(n, i=>σ) * put(
    n, i+1=>σ) for σ in (X, Y, Z)]);

julia> heisenberg(n) = sum([bond(n, i)
            for i in 1:n-1]);

julia> h = heisenberg(16);

julia> w, v = eigsolve(mat(h)
              ,1, :SR, ishermitian=true)
```

The `mat` function creates the sparse matrix representation of the Hamiltonian block. To achieve an optimized performance, we extend Julia's built-in sparse matrix types for various quantum gates. Appendix E lists these customized matrix types and promotion rules among them.

Time evolution under a quantum Hamiltonian invokes the Krylov space method [44], which repeatedly applies the Hamiltonian block to the register. In this case, one can use the `cache` tag to create a `CachedBlock` for the Hamiltonian. Then, the `apply!` method makes use of the sparse matrix representation cached in the memory. Continuing from Listing 6, the following codes in Listing 7 show that constructing and caching the matrix representation boosts the performance of time-evolution.

Listing 7: Hamiltonian evolution is faster with cache

```
julia> using BenchmarkTools

julia> te = time_evolve(h, 0.1);

julia> te_cache = time_evolve(cache(h), 0.1);

julia> @btime $(rand_state(16)) |> $te;
  1.042 s (10415 allocations: 1.32 GiB)

julia> @btime $(rand_state(16)) |> $te_cache;
  71.908 ms (10445 allocations: 61.48 MiB)
```

On the other hand, in many cases `Yao` can make use of efficient specifications of the `apply!` method for various blocks to apply them on the fly even without generating the matrix representation. The codes in Listing 8 show that this approach can be faster for simulating quantum circuits.

Listing 8: Circuit simulation is faster without cache

```
julia> r = rand_state(10);

julia> @btime r |> $(qft(10));
  550.466 μs (3269 allocations: 184.58 KiB)

julia> @btime r |> $(cache(qft(10)));
  1.688 ms (234 allocations: 30.02 KiB)
```

3 Reversible Computing and Automatic Differentiation

Automatic differentiation efficiently computes the gradient of a program. It is one of the essential techniques for the success of deep learning [45]. The technique is particularly relevant to differentiable programming of quantum circuits. In general, there are several modes of AD: the reverse mode caches intermediate state and then evaluate all gradients in a single backward run. The forward mode computes the loss and gradients in a single pass, which does not require caching intermediate state but has to evaluate the gradients of all parameters one by one.

`Yao`'s builtin reverse mode AD engine (Sec. 3.1) supports efficient classical simulation of variational quantum algorithms. By taking advantage of the reversible nature of quantum circuits, the memory complexity is reduced to constant compared to typical reverse mode AD [45]. This property allows one to simulate very deep variational quantum circuits. Besides, `Yao` supports the forward mode AD (Sec. 3.2), which is a faithful quantum simulation of the experimental situation. The complexity of forward mode is unfavorable compared to reverse mode because one needs to run the circuit repeatedly for each component of the gradient.

3.1 Reverse Mode: Builtin AD Engine with Reversible Computing

The submodule `Yao.AD` is a built-in AD engine. It back-propagates through quantum circuits using the computational graph information recorded in the QBIR.

In general, reverse mode AD needs to cache intermediate states in the forward pass for the back-propagation. Therefore, the memory consumption for backpropagating through a quantum simulator becomes unacceptable as the depth of the quantum circuit increases. Hence simply delegating AD to existing machine learning packages [20–25] is not a satisfiable solution. `Yao`'s customized AD engine exploits the inherent reversibility of quantum circuits [46, 47]. By uncomputing the intermediate state in the backward pass, `Yao.AD` mostly performs in-place operations without allocations. `Yao.AD`'s superior performance is in line with the recent efforts of implementing efficient backpropagation through reversible neural networks [47, 48].

In the forward pass we update the wave function $|\psi_k\rangle$ with inplace operations

$$\begin{aligned} &\cdots \\ |\psi_{k+1}\rangle &= U_k|\psi_k\rangle, \\ &\cdots \end{aligned} \tag{1}$$

where U_k is a unitary gate parametrized by θ_k. We define the adjoint of a variable as $\overline{x} = \frac{\partial \mathcal{L}}{\partial x^*}$ according to Wirtingers derivative [49] for complex numbers, where $\mathcal{L}$ is a real-valued objective function that depends on the final state. Starting from $\overline{\mathcal{L}} = 1$ we can obtain the adjoint of the output state.

To pull back the adjoints through the computational graph, we perform the backward calculation [50]

$$\begin{aligned} &\cdots \\ |\psi_k\rangle &= U_k^\dagger|\psi_{k+1}\rangle \\ \overline{|\psi_k\rangle} &= U_k^\dagger\overline{|\psi_{k+1}\rangle} \\ &\cdots \end{aligned} \tag{2}$$

The two equations above are implemented `Yao.AD` with the `apply_back!` method. Based on the obtained information, we can compute the adjoint of the gate matrix using [50]

$$\overline{U_k} = \overline{|\psi_{k+1}\rangle}\langle\psi_k|. \tag{3}$$

This outer product is not explicitly stored as a dense matrix. Instead, it is handled efficiently by customized low rank matrices described in Appendix E. Finally, we use `mat_back!` method to compute the adjoint of gate parameters $\overline{\theta_k}$ from the adjoint of the unitary matrix $\overline{U_k}$.

Figure 6 demonstrates the procedure in a concrete example. The black arrows show the forward pass without any allocation except for the

output state and the objective function $\mathcal{L}$. In the backward pass, we uncompute the states (blue arrows) and backpropagate the adjoints (red arrows) at the same time. For the block defined as `put(nbit, i=>chain(Rz(`α`), Rx(`β`), Rx(`γ`)))`, we obtain the desired $\overline{\alpha}$, $\overline{\beta}$ and $\overline{\gamma}$ by pushing the adjoints back through the `mat` functions of `PutBlock` and `ChainBlock`. The implementation of the AD engine is generic so that it works automatically with symbolic computation. We show an example of calculating the symbolic derivative of gate parameters in Appendix G. One can also integrate `Yao.AD` with classical automatic differentiation engines such as `Zygote` to handle mixed classical and quantum computational graphs, see [51].

Listing 9: 10000-layer VQE

```
julia> using Yao, YaoExtensions

julia> n = 10; depth = 10000;

julia> circuit = dispatch!(
    variational_circuit(n, depth),
    :random);

julia> gatecount(circuit)
Dict{Type{#s54} where #s54 <:
    AbstractBlock,Int64} with 3 entries:
  RotationGate{1,Float64,ZGate} => 200000
  RotationGate{1,Float64,XGate} => 100010
  ControlBlock{10,XGate,1,1}    => 100000

julia> nparameters(circuit)
300010

julia> h = heisenberg(n);

julia> for i = 1:100
    _, grad = expect'(h, zero_state(n)=>
                                circuit)
    dispatch!(-, circuit, 1e-3 * grad)
    println("Step $i, energy = $(expect(
            h, zero_state(10)=>circuit))")
       end
```

To demonstrate the efficiency of `Yao`'s AD engine, we use the codes in Listing 9 to simulate the variational quantum eigensolver (VQE) [52] with depth 10,000 (with 300,010 variational parameters) on a laptop. The simulation would be extremely challenging without `Yao`, either due to overwhelming memory consumption in the reverse mode AD or unfavorable computation cost in the forward mode AD.

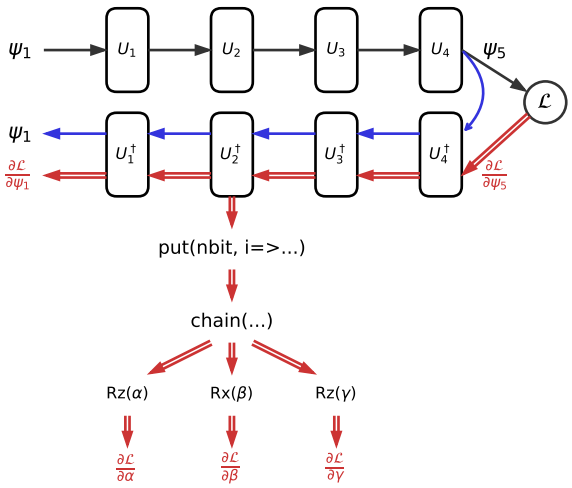


Figure 6: Builtin automatic differentiation engine **Yao.AD**. Black arrows represent the forward pass. The blue arrow represents uncomputing. The red arrows indicate the backpropagation of the adjoints.

Here, `variational_circuit` is predefined in `YaoExtensions` to have a hardware efficient architecture [53]. The `dispatch!` function with the second parameter specified to `:random` gives random initial parameters. The `expect` function evaluates expectation values of the observables; the second argument can be a wave function or a pair of the input wave function and circuit ansatz like above. `expect'` evaluates the gradient of this observable for the input wave function and circuit parameters. Here, we only make use of its second return value. For batched registers, the gradients of circuit parameters are accumulated rather than returning a batch of gradients. `dispatch!(-, circuit, ...)` implements the gradient descent algorithm with energy as the loss function. The first argument is a binary operator that computes a new parameter based on the old parameter in `c` and the third argument, the gradients. Parameters in a circuit can be extracted by calling `parameters(circuit)`, which collects parameters into a vector by visiting the QBIR in depth-first order. The same parameter visiting order is used in `dispatch!`. In case one would like to share parameters in the variational circuit, one can simply use the same block instance in the QBIR. In the training process, gradients can be updated in the same field. After the training, the circuit is fully optimized and returns the ground state of the model Hamiltonian with zero state as input.

3.2 Forward Mode: Faithful Quantum Gradients

Compared to the reverse mode, forward mode AD is more closely related to how one measures the gradient in the actual experiment.

The implementation of the forward mode AD is particularly simple for the "rotation gates" $R_\Sigma(\theta) \equiv e^{-i\Sigma\theta/2}$ with the generator Σ being both hermitian and reflexive ($\Sigma^2 = 1$). For example, Σ can be the Pauli gates X, Y and Z, or multi-qubit gates such as CNOT, CZ, and SWAP. Many other non-reflexive gates can also be decomposed into reflexive gates via gate transformation [54]. Under these conditions, the gradient to a circuit parameter is [6, 55–57]

$$\frac{\partial\langle O\rangle_\theta}{\partial\theta} = \frac{1}{2}\left(\langle O\rangle_{\theta+\frac{\pi}{2}} - \langle O\rangle_{\theta-\frac{\pi}{2}}\right) \tag{4}$$

where $\langle O\rangle_\theta$ denotes the expectation of the observable O with the given parameter θ. Therefore, one just needs to run the simulator twice to estimate the gradient. `YaoExtensions` implements Eq. (4) with Julia's broadcasting semantics and obtains the full gradients with respect to all parameters. Similar features can be found in `PennyLane` [33] and `qulacs` [32]. We refer this approach as the *faithful gradient*, since it mirrors the experimental procedure on a real quantum device. In this way, one can estimate the gradients in the VQE example Listing 9 using Eq. (4)

```
# this will be slow
julia> grad = faithful_grad(h, zero_state(n)
                =>circuit; nshots=100);
```

where one faithfully simulates `nshots` projective measurements. In the default setting `nshots=nothing`, the function evaluates the exact expectation on the quantum state. Note that simulating projective measurement, in general, involves diagonalizing the observable operator and rotating to its eigenbasis. `Yao` implements an efficient way to break the measurement into the expectation of local terms automatically.

The above observation Eq. (4) can also be generalized to statistic functional loss, which is useful for generative modeling with an implicit probability distribution given by the quantum circuits [9]. The symmetric statistic functional of order two reads

$$\mathcal{F}_\theta = \mathop{\mathbb{E}}_{x\sim p_\theta, y\sim p_\theta}[K(x,y)], \tag{5}$$

where K is a symmetric function, p_θ is the output probability distribution of a parametrized quantum circuit measured on the computational basis. If the circuit is parametrized by rotation gates, the gradient of the statistic functional is

$$\begin{aligned}\frac{\partial\mathcal{F}_\theta}{\partial\theta} = &\mathop{\mathbb{E}}_{x\sim p_{\theta+\frac{\pi}{2}}, y\sim p_\theta}[K(x,y)] \\ &- \mathop{\mathbb{E}}_{x\sim p_{\theta-\frac{\pi}{2}}, y\sim p_\theta}[K(x,y)],\end{aligned} \tag{6}$$

which is also related to the measure valued gradient estimator for stochastic optimization [58]. Within this formalism, `Yao` provides the following interfaces to evaluate gradients with respect to the maximum mean discrepancy loss [59, 60], which measures the probabilistic distance between two sets of samples.

Listing 10: gradient of maximum mean discrepancy

```
julia> target_p = normalize!(rand(1<<5));

julia> kf = brbf_kernel(2.0);

julia> circuit = variational_circuit(5);

julia> mmd = MMD(kf, target_p);

julia> g_reg, g_params = expect'(
    mmd, zero_state(5)=>circuit);

julia> g_params = faithful_grad(
    mmd, zero_state(5)=>circuit);
```

4 Quantum Registers

The quantum register stores hardware-specific information about the quantum states. In classical simulation on a CPU, the quantum register is an array containing the quantum wave function. For GPU simulations, the quantum register stores the pointer to a GPU array. In an actual experiment, the register should be the quantum device that hosts the quantum state. `Yao` handles all of these cases with a unified `apply!` interface, which dispatches the instructions depending on different types of QBIR nodes and registers.

4.1 Instructions on Quantum Registers

Quantum registers store quantum states in contiguous memory, which can either be the CPU memory or other hardware memory, such as a CUDA device.

Listing 11: CUDA register

```
julia> using CuYao

# construct the |1010> state
julia> r = ArrayReg(bit"1010");

# transfer data to CUDA
julia> r = cu(r);
```

Each register type has its own device-specific instruction set. They are declared in `Yao` via the "instruction set" interface, which includes

- **gate instruction**: `instruct!`
- **measure instruction**: `measure` and `measure!`
- **qubit management instructions**: `focus!` and `relax!`

The instruction interface provides a clean way to extend supports on various hardwares without worrying about variations in the high level.

Listing 12: instruct! and measure

```
julia> r = zero_state(4);

julia> instruct!(r, Val(:X), (2, ))
ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 4/4

julia> samples = measure(r; nshots=3)
3-element Array{BitBasis.BitStr{4,Int64},1}:
 0010 (2)
 0010 (2)
 0010 (2)

julia> [samples[1]...]
4-element Array{Int64,1}:
 0
 1
 0
 0
```

For example, the rotation gate shown in Fig. 2 is interpreted as `instruct!(reg, Val(:Rx), (2,), θ)`. The second parameter specifies the gate, which is a `Val` type with a gate symbol as a type parameter. The third parameter is the qubit to apply, and the fourth parameter is the rotation angle. The CNOT gate is interpreted as `instruct!(reg, Val(:X), (1,), (2,), (1,))`, where the last three tuples are gate locations, control qubits, and value of the control qubits, respectively. The `measure` function simulates measurement from the quantum register and provides bit strings, while `measure!` returns the bit string and also collapse the state.

In the last line of the above example, we convert a bit string $0010_{(2)}$ to a vector `[0, 1, 0, 0]`. Note that the order is reversed since the readout of a bit string is in the little-endian format.

4.2 Active qubits and environment qubits

In certain quantum algorithms, one only applies the circuit block to a subset of qubits. For example, see the quantum phase estimation [61] shown in Fig. 7.

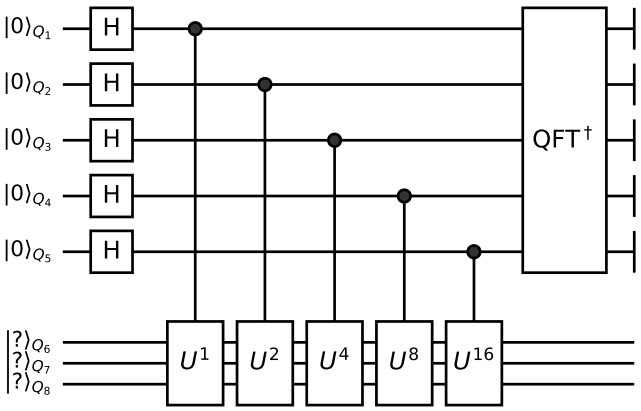


Figure 7: 5-qubit quantum Phase estimation circuit. This circuit contains three components. First, apply Hadamard gates to n ancilla qubits. Then apply controlled unitary to $n+m$ qubits and finally apply inverse QFT to n ancilla qubits.

The QFT circuit block defined in Listing 1 can not be used directly in this case since the block size does not match the number of qubits. We introduce the concept of active and environment qubits to address this issue. Only the active qubits are visible to circuit blocks under operation. We manage the qubit resources with the `focus!` and its reverse `relax!` instructions. For example, if we want to apply the QFT algorithm on qubits 3,6,1 and 2, the `focus!` activates the

four qubits 3,6,1,2 in the given order and deactivates the rest

Listing 13: focus! and relax!

```
julia> reg = rand_state(10)

julia> focus!(reg, (3,6,1,2))

julia> reg |> qft(4)

julia> relax!(reg, (3,6,1,2); to_nactive=10)
```

Since it is a recurring pattern to first `focus!`, then `relax!` on the same qubits in many quantum algorithms, we introduce a `Subroutine` node to manage the scope automatically. Hence, the phase estimation circuit in Fig. 7 can be defined with the following codes.

Listing 14: quantum phase estimation

```
PE(n, m, U) = chain(
    n+m, # total number of qubits
    repeat(H, 1:n), # apply H from 1:n
    chain(control(
        k,
        n+1:n+m=>matblock(U^(2^(k-1))))
        for k in 1:n
    ),

    # apply inverse QFT on a local scope
    subroutine(qft(n)', 1:n)
)
```

The `matblock` method in the codes constructs a quantum circuit from a given unitary matrix.

4.3 Batched Quantum Registers

The batched register is a collection of quantum wave functions. It can be samples of classical data for quantum machine learning tasks [62] or an ensemble of pure quantum states for thermal state simulation [51]. For both applications, having the batch dimension not only provides convenience but may also significantly speed up the simulations.

We adopt the Single Program Multiple Data (SPMD) [63] design in `Yao` similar to modern machine learning frameworks so that it can make use of modern multi-processors such as multi-threading or GPU support (and potentially multi-processor QPUs). Applying a quantum circuit to a batched register means to apply the same quantum circuit to a batch of wave functions in parallel, which is extremely friendly to modern multi-processors.

The memory layout of the quantum register is a matrix of the size $2^a \times 2^r B$, where a is the number of system qubits, r is the number of remaining qubits (or environment qubits), B is the batch size. For gates acting on the active qubits, the remaining qubits and batch dimension can be treated on an equal footing. We put the batch dimension as the last dimension because Julia array is column majored. As the last dimension, it favors broadcasting on the batch dimensions.

One can construct a batched register in `Yao` and perform operations on it. These operations are automatically broadcasted over the batch dimension.

Listing 15: a batch of quantum registers

```
julia> reg = rand_state(4; nbatch=5);

julia> reg |> qft(4) |> measure!
5-element Array{BitBasis.BitStr{4,Int64},1}:
 1011 (2)
 1011 (2)
 0000 (2)
 1101 (2)
 0111 (2)
```

Note that we have used the `measure!` function to collapse all batches.

The measurement results are represented in `BitStr` type which is a subtype of `Integer` and has a static length. Here, it pretty-prints the measurement results and provides a convenient readout of measuring results.

5 Performance

As introduced above, `Yao` features a generic and extensible implementation without sacrificing performance. Our performance optimization strategy heavily relies on Julia's multiple dispatch. As a bottom line, `Yao` implements a general multi-control multi-qubit arbitrary-location gate instruction as the fallback. We then fine-tune various specifications for better performance. Therefore, in many applications, the construction and operation of QBIR do not even invoke matrix allocation. While in cases where the

gate matrix is small (number of qubits smaller than 4), `Yao` automatically employs the corresponding static sized types [64] for better performance. The sparse matrices `IMatrix`, `Diagonal`, `PermMatrix` and `SparseMatrixCSC` introduced in Appendix E also have their static version defined in `LuxurySparse.jl` [65]. Besides, we also utilize unique structures of frequently used gates and dispatch to specialized implementations. For example, Pauli X gate can be executed by swapping the elements in the register directly.

We benchmark `Yao`'s performance with other quantum computing software. Note that the exact classical simulation of the generic quantum circuit is doomed to be exponential [66–69]. `Yao`'s design puts a strong emphasis on the performance of small to intermediate-sized quantum circuits since the high-performance simulation of such circuits is crucial for the design of near-term algorithms that run repeatedly or in parallel.

5.1 Experimental Setup

Package	Language	Version
`Cirq` [31]	Python	0.6.0
`qiskit` [34]	C++/Python	0.13.0
`qulacs` [32]	C++/Python	0.1.8
`PennyLane` [33]	Python	0.7.0
`QuEST` [35]	C/Python	0.1.1
`ProjectQ` [29]	C++/Python	0.4.2
`Yao`	Julia	0.6.0
`CuYao`	Julia	0.1.3

Table 1: Packages in the benchmark.

Although `QuEST` is a package originally written in C, we benchmark it in Python via `pyquest-cffi` [70] for convenience. `Pennylane` is benchmarked with its default backend [71]. Since the package was designed primarily for being run on the quantum hardware, its benchmarks contain a certain overhead that was not present in other frameworks [72]. `qiskit` is benchmarked with `qiskit-aer` 0.2.3 [73] and `qiskit-terra` 0.8.2 [74].

Our test machine contains an Intel(R) Xeon(R) Gold 6230 CPU with a Tesla V100 GPU accelerator. SIMD is enabled with **AVX2** instruction set. The benchmark time is measured via `pytest-benchmark` [75] and `BenchmarkTools` [76] with minimum running time. We ignore the compilation time in Julia since one can always get rid of such time by compiling the program ahead of time. The benchmark scripts and complete reports are maintained online at the repository [77].

Software	Version
Python	3.7.3
Numpy	1.16.3
MKL	2019.3
Julia	1.3.0

Table 2: The environment setup of the machine for benchmark.

5.2 Single Gate Performance

We benchmark several frequently used quantum gates, including the Pauli-X Gate, the Hadamard gate (H), the controlled-NOT gate (CNOT), and the Toffoli Gate. These benchmarks directly measure the raw performance of gate instructions.

Figure 8 shows the running times of various gates in each package in the unit of nano seconds. One can see that `Yao`, `ProjectQ`, and `qulacs` reach similar performance when the number of qubits $n > 20$. They are at least several times faster than other packages. Having similar performance in these three packages suggests that they all reached the top performance for this type of full amplitude classical simulation on CPU.

The overhead of simulating small to intermediate-sized circuits is particularly relevant for designing variational quantum algorithms where the same circuit may be executed million times during training. `Yao` shows the least overhead in these benchmarks. `qulacs` also did an excellent job of suppressing these overheads.

5.3 Parametrized Quantum Circuit Performance

Next, we benchmark the parameterized circuit of depth $d = 10$ shown in Fig. 9(a). This type of hardware-efficient circuits was employed in the

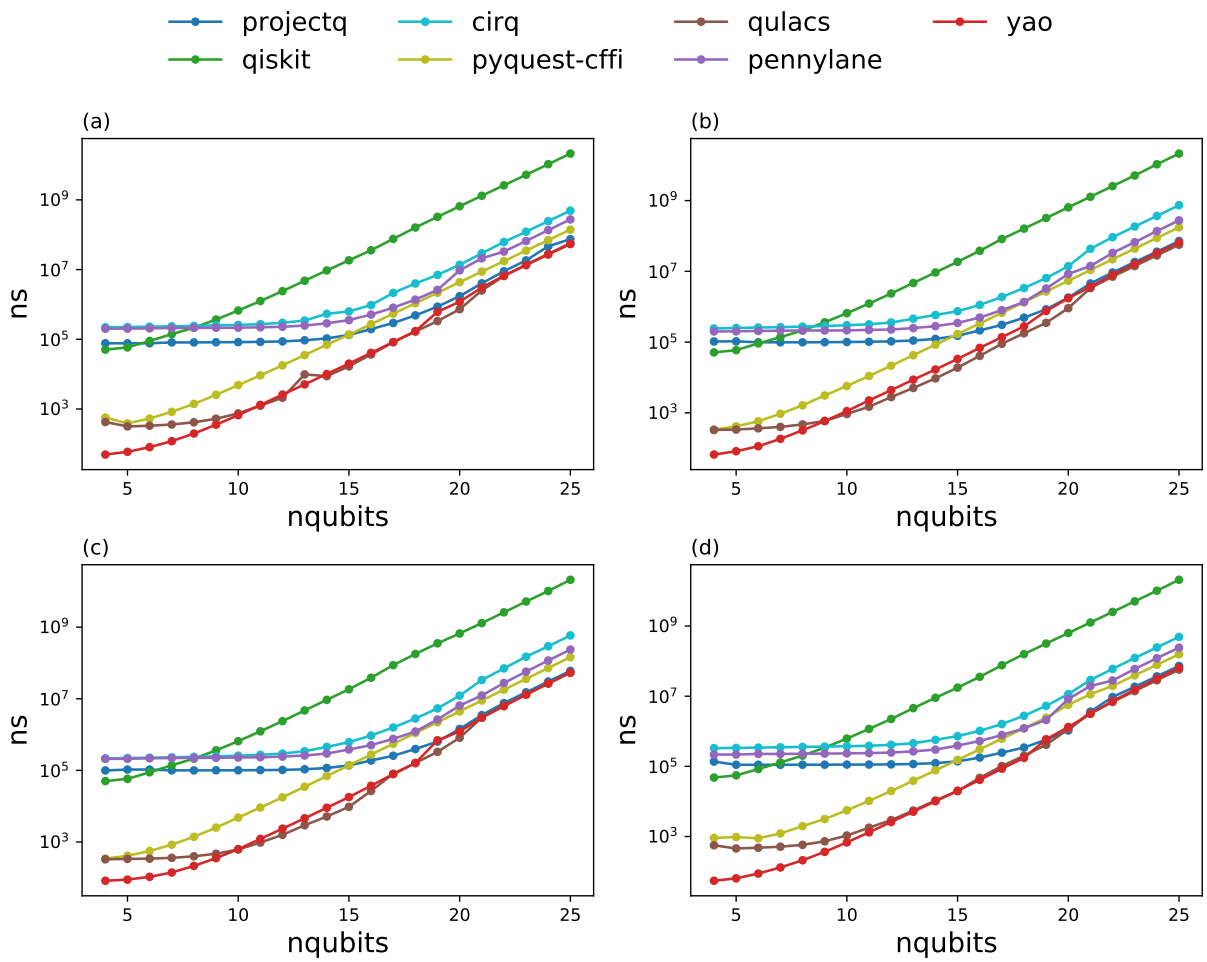


Figure 8: Benchmarks of (a) Pauli-X gate; (b) Hadamard gate; (c) CNOT gate; (d) Toffolli gate.

VQE experiment [53]. The `qiskit` package is excluded in this benchmark due to a lack of rotation gate support in its builtin state vector simulator backend. These benchmarks further test the performance of circuit abstraction in practical applications.

The results in Fig. 9(b) shows that `Yao` reaches the best performance for more than 10 qubits on CPU. `qulacs`'s well tuned C++ simulator is faster than `Yao` for fewer qubits. On a CUDA device, `Yao` and `qulacs` show similar performance. These benchmarks also, show that CUDA parallelization starts to be beneficial for a qubit number larger than 16. Overall, `Yao` is one of the fastest quantum circuit simulators for this type of application.

Lastly, we benchmark the performance of batched quantum register introduced in Sec 4.3 in Fig. 9(c) with a batch size 1000. We only measure `Yao`'s performance due to the lack of native support of SPMD in other quantum simulation frameworks. `Yao`'s CUDA backend (labeled as `yao (cuda)`) offers large speed up (>10x) compared to the CPU backend (labeled as `yao`). For reference, we also plot the timing of a bare loop over the batch dimension on a CPU (labeled as `yao` × `1000`). One can see that batching offers substantial speedup for small circuits.

5.4 Matrix Representation and Automatic Differentiation Performance

As discussed in Sec. 2.3 and Sec. 3.1, `Yao` features highly optimized matrix representation and reverse mode automatic differentiation for the QBIR. We did not attempt a systematic benchmark due to the lack of similar features in other quantum software frameworks.

Here, we simply show the timings of constructing the sparse matrix representation of 20 site Heisenberg Hamiltonian and differentiating its energy expectation through a variational quantum circuit of depth 20 (200 parameters) on a lap-

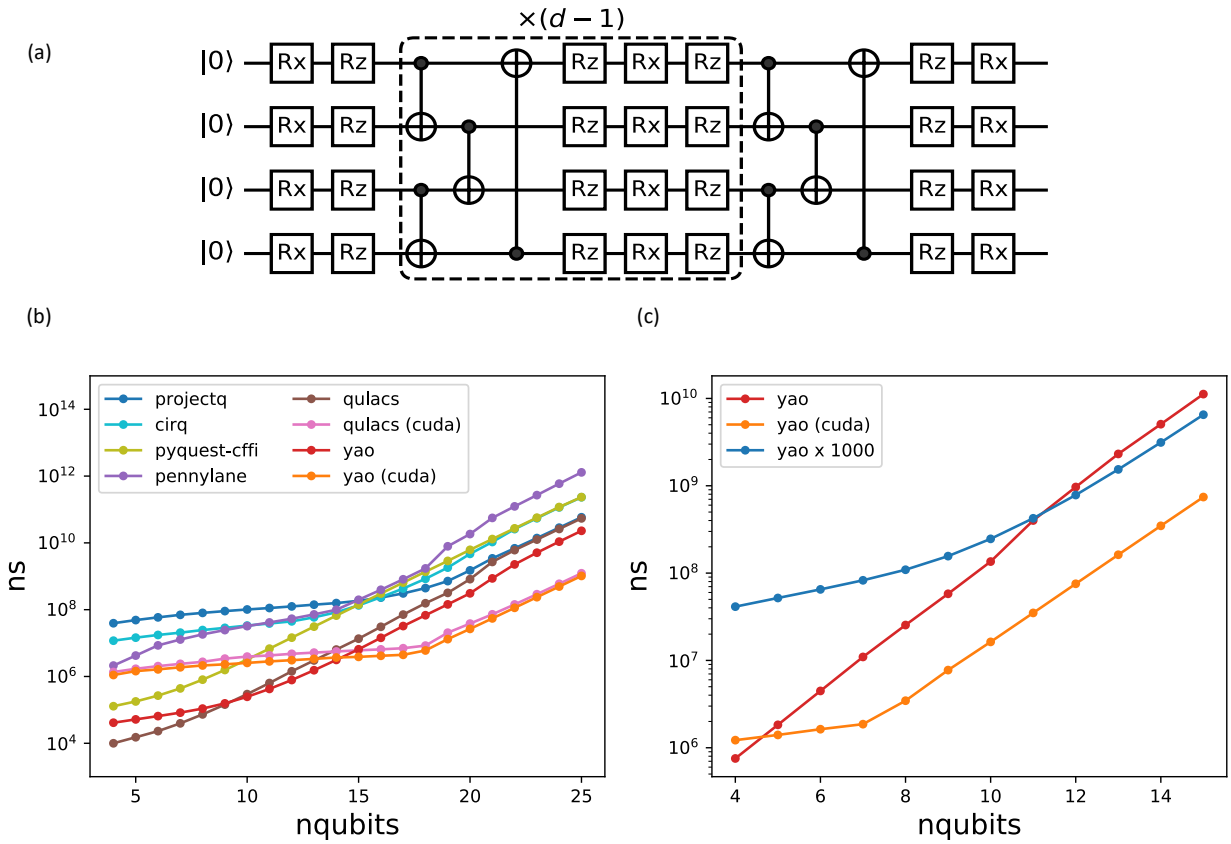


Figure 9: (a) A parameterized quantum circuit with single qubit rotation and CNOT gates; (b) Benchmarks of the parameterized circuit; (c) Benchmarks of the parametrized circuit, the batched version. Line "yao" represents the batched registers, "yao (cuda)" represents the batched register on GPU, "yao × 1000" is running on a non-batched register repeatedly for 1000 times.

top. The forward mode AD discussed in Sec. 3.2 is slower by order of a hundred in such simulations.

Listing 16: benchmark mat and AD performance

```
julia> using BenchmarkTools, Yao,
                    YaoExtensions

julia> @btime mat($(heisenberg(20)));
  6.330 s (10806 allocations: 10.34 GiB)

julia> @btime expect'($(heisenberg(20)),
                      $(zero_state(20))=>
                $(variational_circuit(20)));
  5.054 s (58273 allocations: 4.97 GiB)
```

6 Extensibility

In the previous section we have demonstrated the excellent efficiency of `Yao` in comparison to other frameworks. We nevertheless emphasize that the most important feature of `Yao` is its flexibility and extensibility.

6.1 Extending QBIR nodes

It is easy to extend `Yao` with new gates and quantum block nodes. One can define constant gates by giving its matrix representation. For example, the `FSim` gate that appears in Google supremacy experiments [78] is composed of an `ISWAP` gate and a `cphase` gate with a fixed angle.

Listing 17: FSim gate

```
julia> using Yao, LuxurySparse

julia> @const_gate ISWAP = PermMatrix(
        [1,3,2,4], [1,1.0im,1.0im,1])

# FSim is already defined in YaoExtensions
julia> @const_gate MyFSim = mat((ISWAP*
        control(2, 2, 1=>shift(-π/6)))')

julia> put(10, (4,2)=>MyFSim)
nqubits: 10
put on (4, 2)
└─ MyFSim gate
```

The macro `@const_gate` defines a primitive gate that subtypes `ConstantGate` abstract type. It generates global gate instances `ISWAP` and `MyFSim` as well as new gate types `MyFSimGate` and `ISWAPGate` for dispatch. This macro also binds operators properties such as `ishermitian`, `isreflexive` and `isunitary` by inspecting the given matrix representation.

In Appendix F, we provide a more sophisticated example of extending QBIR.

6.2 Extending the Quantum Register

A new register type can be defined by dispatching the "instruction set" interfaces introduced in Sec. 4.1. For example, in the CUDA backend `CuYao` [79] we dispatch `instruct!` and `measure!` to the `ArrayReg{B,T,<:CuArray}` type. Here besides the batch number `B` and data type `T`, the third type parameter `<:CuArray` specifies the storage type. The dispatch directs the instructions to the CUDA kernels written in `CUDAnative` [16], which significantly boosts the performance by parallelizing both for the batch and the Hilbert space dimensions and. A comparison between the batched or single register of the parameterized circuit is shown in Sec 5. We provide more detailed examples in the developer's guide of Appendix I.

7 Applications

The `Yao` framework has been employed in practical research projects during its development and has evolved with the requirements of research.

`Yao` simplifies the original implementation of Quantum Circuit Born machine [9] originally written in `ProjectQ` [29] from about 200 lines of code to less than 50 lines of code with about 1000x performance improvement as shown in our benchmark Fig. 9. This simplification enabled further exploration of the algorithm in [80] with the much simpler codebase.

The tensor network inspired quantum circuits described in [27, 62] allows the study of large systems with a reduced number of qubits. For example, one can solve the ground state of a 6×6 frustrated Heisenberg lattice model with only 12 qubits. These circuits can also compress a quantum state to the hardware with fewer qubits [81]. These applications need to measure and reuse qubits in the circuits. Thus, one can not take `nshots` measurements to the wavefunction. Instead, one constructs a batched quantum register with `nbatch` states and samples bitstrings in parallel. `Yao`'s SPMD friendly design and CUDA backend significantly simplifies the implementation and boosts performance.

Automatic differentiation can also be used for gate learning. It is well-known that a general two-qubit gate can be compiled to a fixed gate instruction set that includes single-qubit gates and CNOT gates [82]. Given a circuit structure, one can approximate an arbitrary U(4) unitary (up to a global phase) instantly by gradient optimization of the operator fidelity. The code can be found in Appendix H.

A recent project extending VQE [52] to thermal quantum states [51] integrates `Yao` seamlessly with the differentiable programming package `Zygote` [83]. `Yao`'s efficient AD engine and batched quantum register support allow joint training of quantum circuit and classical neural network effortlessly.

8 Roadmap

8.1 Hardware Control

Hardware control is one of `Yao`'s next major missions. In principle, `Yao` already has a lightweight tool `YaoScript` (Appendix D) to serialize QBIR to the data stream for dump/load and internet communication, which can be used to control devices on the cloud.

For even better integration with existing protocols, we plan to support the parsing and code generation for other low level languages such as

OpenQASM [84], eQASM [85] and Quil [86]. As an ongoing project towards this end, we and the author of the general-purpose parser generator `RBNF` [87] developed a QASM parser and codegen in `YaoQASM` [88]. It allows one to parse the OpenQASM [84] to QBIR and dump QBIR to OpenQASM, which can be used for controlling devices in the IBM Q Experience [17, 29].

8.2 Circuit Compilation and Optimization

The next edition of QBIR [89] should be more compilation friendly with a design based on Julia's native abstract syntax tree. Better integration with Julia compiler will also avoid overheads in the parameter management and location reordering of the `focus!` function. With better support on the compilation, the next edition of QBIR will be even more useful for quantum circuit simplification and optimization, which are crucial for reducing the cost of both simulations and experiments. The new QBIR will also support better pattern matching and term rewriting system, which allows smarter and more systematic circuit simplifications such as the ones in Refs. [90, 91].

8.3 Tensor Networks Backend

Quantum circuits are special cases of tensor networks with the unitary constraints on the gates. Thus, tensor network algorithms developed in the quantum many-body community are potentially useful for simulating quantum circuits, especially the shallow ones with a large number of qubits where the state vector does not fit into memory [92–94]. In this sense, one can perform more efficient simulations at a larger scale by exploring low-rank structures in the tensor networks with a trade-off of almost negligible errors [95].

Also, to serve as an efficient simulation backend, the tensor networks are helpful for quantum machine learning, given the fact that they have already found many applications in various machine learning tasks [96–100]. As an example, one can envision, training a unitary tensor network with classical algorithms and then load it to an actual quantum device for fast sampling. In this regard, quantum devices become specialized inference hardware.

The ongoing project `YaoTensorNetwork` [101] provides utilities to dump a quantum circuit into a tensor network format. There are already working examples of generating tensor networks for QFT, variational circuits [11], and for demonstrating the quantum supremacy on random circuits [78, 102]. The dumped tensor networks can be further used for quantum circuit simplification [103] and quantum circuit simulation based on exact or approximated tensor network contractions.

9 Summary

We have introduced `Yao`, an open source Julia package for quantum algorithm design. `Yao` features

- differentiable programming quantum circuits with a built-in AD engine leveraging reversible computing,
- batched quantum registers with CUDA parallelization,
- symbolic manipulation of quantum circuits,
- strong extensibility,
- top performance for relevant applications.

The quantum block abstraction of the quantum circuit is central to these features. Generic programming, which in Julia is based on the type system and multiple dispatch, is key to the extensibility and efficiency of `Yao`. Along with the roadmap Sec. 8, `Yao` is evolving towards an even more versatile framework for quantum computing research and development.

10 Acknowledgement

Thanks to Jin Zhu for the Chinese calligraphy of `Yao`'s logo. The QASM compilation was greatly aided by Taine Zhao's effort on the Julia parser generator `RBNF.jl`, we appreciate his help and insightful discussion about compilation. This work owes much to enlightening conversation and help from open source community including: Tim Besard and Valentin Churavy for their work on the CUDA transpiler `CUDAnative` and suggestions on our CUDA backend implementation, Mike Innes and Harrison Grodin for their helpful discussion about automatic differentiation and symbolic manipulation, Juan Gomez, Damian Steiger,

Damian Steiger, Craig Gidney, corryvrequan, Johannes Jakob Meyer and Nathan Killoran for reviewing the performance benchmarks [77]. We thank Divyanshu Gupta for integrating `Yao` with `DifferentialEquations.jl` [104], Wei-Shi Wang, Yi-Hong Zhang, Tong Liu, Yu-Kun Zhang, and Si-Rui Lu for being beta users and offering valuable suggestions. We thank Roger Melko for helpful suggestions of this manuscript, Hao Xie and Arthur Pesah for proofreading this manuscript. The first author would like to thank Roger Melko, Miles Stoudenmire, Xi Xiong for their kindly help on his Ph.D. visa issue during the development. `Yao`'s development is supported by the National Natural Science Foundation of China under the Grant No. 11774398, No. 11747601 and No. 11975294, the Ministry of Science and Technology of China under the Grant No. 2016YFA0300603 and No. 2016YFA0302400, the Strategic Priority Research Program of Chinese Academy of Sciences Grant No. XDB28000000, and Huawei Technologies under Grant No. YBN2018095185.

References

[1] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[2] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5:4213, 2014.

[3] Dave Wecker, Matthew B Hastings, and Matthias Troyer. Progress towards practical quantum variational algorithms. *Physical Review A*, 92(4):042303, 2015.

[4] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.

[5] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[6] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum circuit learning. *Physical Review A*, 98(3): 032309, 2018.

[7] Vojtěch Havlíček, Antonio D Córcoles, Kristan Temme, Aram W Harrow, Abhinav Kandala, Jerry M Chow, and Jay M Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567 (7747):209, 2019.

[8] Marcello Benedetti, Delfina Garcia-Pintos, Oscar Perdomo, Vicente Leyton-Ortega, Yunseong Nam, and Alejandro Perdomo-Ortiz. A generative modeling approach for benchmarking and training shallow quantum circuits. *npj Quantum Information*, 5(1), May 2019. ISSN 2056-6387. DOI: 10.1038/s41534-019-0157-8. URL `http://dx.doi.org/10.1038/s41534-019-0157-8`.

[9] Jin-Guo Liu and Lei Wang. Differentiable learning of quantum circuit born machines. *Physical Review A*, 98(6):062324, 2018.

[10] Peter JJ O'Malley et al. Scalable quantum simulation of molecular energies. *Physical Review X*, 6(3):031007, 2016.

[11] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242, 2017. URL `https://www.nature.com/articles/nature23879`.

[12] Daiwei Zhu et al. Training of quantum circuits on a hybrid quantum computer. *Science Advances*, 5(10):eaaw9918, 2019.

[13] G. Pagano, A. Bapat, P. Becker, K. S. Collins, A. De, P. W. Hess, H. B. Kaplan, A. Kyprianidis, W. L. Tan, C Baldwin, L T Brady, A Deshpande, F Liu, S Jordan, A V Gorshkov, and C Monroe. Quantum Approximate Optimization with a Trapped-Ion Quantum Simulator. 2019. URL `http://arxiv.org/abs/1906.02700`.

[14] Vicente Leyton-Ortega, Alejandro Perdomo-Ortiz, and Oscar Perdomo. Robust implementation of generative modeling with parametrized quantum circuits. *arXiv preprint arXiv:1901.08047*, 2019.

[15] Jarrod R. McClean, Sergio Boixo, Vadim N. Smelyanskiy, Ryan Babbush, and Hartmut Neven. Barren plateaus in quantum neural network training landscapes. *Nat. Commun.*, 9(1):4812, 2018. ISSN 2041-

1723. DOI: 10.1038/s41467-018-07090-4. URL https://doi.org/10.1038/s41467-018-07090-4.

[16] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing julia on gpus. *CoRR*, abs/1712.03112, 2017. URL http://arxiv.org/abs/1712.03112.

[17] Guillermo García-Pérez, Matteo A. C. Rossi, and Sabrina Maniscalco. Ibm q experience as a versatile experimental testbed for simulating open quantum systems, 2019. URL https://arxiv.org/abs/1906.07099.

[18] Differentiable Programming. https://en.wikipedia.org/wiki/Differentiable_programming, .

[19] Karpathy, Andrej. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35.

[20] Tianqi Chen et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[21] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[23] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

[24] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018. URL http://arxiv.org/abs/1811.01457.

[25] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckus, Elliot Saba, Viral B Shah, and Will Tebbutt. Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

[26] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, Nov 1973. ISSN 0018-8646. DOI: 10.1147/rd.176.0525.

[27] Jin-Guo Liu, Yi-Hong Zhang, Yuan Wan, and Lei Wang. Variational quantum eigensolver with fewer qubits. *Phys. Rev. Research*, 1:023025, Sep 2019. DOI: 10.1103/PhysRevResearch.1.023025. URL https://link.aps.org/doi/10.1103/PhysRevResearch.1.023025.

[28] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Notices*, volume 48, pages 333–342. ACM, 2013.

[29] Damian S Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *arXiv preprint arXiv:1612.08091*, 2016.

[30] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. Q#: Enabling scalable quantum computing and development with a high-level dsl. *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*, 2018. DOI: 10.1145/3183895.3183901. URL http://dx.doi.org/10.1145/3183895.3183901.

[31] Cirq: A Python framework for creating, editing, and invoking noisy intermediate scale quantum (NISQ) circuits. https://github.com/quantumlib/Cirq.

[32] qulacs: Variational Quantum Circuit Simulator for Quantum Computation Research. https://github.com/qulacs/qulacs.

[33] Ville Bergholm, Josh Izaac, Maria Schuld,

Christian Gogolin, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[34] Héctor Abraham et al. Qiskit: An open-source framework for quantum computing, 2019.

[35] Tyson Jones, Anna Brown, Ian Bush, and Simon C. Benjamin. Quest and high performance simulation of quantum computers. *Scientific Reports*, 9(1), Jul 2019. ISSN 2045-2322. DOI: 10.1038/s41598-019-47174-9. URL http://dx.doi.org/10.1038/s41598-019-47174-9.

[36] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. Open source software in quantum computing. *PloS one*, 13(12):e0208561, 2018.

[37] Ryan LaRose. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum*, 3:130, March 2019. ISSN 2521-327X. DOI: 10.22331/q-2019-03-25-130. URL https://doi.org/10.22331/q-2019-03-25-130.

[38] Marcello Benedetti, Erika Lloyd, Stefan Sack, and Mattia Fiorentini. Parameterized quantum circuits as machine learning models. *Quantum Science and Technology*, 4(4):043001, nov 2019. DOI: 10.1088/2058-9565/ab4eb5. URL https://doi.org/10.1088%2F2058-9565%2Fab4eb5.

[39] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[40] Jarrod R. McClean et al. Openfermion: The electronic structure package for quantum computers, 2017.

[41] D Coppersmith. An approximate fourier transform useful in quantum computing. Technical report, Technical report, IBM Research Division, 1994.

[42] Artur Ekert and Richard Jozsa. Quantum computation and shor's factoring algorithm. *Reviews of Modern Physics*, 68(3): 733, 1996.

[43] Richard Jozsa. Quantum algorithms and the fourier transform. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):323–337, 1998.

[44] Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5 (1), 2017. DOI: 10.5334/jors.151. URL https://app.dimensions.ai/details/publication/pub.1085583166andhttp://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/. Exported from https://app.dimensions.ai on 2019/05/05.

[45] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153), 2018.

[46] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[47] Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems*, pages 2214–2224, 2017.

[48] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.

[49] Akira Hirose. *Complex-valued neural networks: theories and applications*, volume 5. World Scientific, 2003.

[50] Mike Giles. An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation. Technical report, 2008. URL https://people.maths.ox.ac.uk/gilesm/files/NA-08-01.pdf.

[51] Jin-Guo Liu, Liang Mao, Pan Zhang, and Lei Wang. Solving quantum statistical mechanics with variational autoregressive networks and quantum circuits. 2019. URL http://arxiv.org/abs/1912.XXXXX.

[52] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigen-

value solver on a photonic quantum processor. *Nat. Commun.*, 5:4213, 2014. URL https://www.nature.com/articles/ncomms5213.

[53] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242, 2017.

[54] Gavin E Crooks. Gradients of parameterized quantum gates using the parameter-shift rule and gate decomposition. URL https://arxiv.org/abs/1905.13311.

[55] Jun Li, Xiaodong Yang, Xinhua Peng, and Chang-Pu Sun. Hybrid quantum-classical approach to quantum optimal control. *Phys. Rev. Lett.*, 118:150503, Apr 2017. URL https://link.aps.org/doi/10.1103/PhysRevLett.118.150503.

[56] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. Evaluating analytic gradients on quantum hardware. *Phys. Rev. A*, 99(3):032331, 2019. ISSN 24699934. DOI: 10.1103/PhysRevA.99.032331.

[57] Ken M Nakanishi, Keisuke Fujii, and Synge Todo. Sequential minimal optimization for quantum-classical hybrid algorithms. URL https://arxiv.org/abs/1903.12166.

[58] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte carlo gradient estimation in machine learning. URL https://arxiv.org/abs/1906.10652.

[59] Chun-Liang Li, Wei-Cheng Chang, Yu Cheng, Yiming Yang, and Barnabás Póczos. MMD GAN: Towards Deeper Understanding of Moment Matching Network. URL http://arxiv.org/abs/1705.08584.

[60] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012. URL http://www.jmlr.org/papers/v13/gretton12a.html.

[61] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information*. Cambridge university press, 2010.

[62] William James Huggins, Piyush Patil, Bradley Mitchell, K Birgitta Whaley, and Miles Stoudenmire. Towards quantum machine learning with tensor networks. *Quantum Science and Technology*, 2018.

[63] Frederica Darema, David A George, V Alan Norton, and Gregory F Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.

[64] Statically sized arrays for Julia. https://github.com/JuliaArrays/StaticArrays.jl.

[65] A luxury sparse matrix package for julia. https://github.com/QuantumBFS/LuxurySparse.jl.

[66] 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*, page 33. ACM, 2017.

[67] Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

[68] Edwin Pednault, John A Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, and Robert Wisnieff. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867*, 2017.

[69] Fang Zhang et al. Alibaba cloud quantum development kit: Large-scale classical simulation of quantum circuits. *arXiv preprint arXiv:1907.11217*, 2019.

[70] Pyquest-cffi: A python interface to the quest quantum simulator (cffi based). https://github.com/HQSquantumsimulations/PyQuEST-cffi.

[71] PennyLane is a cross-platform Python library for quantum machine learning, automatic differentiation, and optimization of hybrid quantum-classical computations. https://github.com/XanaduAI/pennylane, .

[72] Review of PennyLane benchmark. https://github.com/Roger-luo/quantum-benchmarks/pull/7, .

[73] Aer is a high performance simulator for quantum circuits that includes noise models. https://github.com/Qiskit/qiskit-aer, .

[74] Terra provides the foundations for Qiskit. It allows the user to write quantum circuits easily, and takes care of the constraints of real hardware. https://github.com/Qiskit/qiskit-terra, .

[75] py.test fixture for benchmarking code. https://github.com/ionelmc/pytest-benchmark.

[76] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *arXiv preprint arXiv:1608.04295*, 2016.

[77] Benchmarking Quantum Circuit Emulators For Your Daily Research Usage. https://github.com/Roger-luo/quantum-benchmarks.

[78] Frank Arute et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[79] CuYao.jl: CUDA extension for Yao.jl. https://github.com/QuantumBFS/CuYao.jl.

[80] Jinfeng Zeng, Yufeng Wu, Jin-Guo Liu, Lei Wang, and Jiangping Hu. Learning and inference on generative adversarial quantum circuits. *Physical Review A*, 99(5):052306, 2019.

[81] Weishi Wang, Jin-Guo Liu, and Lei Wang. A variational quantum state compression algorithm. *to appear*.

[82] Vivek V. Shende, Igor L. Markov, and Stephen S. Bullock. Minimal universal two-qubit controlled-not-based circuits. *Phys. Rev. A*, 69:062321, Jun 2004. DOI: 10.1103/PhysRevA.69.062321. URL https://link.aps.org/doi/10.1103/PhysRevA.69.062321.

[83] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018. URL http://arxiv.org/abs/1810.07951.

[84] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[85] Xiang Fu et al. eqasm: An executable quantum instruction set architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 224–237. IEEE, 2019.

[86] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[87] RBNF: A DSL for modern parsing. https://github.com/thautwarm/RBNF.jl.

[88] Bidirectional transformation between Yao Quantum Block IR and QASM. https://github.com/QuantumBFS/YaoQASM.jl, .

[89] YaoIR: Intermediate Representation for Quantum Programs. https://github.com/QuantumBFS/YaoIR.jl, .

[90] Raban Iten, David Sutter, and Stefan Woerner. Efficient template matching in quantum circuits. *arXiv preprint arXiv:1909.05270*, 2019.

[91] Dmitri Maslov, Gerhard W Dueck, D Michael Miller, and Camille Negrevergne. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, 2008.

[92] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, and Hartmut Neven. Simulation of low-depth quantum circuits as complex undirected graphical models. *arXiv preprint arXiv:1712.05384*, 2017.

[93] Jianxin Chen, Fang Zhang, Mingcheng Chen, Cupjin Huang, Michael Newman, and Yaoyun Shi. Classical simulation of intermediate-size quantum circuits. *arXiv preprint arXiv:1805.01450*, 2018.

[94] Chu Guo, Yong Liu, Min Xiong, Shichuan Xue, Xiang Fu, Anqi Huang, Xiaogang Qiang, Ping Xu, Junhua Liu, Shenggen Zheng, He-Liang Huang, Mingtang Deng, Dario Poletti, Wan-Su Bao, and Junjie Wu. General-purpose quantum circuit simulator with projected entangled-pair states and the quantum supremacy frontier. *Phys. Rev. Lett.*, 123:190501, Nov 2019. DOI: 10.1103/PhysRevLett.123.190501. URL https://link.aps.org/doi/10.1103/PhysRevLett.123.190501.

[95] Feng Pan, Pengfei Zhou, Sujie Li, and Pan Zhang. Contracting arbitrary tensor networks: general approximate algorithm and applications in graphical models and quan-

tum circuit simulations. *arXiv preprint arXiv:1912.03014*, 2019.

[96] E Miles Stoudenmire and David J Schwab. Supervised learning with quantum-inspired tensor networks. *arXiv preprint arXiv:1605.05775*, 2016.

[97] Zhao-Yu Han, Jun Wang, Heng Fan, Lei Wang, and Pan Zhang. Unsupervised generative modeling using matrix product states. *Phys. Rev. X*, 8:031012, Jul 2018. DOI: 10.1103/PhysRevX.8.031012. URL https://link.aps.org/doi/10.1103/PhysRevX.8.031012.

[98] Song Cheng, Lei Wang, Tao Xiang, and Pan Zhang. Tree tensor networks for generative modeling. *Phys. Rev. B*, 99:155131, Apr 2019. DOI: 10.1103/PhysRevB.99.155131. URL https://link.aps.org/doi/10.1103/PhysRevB.99.155131.

[99] Ivan Glasser, Ryan Sweke, Nicola Pancotti, Jens Eisert, and Ignacio Cirac. Expressive power of tensor-network factorizations for probabilistic modeling. In *Advances in Neural Information Processing Systems*, pages 1496–1508, 2019.

[100] Tai-Danae Bradley, E Miles Stoudenmire, and John Terilla. Modeling sequences with quantum states: A look under the hood. *arXiv preprint arXiv:1910.07425*, 2019.

[101] YaoTensorNetwork: Dump a quantum circuit in Yao to a tensor network graph model. https://github.com/QuantumBFS/YaoTensorNetwork.jl, .

[102] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595, 2018.

[103] Miriam Backens. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9): 093021, sep 2014. DOI: 10.1088/1367-2630/16/9/093021. URL https://doi.org/10.1088%2F1367-2630%2F16%2F9%2F093021.

[104] Multi-language suite for high-performance solvers of differential equations. https://github.com/JuliaDiffEq/DifferentialEquations.jl, .

[105] General Permutation Matrix. https://en.wikipedia.org/wiki/Generalized_permutation_matrix.

[106] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, and Matthias Troyer. High performance emulation of quantum circuits. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 866–874. IEEE, 2016.

[107] Ryan LaRose, Arkin Tikku, Étude O'Neel-Judy, Lukasz Cincio, and Patrick J. Coles. Variational quantum state diagonalization. *npj Quantum Information*, 5(1), Jun 2019. ISSN 2056-6387. DOI: 10.1038/s41534-019-0167-6. URL http://dx.doi.org/10.1038/s41534-019-0167-6.

[108] Cristina Cirstoiu, Zoe Holmes, Joseph Iosue, Lukasz Cincio, Patrick J. Coles, and Andrew Sornborger. Variational fast forwarding for quantum simulation beyond the coherence time, 2019.

[109] Lukasz Cincio, Yiğit Subaşı, Andrew T Sornborger, and Patrick J Coles. Learning the quantum algorithm for state overlap. *New Journal of Physics*, 20(11): 113022, Nov 2018. ISSN 1367-2630. DOI: 10.1088/1367-2630/aae94a. URL http://dx.doi.org/10.1088/1367-2630/aae94a.

[110] Xiaoguang Wang, Zhe Sun, and Z. D. Wang. Operator fidelity susceptibility: An indicator of quantum criticality. *Physical Review A*, 79(1), Jan 2009. ISSN 1094-1622. DOI: 10.1103/physreva.79.012105. URL http://dx.doi.org/10.1103/PhysRevA.79.012105.

[111] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995. URL https://doi.org/10.1137/0916069.

[112] Patrick Kofod Mogensen and Asbjørn Nilsen Riseth. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24):615, 2018. DOI: 10.21105/joss.00615.

[113] Piotr Gawron, Dariusz Kurzyk, and Łukasz Pawela. Quantuminformation.jl—a julia package for numerical computation in quantum information theory. *PLOS ONE*,

13(12):e0209358, Dec 2018. ISSN 1932-6203. DOI: 10.1371/journal.pone.0209358. URL http://dx.doi.org/10.1371/journal.pone.0209358.

A Using external libraries

It is straightforward to make use of external libraries written in other languages. For example, the following codes import modules from the `OpenFermion` [40] and construct a molecule Hamiltonian in Julia.

```
using PyCall

of_hamil = pyimport("openfermion.hamiltonians")
of_trsfm = pyimport("openfermion.transforms")
of_pyscf = pyimport("openfermionpyscf")

diatomic_bond_length = 1.0
geometry = [("H", (0., 0., 0.)), ("H", (0., 0., diatomic_bond_length))]
basis = "sto-3g"
multiplicity = 1
charge = 0
description = string(diatomic_bond_length)

molecule = of_hamil.MolecularData(geometry, basis, multiplicity, charge, description)
molecule = of_pyscf.run_pyscf(molecule,run_scf=1,run_fci=1)

m_h = molecule.get_molecular_hamiltonian()
nbits = m_h.n_qubits
jw_h = of_trsfm.jordan_wigner(of_trsfm.get_fermion_operator(m_h))
```

Up to here, the codes follow the `OpenFermion`'s tutorial closely. Next, we construct a quantum block representation of the molecule Hamiltonian. One can then use the Hamiltonian block as wish in `Yao`, such as exact diagonalization (Listing 6), time evolution (Listing 7) or VQE (Listing 9).

```
julia> using Yao
julia> function yao_hamiltonian(nbits, jw_h)
          gates = Dict("X"=>X, "Y"=>Y, "Z"=>Z)
          h = Add{nbits}()
          for (k, v) in jw_h.terms
             push!(h, v*kron(nbits, [site+1 => gates[opname] for (site, opname) in k]...))
          end
          return h
        end

julia> yao_hamiltonian(nbits, jw_h)
+
├─ [scale: 0.10622904490856075 + 0.0im] kron
│      ├─ 2=>Z
│      └─ 4=>Z
├─ [scale: -0.04919764587136755 + 0.0im] kron
│      ├─ 1=>X
│      ├─ 2=>X
│      ├─ 3=>Y
│      └─ 4=>Y
.
.
.
└─ [scale: 0.13716572937099503 + 0.0im] kron
       └─ 2=>Z
```

B Builtin Block Types

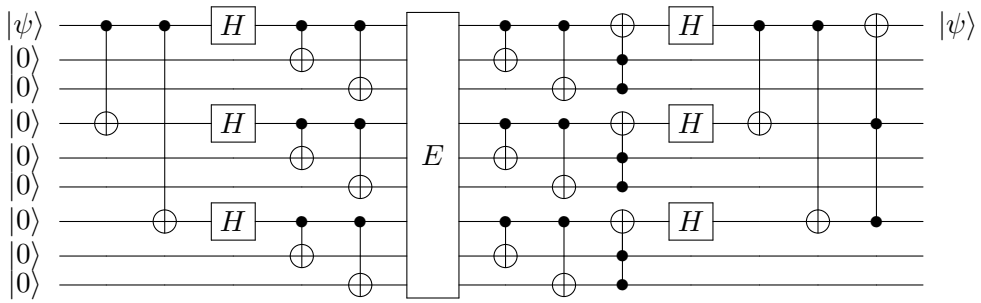
One can inspect the builtin block types in `Yao` with the following codes.

```
julia> using Yao, AbstractTrees

julia> AbstractTrees.children(x::Type) = subtypes(x)

julia> AbstractTrees.print_tree(AbstractBlock)
AbstractBlock
├─ CompositeBlock
│  ├─ AbstractContainer
│  │  ├─ ControlBlock
│  │  ├─ PutBlock
│  │  ├─ RepeatedBlock
│  │  ├─ Subroutine
│  │  └─ TagBlock{BT,N} where N where BT<:AbstractBlock
│  │     ├─ CachedBlock{ST,BT,N} where N where BT<:AbstractBlock where ST
│  │     ├─ Daggered
│  │     ├─ Scale
│  │     └─ NoParams
│  ├─ Add
│  ├─ ChainBlock
│  ├─ KronBlock
│  └─ UnitaryChannel
└─ PrimitiveBlock
   ├─ ConstantGate
   │  ├─ HGate
   │  ├─ I2Gate
   │  ├─ SWAPGate
   │  ├─ TGate
   │  ├─ XGate
   │  ├─ YGate
   │  ├─ CNOTGate
   │  ├─ CZGate
   │  ├─ IGate
   │  ├─ P0Gate
   │  ├─ P1Gate
   │  ├─ PdGate
   │  ├─ PuGate
   │  ├─ SGate
   │  ├─ SdagGate
   │  ├─ TdagGate
   │  ├─ ToffoliGate
   │  └─ ZGate
   ├─ GeneralMatrixBlock
   ├─ Measure
   ├─ PhaseGate
   ├─ RotationGate
   ├─ ShiftGate
   ├─ TimeEvolution
   └─ TrivialGate
      └─ IdentityGate
```

More blocks, such as `SqrtX` and `FSimGate`, are avaiblable in `YaoExtensions`.

C Symbolic Computation: Shor's 9 qubit code

The well-known Shor's 9 qubit code can correct any single-qubit error E and restores the state $|\psi\rangle$.



The circuit can be constructed by the following code.

```
using Yao

shor(E) = chain(9,
    # encode circuit
    cnot(1, 4), cnot(1, 7),
    put(1=>H), put(4=>H), put(7=>H),
    cnot(1,2), cnot(1,3), cnot(4,5), cnot(4,6), cnot(7,8), cnot(7,9),

    E, # the error

    # decode circuit
    cnot(1,2), cnot(1,3), cnot((2, 3), 1),
    cnot(4,5), cnot(4,6), cnot((5, 6), 4),
    cnot(7,8), cnot(7,9), cnot((8, 9), 7),

    put(1=>H), put(4=>H), put(7=>H),
    cnot(1, 4), cnot(1, 7), cnot((4, 7), 1)
    )
```

Now we can check whether it can correct a given error by doing symbolic computation on an arbitrary 1-qubit pure quantum state $\alpha|0\rangle + \beta|1\rangle$ and a specific weight-9 error.

```
julia> @vars α β
(α, β)

julia> s = α * ket"0" + β * ket"1" |> addbits!(8)
α|000000000> + β|000000001>

julia> E = kron(1=>X, 2=>Z, 3=>Z, 4=>X, 5=>Z, 6=>Z, 7=>X, 8=>Z, 9=>Z);

julia> s |> shor(E) |> partial_tr(2:9) |> expand
α|0> + β|1>
```

Yeah!

D Yao Script

We introduce `YaoScript` as an alternating way to define quantum circuits. We provide a Julia non-standard string literal `@yao_str` for circuit parsing, which can be invoked through `yao"..."`. This macro reads and parses a string to a QBIR. With `YaoScript`, one can dump (load) a circuit to (from) a file or internet data stream. The `YaoScript` appears like native Julia code but will be parsed by the macro to prevent code injection.

The error correction circuit used in Appendix C can be defined as

```
using Yao
c = yao"""let nqubits=9, version="0.6.0"
    begin # encode circuit
        1=>C, 4=>X
        1=>C, 7=>X
        1=>H, 4=>H, 7=>H
        1=>C, 2=>X
        1=>C, 3=>X
        4=>C, 5=>X
        4=>C, 6=>X
        7=>C, 8=>X
        7=>C, 9=>X
    end

    # the error
    1=>X, 2=>Z, 3=>Z, 4=>X, 5=>Z, 6=>Z, 7=>X, 8=>Z, 9=>Z

    begin # decode circuit
        1=>C, 2=>X
        1=>C, 3=>X
        2=>C, 3=>C, 1=>X
        4=>C, 5=>X
        4=>C, 6=>X
        5=>C, 6=>C, 4=>X
        7=>C, 8=>X
        7=>C, 9=>X
        8=>C, 9=>C, 7=>X

        1=>H, 4=>H, 7=>H
        1=>C, 4=>X
        1=>C, 7=>X
        4=>C, 7=>C, 1=>X
    end
end"""
```

In this script, it is easy to find the following correspondance with QBIR

Script	**Block**
1=>C, 2=>X	`control(9, 1, 2=>X)`
1=>H, 4=>H, 7=>H	`kron(9, 1=>H, 4=>H, 7=>H)`
1=>H	`put(9, 1=>H)`
beigin ... end	`chain(9, [...])`

Table 3: The correspondence between `YaoScript` and QBIR.

E Matrix Types in Yao

Table 4 summaries the matrix types used for the basic quantum gates.

Gate	Matrix Type
I2	`IMatrix`
Z, T, S, Rz	`Diagonal`
X, Y, CNOT, CZ, SWAP	`PermMatrix`
P0, P1, Pu, Pd, PSwap	`SparseMatrixCSC`
H, Rx, Ry	`Matrix`

Table 4: Matrix types of gates in `Yao`.

The `SparseMatrixCSC` type is provided in Julia's builtin `SparseArrays`. The identity matrix `IMatrix` and general permutation matrix `PermMatrix` [105] are defined in `LuxurySparse.jl` [65]. The `PermMatrix` allows having values other than one in the non-zero entries. For example, the matrix of ISWAP gate defined in Sec. 6.1 of the main text

```
julia> PermMatrix([1,3,2,4], [1,1.0im,1.0im,1])
4×4 PermMatrix{Complex{Float64},Int64,Array{Complex{Float64},1},Array{Int64,1}}:
 1.0+0.0im      0           0           0
    0           0        0.0+1.0im      0
    0        0.0+1.0im      0           0
    0           0           0        1.0+0.0im
```

where the first argument represents the column indices and the second argument the entries.

These type specifications for quantum gates allow fast arithmetics. Table 5 lists the type conversion under matrix multiplication, Kronecker product, and addition operations.

	I	D	P	S	M
I	I/I/D/I	D/D/D/D	P/P/S/D	S/S/S/D	M/S/M/D
D	D/D/D/D	D/D/D/D	P/P/S/D	S/S/S/D	M/S/M/D
P	P/P/S/D	P/P/S/D	P/P/S/P	S/S/S/P	M/S/M/P
S	S/S/S/D	S/S/S/D	S/S/S/P	S/S/S/S	M/S/M/S
M	M/S/M/D	M/S/M/D	M/S/M/P	M/S/M/S	M/S/M/M

Table 5: Matrix types conversion under matrix multiplication (*)/kronecker product (kron)/addition (+)/hadamard product (`.*`). Here I, D, P, S, M stands for `IMatrix`, `Diagonal`, `PermMatrix`, `SpasreMatrixCSC` and `Matrix` respectively.

Besides these specialised sparse matrices, `Yao.AD` uses low rank matrix types for backpropagation, c.f. Eq. (3) in the main texts. For this we define the `OuterProduct` matrix type for both memory and computation efficiency.

F Extending QBIR: emulating QFT circuit as an example

In most cases, one can build quantum circuits using basic blocks in `YaoBlocks` such as `put`, `control` in the Listing 1. In certain cases, one needs to define a new QBIR node, e.g., to dispatch a more specialized simulation method. For example, one can emulate QFT by using highly optimized FFT on classical computers [106]. Since the nodes are just normal Julia type, it is straightforward to do this in `Yao`. One can define a new block type by subtyping from the primitive block. In principle, the only thing to make it work is to define its matrix representation.

```
using Yao

struct QFT{N} <: PrimitiveBlock{N} end
QFT(n::Int) = QFT{n}()
Yao.mat(::Type{T}, x::QFT) where T = mat(T, qft(nqubits(x)))
```

where `qft(n)` can be the function defined in Listing 1. In this way, one can wrap an implementation of the QFT as a primitive block.

However, to emulate QFT in a more efficient way, we can overload the `apply!` method using the classical inverse FFT for the QFT circuit.

```
using FFTW, LinearAlgebra, BitBasis

function Yao.apply!(r::ArrayReg, x::QFT)
    α = sqrt(length(statevec(r)))
    invorder!(r)
    lmul!(α, ifft!(statevec(r)))
    return r
end
```

With this minimal definition, `Yao` is able to make use of builtin methods to infer its properties such as `ishermitian`, `isunitary`, `isreflexive` and `iscommute` and other functionalities. For example, the inverse QFT can be simply obtained by `QFT(4)'`. After the extension, the `QFT` type will appear in the type tree shown in Appendix B under the `PrimitiveBlock`.

Both the faithful simulation approach introduced in the main text and this classical emulation have been included in **YaoExtensions** as `qft_circuit` (for faithful classical simulation) and `QFT` (for FFT emulation).

G Symbolic Differentiation

The following program computes the gradient of a Heisenberg Hamiltonian with respect to circuit parameters symbolically and analytically with `Yao`'s AD engine.

```
julia> using Yao, YaoExtensions

julia> @vars α β γ
(α, β, γ)

julia> circuit = chain(put(3, 2=>Rx(α)), control(3, 2, 1=>Ry(β)), put(3, (1,2)=>rot(kron(X, X), γ)))
nqubits: 3
chain
├─ put on (2)
│  └─ rot(XGate, α)
├─ control(2)
│  └─ (1,) rot(YGate, β)
└─ put on (1, 2)
   └─ rot(KronBlock{2,XGate}, γ)

julia> h = heisenberg(3);

julia> energy_symbolic = expect(h, zero_state(Basic, 3)=>circuit)
-(-I*sin((1/2)*γ)*cos((1/2)*α) - ... - sin((1/2)*γ)*sin((1/2)*β)*sin((1/2)*α))^2

julia> grad_symbolic = expect'(h, zero_state(Basic, 3)=>circuit).second
3-element Array{Basic,1}:
 -((sin((1/2)*β)*(-sin((1/2)*γ)* ... + cos((1/2)*γ)^2*cos((1/2)*β)*sin((1/2)*α))))
 2*((-1/2)*sin((1/2)*β)*(I*sin((1/2)*γ)* ... + I*cos((1/2)*γ)*sin((1/2)*β)*sin((1/2)*α)))))
 -((cos((1/2)*γ)*cos((1/2)*α) - ... - 3*sin((1/2)*γ)*sin((1/2)*β)*sin((1/2)*α)))

julia> assign = Dict(α=>0.5, β=>0.7, γ=>0.8);

julia> energy_eval = subs(energy_symbolic, assign...)
1.9542144196548 + -0.0*I

julia> grad_eval = map(x->subs(x, assign...), grad_symbolic)
3-element Array{Basic,1}:
          -1.22808300500511
 -0.311108582564352 + 0.0*I
          -1.56563863069374

julia> circuit_numeric = subs(Float64, circuit, assign...)
nqubits: 3
chain
├─ put on (2)
│  └─ rot(X gate, 0.5)
├─ control(2)
│  └─ (1,) rot(Y gate, 0.7)
└─ put on (1, 2)
   └─ rot(KronBlock{2,XGate}, 0.8)

julia> energy_numeric = expect(h, zero_state(3)=>circuit_numeric)
1.9542144196547988 + 0.0im

julia> grad_numeric = expect'(h, zero_state(3)=>circuit_numeric).second
3-element Array{Float64,1}:
 -1.2280830050051128
 -0.31110858256435187
 -1.5656386306937393
```

H Gate Learning

Given a target unitary matrix and a parameterized circuit, we can learn the gate parameters by minimizing the distance between the two unitaries. Gate learning is useful for quantum compiling, diagonalization [107, 108], and automated quantum algorithm design [109].

```
julia> using YaoExtensions, Yao, Optim

julia> function learn_u4(u::AbstractBlock; niter=100)
           ansatz = general_U4()
           params = parameters(ansatz)
           println("initial fidelity = $(operator_fidelity(u,ansatz))")
           Optim.optimize(x->-operator_fidelity(u, dispatch!(ansatz, x)),
                   (G, x) -> (G .= -operator_fidelity'(u, dispatch!(ansatz, x))[2]),
                   parameters(ansatz), Optim.LBFGS(), Optim.Options(iterations=niter))
           println("final fidelity = $(operator_fidelity(u,ansatz))")
           return ansatz
       end;

julia> u = matblock(rand_unitary(4));

julia> c = learn_u4(u; niter=150)
initial fidelity = 0.1387857997337068
final fidelity = 0.9999999999999997
nqubits: 2
chain
├─ put on (1)
│  └─ chain
│     ├─ rot(Z, 0.19669849872229161)
│     ├─ rot(Y, 1.7777036486174476)
│     └─ rot(Z, 0.5789834641533261)
├─ put on (2)
│  └─ chain
│     ├─ rot(Z, 1.5080795415675794)
│     ├─ rot(Y, 0.75809724318303)
│     └─ rot(Z, -2.131912688745343)
.
.
.
├─ put on (1)
│  └─ chain
│     ├─ rot(Z, 2.179888127260871)
│     ├─ rot(Y, -0.348765381903638)
│     └─ rot(Z, -1.4060784496633136)
└─ put on (2)
   └─ chain
      ├─ rot(Z, 0.40168928969314993)
      ├─ rot(Y, 0.8982267608669345)
      └─ rot(Z, -0.9468001921281308)
```

In this example, the loss is the operator fidelity defined as $F(U_1, U_2) = |\mathrm{Tr}(U_1^\dagger U_2)|/d$ [110], with d the size of Hilbert space. `operator_fidelity'(u1, u2)` returns the gradients of parameters in `u1` and `u2`. We use the LBFGS optimizer [111] provided in `Optim.jl` [112] for optimization. The circuit ansatz (`general_U4`) is the minimal universal two-qubit gate decomposition of Ref. [82] defined in `YaoExtensions`.

I Further Readings

I.1 Quantum Algorithms

A growing list of quantum algorithms that are implemented in `Yao` is shown below.

- Quantum Fourier transformation
- Phase estimation
- Shor's algorithm
- Imaginary time evolution quantum eigensolver
- Variational quantum eigensolver
- Hadamard test
- Quantum singular value decomposition
- HHL algorithm for linear systems of equations
- Quantum approximate optimization algorithm
- Quantum circuit Born machine for generative modeling
- Quantum generative adversarial circuits
- Grover search
- Quantum ordinary differential equation
- Qubit efficient VQE with tensor network inspired circuits

I.2 Developer's guide

`Yao`'s tutorials contain several useful examples for prospective developers.

- Extending the register type: the echo register
- Implementing the SWAP gate in CUDA

I.3 Porting Yao to other parts of Julia ecosystem

- Porting the builtin AD engine to Zygote for gate learning.
- Porting Yao to QuantumInformation [113]