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 built-in 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 complement 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 for ground states [2–4], quantum approximate optimization algorithm for combinatorial problems [5], quantum circuit learning for classification and regression [6, 7], and quantum circuit Born machine for generative modeling [8, 9] have had small scale demonstrations in experiments [10–15]. 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 [16] 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, deep learning and its extension differentiable programming offer great inspiration and techniques for programming quantum computers. Differentiable programming [19] composes 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...
Figure 2: Quantum block intermediate representation plays a central role in Yao. The images of GPU and quantum circuits are taken from JuliaGPU [17] and IBM q-experience [18].

Differentiable programming removes laborious human efforts and sometimes produces even better programs than humans can produce themselves [20].

Differentiable programming is a sensible 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 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, non-trivial 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 [20].

The algorithmic advances in differentiable programming hugely benefit from rapid development in software frameworks [21–26], among which the automatic differentiation (AD) of the computational graph is the key technique behind the scene. A computational graph is a directed acyclic graph that models the computational process from input to output of a program. In order to evaluate gradients via the automatic differentiation, machine learning packages [21–26] construct computational graphs in various 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 [27] 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 quantum 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 using reverse-mode AD on a laptop, see Listing 9.
- Construct sparse matrix representation of 20 site Heisenberg Hamiltonian in approximately 5 seconds, see Listing 17.
- Simulate 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 $6 \times 6$ lattice spin model with a tensor network inspired quantum circuit on GPU [28].

A distinct feature of Yao is its builtin automatic differentiation engine. Instead of building upon existing machine learning frameworks [21–26], 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 with respect to the 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 Sec.5, 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 [29], ProjectQ [30], Q# [31], Cirq [32], qulacs [33], PennyLane [34], qiskit [35], and QuEST [36]. References [37–39] 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 [29] 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 [34] and qulacs [33].

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. 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. In Sec.
we emphasize the flexibility and extensibility of Yao, perhaps its most important features for integrating with existing tools. The applications of Yao and future directions for developing Yao are discussed in Sec. 7 and Sec. 8 respectively. Finally we summarize in Sec. 9. The Appendices (A-I) show various aspects and versatile applications of Yao.

Why Julia?

Julia is fast! The language design avoids the typical compilation and execution uncertainties associated with dynamic languages [40]. Generic programming in Julia [41] 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 [17] powers Yao’s CUDA extension.

Julia integrates 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 [42] 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.

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.
2.1 Representing Quantum Circuits

Figure 4 shows the quantum Fourier transformation circuit [43–45] which contains the \texttt{hcphases} blocks (marked in red) of different sizes. Each block itself is also a composition of Hadamard gates and \texttt{cphase} blocks (marked in blue) on various locations. In \texttt{Yao}, it takes three lines of code to construct the QBIR of the QFT circuit.

The function \texttt{cphase} defines a control phase shift gate with the \texttt{control} and \texttt{shift} functions. The function \texttt{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. The \texttt{chain} block is a composition of blocks with the same number of qubits. It is equivalent to matrix multiplication in reverse order mathematically. Finally, one composes the QFT circuit of a given size by chaining the \texttt{hcphases} blocks. Overall, these codes construct a tree representation of the circuit shown in Fig. 5. The subtrees are composite blocks (\texttt{ChainBlock}, \texttt{ControlBlock}, and \texttt{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 \texttt{Yao}, which are open to extension as shown in Appendix F.

In \texttt{Yao}, to execute a quantum circuit, one can simply feed a quantum state into the QBIR.

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

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

Listing 3: inspecting gates

```
julia> using Yao, SymEngine
julia> @vars θ
(θ,)
julia> shift(θ) |> mat
2×2 LinearAlgebra.Diagonal
{Basic,Array{Basic,1}}:
   1 . .
   . 1 .
   . . exp(im*θ)
julia> control(2,1,2=>shift(θ)) |> mat
4×4 LinearAlgebra.Diagonal{Basic,
  Array{Basic,1}}:
   1 . . .
   . 1 . .
   . . 1 .
   . . . exp(im*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 algebraic data type system, QBIR automatically allows pattern matching with Julia’s multiple dispatch mechanics. Thus, one can manipulate quantum circuits in a straightforward manner using pattern matching on their QBIR.

For example, consider a practical situation where one needs to decompose the Hadamard gate into three rotation gates [46]. 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 re-
turns an empty set. Due to the specialization of \texttt{decompose} method on Hadamard gates, a chain of three rotation gates are returned as a subblock instead.

Besides replacing gates, one can also modify a block by applying tags to it. For example, the \texttt{Daggered} tag takes the hermitian conjugate of the block. We use the ' operator to apply the \texttt{Daggered} tag. Similar to the implementation of \texttt{Transpose} on matrices in Julia, the dagger operator in \texttt{Yao} is "lazy" in the sense that one simply marks the block as \texttt{Daggered} unless there are specific daggered rules defined for the block. For example, the hermitian conjugate of a \texttt{ChainBlock} reverses the order of its child nodes and propagate the \texttt{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 $R_{\sigma}(\theta) \rightarrow R_{\sigma}(-\theta)$
- Time evolution block $e^{-iHt} \rightarrow e^{-iH(-t^*)}$
- Some special constant gates are hermitian conjugate to each other, e.g. \texttt{T} and \texttt{Tdag}.

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

### Listing 5: inverse QFT

```julia
github.com
julia> iqft(n) = qft(n)

julia> iqft(3)
nqubits: 3
chain
<table>
<thead>
<tr>
<th>chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>put on (3)</td>
</tr>
<tr>
<td>H gate</td>
</tr>
<tr>
<td>chain</td>
</tr>
<tr>
<td>control(3)</td>
</tr>
<tr>
<td>(2,) shift(-1.5707963267948966)</td>
</tr>
<tr>
<td>put on (2)</td>
</tr>
<tr>
<td>H gate</td>
</tr>
<tr>
<td>chain</td>
</tr>
<tr>
<td>control(3)</td>
</tr>
<tr>
<td>(1,) shift(-0.7853981633974483)</td>
</tr>
<tr>
<td>control(2)</td>
</tr>
<tr>
<td>(1,) shift(-1.5707963267948966)</td>
</tr>
<tr>
<td>put on (1)</td>
</tr>
<tr>
<td>H gate</td>
</tr>
</tbody>
</table>
```

### 2.3 Matrix Representation

Quantum blocks have a matrix representations of different types for optimized performance. By default, the \texttt{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 \texttt{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 via the \texttt{KrylovKit.jl} in Listing 6. The arithmetic operations \texttt{*} and \texttt{sum} return \texttt{ChainBlock} and \texttt{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. \texttt{YaoExtensions} provides some convenience functions for creating Hamiltonians on various lattices and variational quantum circuits.

### Listing 6: Heisenberg Hamiltonian

```julia
github.com
julia> using KrylovKit: eigsolve

julia> bond(n, i) = sum([put(n, i=>\sigma) * put(n, i+1=>\sigma) for \sigma 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 \texttt{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 \cite{48}, which repeatedly applies the Hamiltonian block to the register. In this case, one can use the \texttt{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
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 and apply them on the fly 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
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 the engine behind the success of deep learning [49]. 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 gradients in a single pass together with the objective function, 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) provides more efficient circuit differentiation for variational quantum algorithms compared to conventional reverse mode differentiation and forward mode differentiation (Sec. 3.2). By taking advantage of the reversible nature of quantum circuits, the memory complexity is reduced to constant compared to typical reverse mode AD [49]. 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. In the classical simulation, 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 backpropagation. 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 [21–26] is not a satisfiable solution. Yao’s customized AD engine exploits the inherent reversibility of quantum circuits [50, 51]. 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 [51, 52].

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

$$|\psi_{k+1}\rangle = U_k |\psi_k\rangle,$$

where $U_k$ is a unitary gate parametrized by $\theta_k$. We define the adjoint of a variable as $\tau = \frac{\partial C}{\partial x}$.
according to Wirtinger’s derivative [53] for complex numbers, where $L$ is a real-valued objective function that depends on the final state. Starting from $Z = 1$ we can obtain the adjoint of the output state.

To pull back the adjoints through the computational graph, we perform the backward calculation [54]

\[
\begin{align*}
|\psi_k\rangle &= U_k^|\psi_{k+1}\rangle \\
|\psi_k\rangle &= U_k^T |\psi_{k+1}\rangle
\end{align*}
\] (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 [54]

\[
U_k = |\psi_{k+1}\rangle \langle \psi_k|.
\] (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 $\theta_k$ from the adjoint of the unitary matrix $U_k$.

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 $L$. 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 $\text{put}(\text{nbit}, i=>\text{chain}(\text{Rz}(\alpha), \text{Rx}(\beta), \text{Rx}(\gamma)))$, we obtain the desired $\pi$, $\beta$, and $\gamma$ 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 [55].

To demonstrate the efficiency of Yao’s AD engine, we use the codes in Listing 9 to simulate the variational quantum eigensolver (VQE) [56] 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.

Here, variational_circuit is predefined in YaoExtensions to have a hardware efficient architecture [57] shown in Fig. 9. 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.

Listing 9: 10000-layer VQE

```julia
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\(\text{'ts1}(h, zero_state(10)=>circuit)
    dispatch!(-, circuit, 1e-3 * grad)
    println("Step $i, energy = $(expect(h, zero_state(10)=>circuit))")
end
```
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 $\Sigma$ being both hermitian and reflexive ($\Sigma^2 = 1$). For example, $\Sigma$ can be the Pauli gates X, Y and Z, or multi-qubit gates such as CNOT, CZ, and SWAP. Every two-qubit gate can be decomposed into Pauli rotations and CNOTs (or CZs) via gate transformation [58]. Under these conditions, the gradient to a circuit parameter is [7, 59–61]

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

where $\langle O \rangle_\theta$ denotes the expectation of the observable $O$ with the given parameter $\theta$. 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 [34] and qulacs [33]. 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)

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

Yao implements an efficient way to break the measurement into the expectation of local terms by diagonalizing the observed operator symbolically as bellow.
The return value of \texttt{eigenbasis} contains two QBIRs \( E \) and \( U \) such that \( O = U*E*U \). \( E \) is a diagonal operator that represents the observable in the measurement basis. \( U \) is a circuit that rotates computational basis to the measurement basis.

The above gradient estimator 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

\[
F_{\theta} = \mathbb{E}_{x \sim p_{\theta}, y \sim p_{\theta}}[K(x, y)],
\]

where \( K \) is a symmetric function, \( p_{\theta} \) 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

\[
\frac{\partial F_{\theta}}{\partial \theta} = \mathbb{E}_{x \sim p_{\theta}, y \sim p_{\theta}}[K(x, y)]
- \mathbb{E}_{x \sim p_{\theta}, y \sim p_{\theta}}[K(x, y)],
\]

which is also related to the measure valued gradient estimator for stochastic optimization [62]. Within this formalism, \texttt{Yao} provides the following interfaces to evaluate gradients with respect to the maximum mean discrepancy loss [63, 64], which measures the probabilistic distance between two sets of samples.

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. \texttt{Yao} handles all of these cases with a unified \texttt{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.

\[
\text{Listing 12: CUDA register}
\]

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

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

Each register type has its own device-specific instruction set. They are declared in \texttt{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 support to various backends without the user having to worry about changes to frontend interfaces.

**Listing 13: instruct! and measure**

```julia
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
0010
0010
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), (2,), (2,), (1,))`, where the last three tuples are gate locations, control qubits, and configuration of the control qubits (0 for inverse control, 1 for control). Respectively. The `measure` function simulates measurement from the quantum register and provides bit strings, while `measure!` returns the bit string and also collapses 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 read-out 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 [65] shown in Fig. 7.

![Figure 7: 5-qubit quantum Phase estimation circuit.](image)

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 cannot 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 14: focus! and relax!**

```julia
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 15: quantum phase estimation**

```julia
PE(n, m, U) = chain(
    n+m, # total number of qubits
    repeat(H, 1:n), # apply H from 1:n
    chain(control(
        k, # apply inverse QFT on a local scope
        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 [66] or an ensemble of pure quantum states for thermal state simulation [55]. 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) [67] 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 majorized. 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 16: a batch of quantum registers**

```julia
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 [68] for better performance. The sparse matrices `IMatrix`, `Diagonal`, `PermMatrix` and `SparseMatrixCSC` introduced in Appendix E also have their static version defined in `LuxurySparse.jl` [69]. 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 [70–73]. 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

<table>
<thead>
<tr>
<th>Package</th>
<th>Language</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cirq [32]</td>
<td>Python</td>
<td>0.8.0</td>
</tr>
<tr>
<td>qiskit [35]</td>
<td>C++/Python</td>
<td>0.19.2</td>
</tr>
<tr>
<td>qulacs [33]</td>
<td>C++/Python</td>
<td>0.1.9</td>
</tr>
<tr>
<td>PennyLane [34]</td>
<td>Python</td>
<td>0.7.0</td>
</tr>
<tr>
<td>QuEST [36]</td>
<td>C/Python</td>
<td>3.0.0</td>
</tr>
<tr>
<td>ProjectQ [30]</td>
<td>C++/Python</td>
<td>0.4.2</td>
</tr>
<tr>
<td>Yao</td>
<td>Julia</td>
<td>0.6.2</td>
</tr>
<tr>
<td>CuYao</td>
<td>Julia</td>
<td>0.2.2</td>
</tr>
</tbody>
</table>

Table 1: Packages in the benchmark.

Although QuEST is a package originally written in C, we benchmark it in Python via pyquest-cffi [74] for convenience. Pennylane is benchmarked with its default backend [75]. 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 [76]. qiskit is benchmarked with qiskit-aer 0.5.1 [77] and qiskit-terra 0.14.1 [78] using the statevector method of the qasm simulator.

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 [79] and BenchmarkTools [80] 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.
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 measure the performance of executing one single gate instruction.

Figure 8 shows the running times of various gates applied on the second qubit of the register from size 4 to 25 qubits 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.

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 VQE experiment [57]. 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. qiskit cuda backend shows better performance for more than 20 qubits. 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 \texttt{yao} \times 1000). One can see that batching offers substantial speedup for small circuits.

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. \texttt{Yao} shows the least overhead in these benchmarks. \texttt{qulacs} also did an excellent job of suppressing these overheads.

### 5.4 Matrix Representation and Automatic Differentiation Performance

As discussed in Sec. 2.3 and Sec. 3.1, \texttt{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 laptop. The forward mode AD discussed in Sec. 3.2 is slower by order of a hundred in such simulations.

#### Listing 17: 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 \texttt{Yao} in comparison to other frameworks. We nevertheless emphasize that the most important feature of \texttt{Yao} is its flexibility and extensibility.

#### 6.1 Extending QBIR nodes

It is easy to extend \texttt{Yao} with new gates and quantum block nodes. One can define constant gates by giving its matrix representation. For example, the \text{FSim} gate that appears in Google supremacy experiments [82] is composed of an \text{ISWAP} gate and a \text{cphase} gate with a fixed angle.

#### Listing 18: \text{FSim} gate

```
 julia> using Yao, LuxurySparse
 julia> @const_gate ISWAP = PermMatrix((\texttt{[1,3,2,4]}, [1,1.0im,1.0im,1]))
 # \text{FSim} is already defined in YaoExtensions
 julia> @const_gate MyFSim = mat((ISWAP\* 
                                      control(2, 2, i=>shift(-\pi/6)))')
 julia> put(10, (4,2)=>MyFSim)
 nqubits: 10
 put on (4, 2)
 L. MyFSim gate
```

The macro \texttt{@const_gate} defines a primitive gate that subtypes \texttt{ConstantGate} abstract type. It generates global gate instances \texttt{ISWAP} and \texttt{MyFSim} as well as new gate types \texttt{MyFSimGate} and \texttt{ISWAPGate} for dispatch. This macro also binds operators properties such as \texttt{ishermitian}, \texttt{isreflexive} and \texttt{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 \texttt{CuYao} [83] we dispatch \texttt{instruct!} and \texttt{measure!} to the \texttt{ArrayReg\{B,T,<:CuArray\}} type. Here besides the batch number \texttt{B} and data type \texttt{T}, the third type parameter \texttt{<:CuArray} specifies the storage type. The dispatch directs the instructions to the CUDA kernels written in \texttt{CUDAnative} [17], 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 1.
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 [30] 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 [84] with the much simpler codebase.

The tensor network inspired quantum circuits described in [28, 66] allow the study of large systems with a reduced number of qubits. For example, one can solve the ground state of a $6 \times 6$ frustrated Heisenberg lattice model with only 12 qubits. These circuits can also compress a quantum state to the hardware with fewer qubits [85]. These applications need to measure and reuse qubits in the circuits. Thus, one can not take $n_{shots}$ measurements to the wavefunction. Instead, one constructs a batched quantum register with $n_{batch}$ 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 [86]. 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.

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 [88], eQASM [89] and Quil [90]. As an ongoing project towards this end, we and the author of the general-purpose parser generator RBNF [91] developed a QASM parser and codegen in YaoQASM [92]. It allows one to parse the OpenQASM [88] to QBIR and dump QBIR to OpenQASM, which can be used for controlling devices in the IBM Q Experience [18, 30].

8.2 Circuit Compilation and Optimization

Circuit compilation and optimization is a key topic for quantum computing towards practical experiments. A new language interface along with a compiler for Yao [93] is under development. It should be more compilation friendly with a design based on Julia’s native abstract syntax tree. By integrating with Julia compiler, it will allow Yao to model quantum channels in a seamlessly way.

On the other hand, quantum circuit simplification and optimization are crucial for reducing the cost of both simulations and experiments. The ongoing circuit simplification project [94] will also support better pattern matching and term rewriting system with support of ZX calculus [95]. This will allows smarter and more systematic circuit simplifications such as the ones in Refs. [96–98].

8.3 Noisy Simulation

Noise important for simulation of near-term quantum devices. Currently, Yao does not support noisy simulations directly. However, a batched register in Yao be conveniently converted to a reduced density matrix. By porting Yao with QuantumInformation.jl [99], one can carry out noisy simulation with density matrices. One can find a blog in Appendix I.

8.4 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 [100–102]. 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 [103].

Besides serving as an efficient simulation backend, the tensor networks are also helpful for quantum machine learning, given the fact that they have already found many applications in various machine learning tasks [104–108]. 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 [109] 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 [82, 110]. The dumped tensor networks can be further used for quantum circuit simplification [111] 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, Christopher J. Wood, Damian Steiger, Damion Steiger, Craig Gidney, corryvreckan, Johannes Jakob Meyer and Nathan Killoran for reviewing the performance benchmarks [81]. We thank Divyanshu Gupta for integrating Yao with DifferentialEquations.jl [112], 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

[2] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou,


[73] Fang Zhang et al. Alibaba cloud quantum


[78] 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.


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 [42] and construct a molecule Hamiltonian in Julia.

```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
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
[jscale: -0.04919764587136755 + 0.0im] kron
  1=>X
  2=>X
  3=>Y
  4=>Y
+ [scale: 0.13716572937099503 + 0.0im] kron
  2=>Z
```

Accepted in Quantum 2020-09-29, click title to verify
B  Builtin Block Types

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

```julia
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
      POGate
      P1Gate
      PdGate
      PuGate
      SGate
      SdagGate
      TdagGate
      ToffoliGate
      ZGate
    GeneralMatrixBlock
    Measure
    PhaseGate
    RotationGate
    ShiftGate
    TimeEvolution
    TrivialGate
    IdentityGate
```

More blocks, such as SqrtX and FSimGate, are available in YaoExtensions.
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> using Yao, SymEngine

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

```julia
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

<table>
<thead>
<tr>
<th>Script</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=&gt;C, 2=&gt;X</td>
<td>control(9, 1, 2=&gt;X)</td>
</tr>
<tr>
<td>1=&gt;H, 4=&gt;H, 7=&gt;H</td>
<td>kron(9, 1=&gt;H, 4=&gt;H, 7=&gt;H)</td>
</tr>
<tr>
<td>1=&gt;H</td>
<td>put(9, 1=&gt;H)</td>
</tr>
<tr>
<td>begin ... end</td>
<td>chain(9, [...])</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Matrix Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2</td>
<td>IMatrix</td>
</tr>
<tr>
<td>Z, T, S, Rz</td>
<td>Diagonal</td>
</tr>
<tr>
<td>X, Y, CNOT, CZ, SWAP</td>
<td>PermMatrix</td>
</tr>
<tr>
<td>P0, P1, Pu, Pd, PSwap</td>
<td>SparseMatrixCSC</td>
</tr>
<tr>
<td>H, Rx, Ry</td>
<td>Matrix</td>
</tr>
</tbody>
</table>

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 [113] are defined in LuxurySparse.jl [69]. 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
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.

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>D</th>
<th>P</th>
<th>S</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P/P/S/D</td>
<td>P/P/S/D</td>
<td>P/P/S/P</td>
<td>S/S/S/P</td>
<td>M/S/M/P</td>
</tr>
<tr>
<td>M</td>
<td>M/S/M/D</td>
<td>M/S/M/D</td>
<td>M/S/M/P</td>
<td>M/S/M/S</td>
<td>M/S/M/M</td>
</tr>
</tbody>
</table>

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, SparseMatrixCSC 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.
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 [114]. 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.

```julia
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.

```julia
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 built-in 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.

```
using Yao, YaoExtensions, SymEngine

@vars α β γ

chain = chain(put(3, 2=>Rx(α)), control(3, 2, 1=>Ry(β)), put(3, (1,2)=>rot(kron(X, X), γ)))

julia> h = heisenberg(3);

julia> energy_symbolic = expect(h, zero_state(Basic, 3)=>circuit)

julia> grad_symbolic = expect(h, zero_state(Basic, 3)=>circuit).second

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
-0.31110858256435187
-1.56563863069374

julia> circuit_numeric = subs(Float64, circuit, assign...)

chain

- put on (2)
  - rot(XGate, 0.5)
  - control(2)
    - (1,) rot(YGate, 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
```

Accepted in Quantum 2020-09-29, click title to verify
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 [115, 116], and automated quantum algorithm design [117].

```
using YaoExtensions, Yao, Optim

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;

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, 0.1966984987229161)
      rot(Y, 1.7777036486174476)
      rot(Z, 0.57983464153261)
    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, -1.406078449663136)
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 $F(U_1, U_2) = |\text{Tr}(U_1^\dagger U_2)|/d$ [118], 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 [119] provided in Optim.jl [120] for optimization. The circuit ansatz (general_U4) is the minimal universal two-qubit gate decomposition of Ref. [86] 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 [99]