Yao.jl

An extensible efficient framework for quantum algorithm design

Today I will introduce our software framework for quantum algorithm design Yao. It is an extensible efficient framework for quantum algorithms design.

The name Yao means normalized but not orthogonal in Chinese. And you might be able to find an easter egg in our framework later when you try it.

About me

Master Student in UWaterloo

Xiuzhe (Roger) Luo
罗秀哲

Github: Roger-luo

Former RA in Lei Wang, Institute of Physics, Chinese Academy of Science

I'm a new master student in Roger Melko's group. This work was initiated when I was an Research Assistant in Lei Wang's group in IOP, because of my PhD visa issue in Australia, so I didn’t actually have any pressure on papers. And the idea was begin with an introduction of Julia I did in IOP.

QC 101

So maybe I should ask who is currently doing quantum information or has an quantum information background here?

OK, so I'll introduce a some basics about quantum computing, so you could understand what I'm talking about next.

1. Logic bit to qubits.

Classical Logic Bits

So first, we can represent classical bits as one hot vectors, like we can mark the first one is zero and second one is one, so bit zero becomes this, and in the same way we have bit one.

Classical Logic Bits

$$0 \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{matrix} 0 \\ 1 \end{matrix} \qquad 1 \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{matrix} 0 \\ 1 \end{matrix}$$

Classical Logic Bits

$$0 \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{matrix} 0 \\ 1 \end{matrix} \qquad 1 \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{matrix} 0 \\ 1 \end{matrix}$$

$$00 \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix} \qquad 01 \rightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix}$$

Classical Operations (Gates)

$$\mathbf{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad X \cdot 0 \rightarrow \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

For classical gates, we can also have their matrix form, Like for the X gate, or NOT gate, it simply flips the bit. Therefore its matrix is (click)

Quantum

In quantum case, we could generalize this notation to Hilbert space, which is a normalized complex value vector and unitary matrices, and we need the Dirac notation to annotate the current space and its adjoint space.

We all the element of these vector as amplitudes, since in quantum physics, this is actually a wave function’s expansion on computational basis.

And quantum physics tell us this these operations are possible to be implemented physically not just in your imagination.

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix}0 & -im\\ im & 0\end{pmatrix}$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix}0 & -im\\ im & 0\end{pmatrix} \qquad \mathbf{Z} = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

Quantum

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}|0\rangle\\ |1\rangle\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$\mathbf{X} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \qquad \mathbf{Y} = \begin{pmatrix}0 & -im\\ im & 0\end{pmatrix} \qquad \mathbf{Z} = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\ im\end{pmatrix}\begin{matrix}\langle 0|\\ \langle 1|\end{matrix} \rightarrow \frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)$$

Quantum Circuit

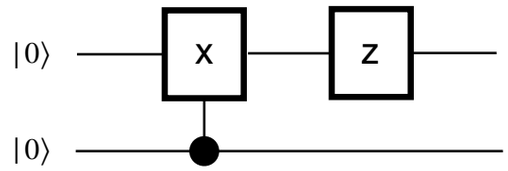
- **Probability based**
- **Unitary**
- **Normalized**

Like classical circuit, we can still use a similar diagram to represent how those operations are applied to the current (register) state.

We can use a rectangle to annotate the gate and a dot annotate the control qubits, e.g (click)

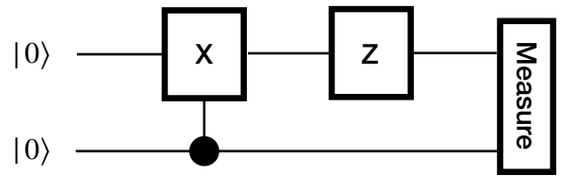
But however, quantum mechanics says you can’t directly read the amplitudes, you have to measure them with a physical operator to get the classical readable results. So at last we need to measure the state to get the results.

Quantum Circuit



- **Probability based**
- **Unitary**
- **Normalized**

Quantum Circuit



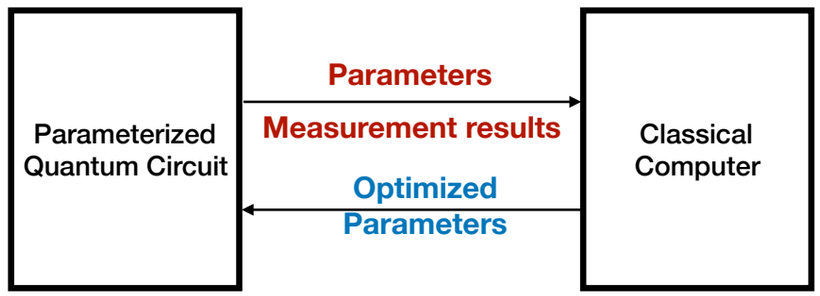
- **Probability based**
- **Unitary**
- **Normalized**

Quantum Algorithms

- Shor algorithm
- HHL
- etc.

So now we have some kind of quantum computation model, and we will have some algorithms on it, there are some famous algorithms you must heard of like these two whose authors are here…

Variational Quantum Algorithms/Circuits

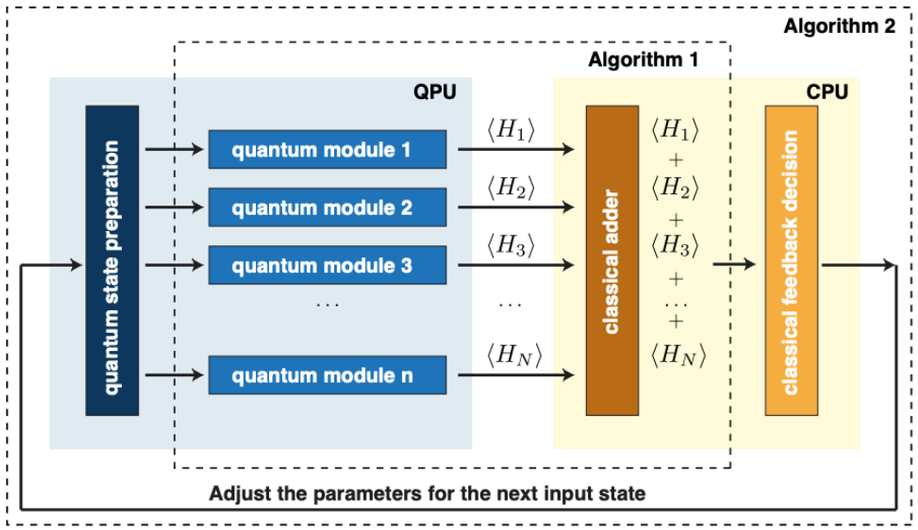


So instead of finding out a certain structure of circuit that can accelerate a problem exponentially, which is really really hard, we instead want to make use the power of classical computers and combine them with a quantum processor.

These are part of the motivation of variational quantum circuits. It is kinda like some machine learning algorithms today, so some people also call this quantum machine learning.

There can be some advantage for this framework, e.g robustness against noise, since the variational algorithms are usually work with problem with loss and it is fine to have a little bit error in the loss usually, which is important since near term quantum device may not be large enough to have error correction and logical qubits.

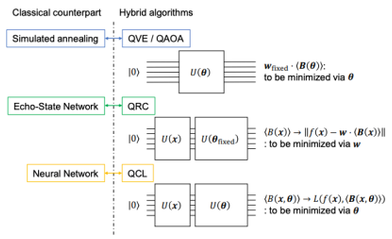
Variational Quantum Eigensolver (arxiv: 1304.3061)

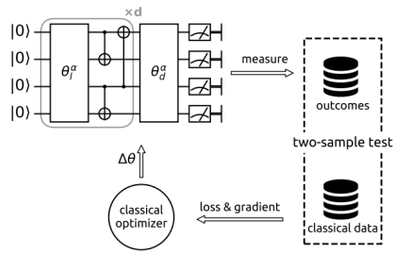


Although we haven't find a very practical algorithm that can show us the quantum advantage, a lot people believe the quantum computer or a quantum device itself, is a nice tool for solving and simulating quantum physics related problems. Intuitively, the structure of quantum circuit itself may embed a lot prior knowledge on quantum physics, e.g the unitary operation.

Thus, an intuitive application would be solving a Hamiltonian with quantum device, which is the goal of this variational quantum eigensolver. And the very first demonstration of this circuit only has one parameter.

Quantum Circuit Learning (arxiv: 1803.00745)



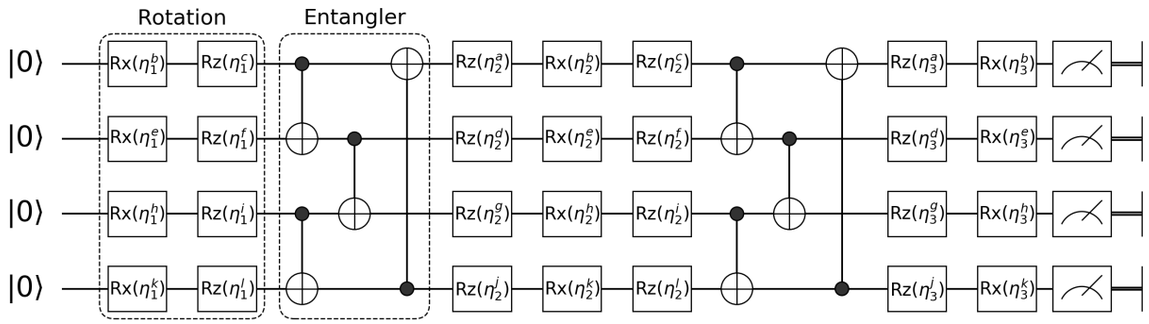


Quantum Circuit Born Machine (arxiv: 1804.04168)

So is there any variational quantum circuit that has better capacity? later there are several paper revealed that one can actually construct a variational quantum circuit with rotation gates and CNOT entangler, which can provide hierarchical structure to provide more capacity for the model.

In QCL, they find the phase kicking trick to calculate the gradient of quantum circuits, and in quantum circuit born machine paper, they find the MMD loss to connect quantum models to classical distribution or in another word classical data.

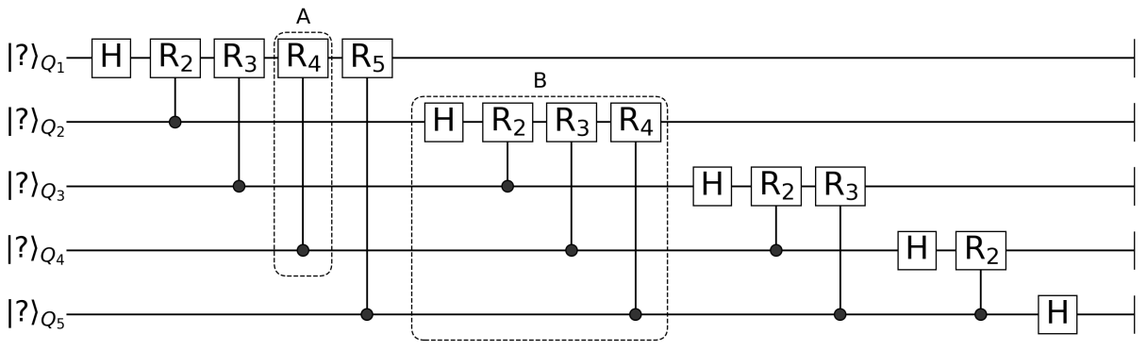
And our story begin with quantum circuit born machine, but I will show you later, it is actually also connected to quantum programming language community's work.



Differentiable Circuit of Size 4, depth 2

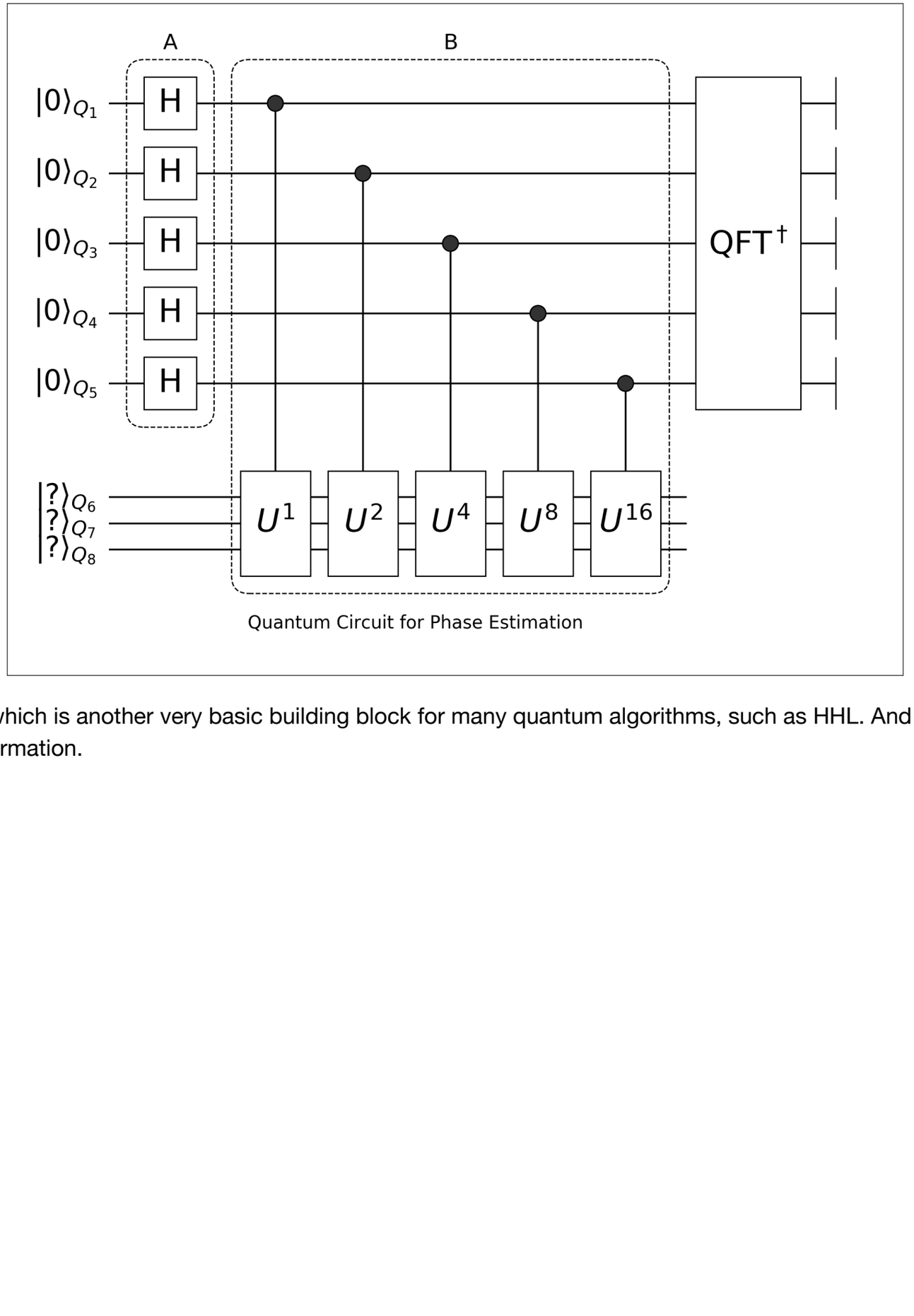
This is a general structure for quantum circuit born machine. It has two patterns, one is the rotation hierarchy the other is the entangler hierarchy. And a measurement in the end.

So this quantum circuit actually has a repeat pattern in it. This is actually something very common among many quantum algorithms.



Quantum Fourier Transformation circuit of size 5

This is a circuit for quantum Fourier transformation, which is very basic building block for many quantum algorithms, such as the Shor algorithm. It has two patterns, but a little bit different.



Quantum Circuit for Phase Estimation

This is the quantum phase estimation, which is another very basic building block for many quantum algorithms, such as HHL. And besides two different pattern, it also use the inverse quantum Fourier transformation.

Quipper: A Scalable Quantum Programming Language
(arxiv: 1304.3390)

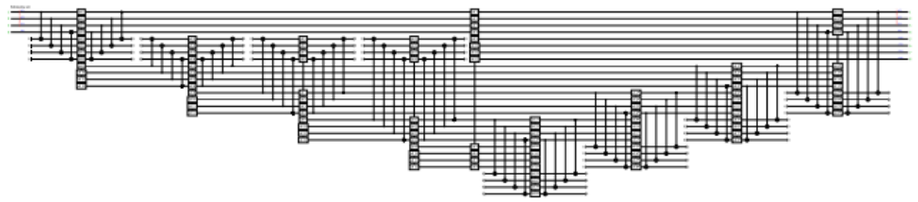
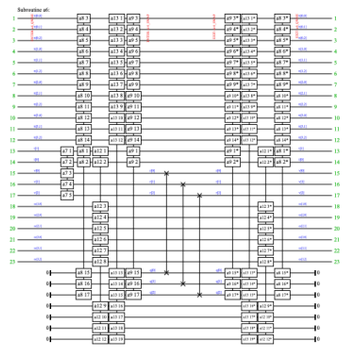


Figure 2. The circuit for `o4_POW17`

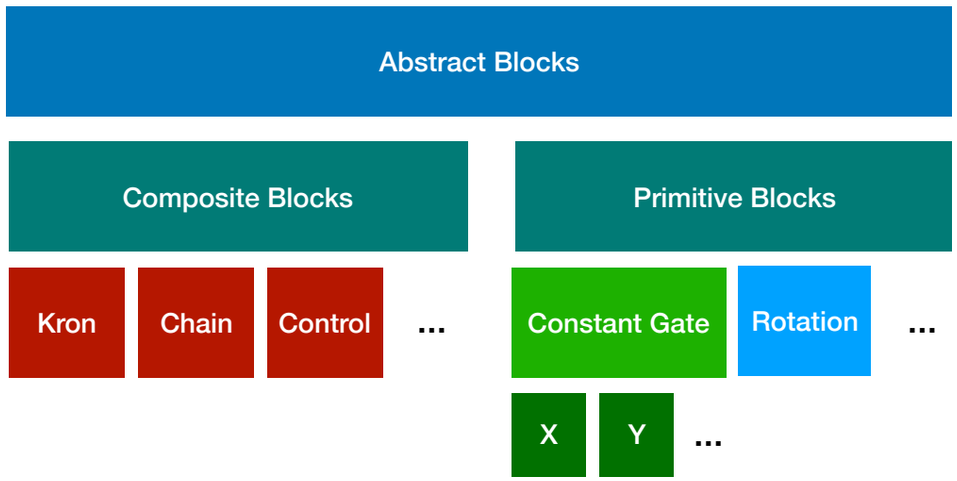


So the idea is simple why don’t we just represent the quantum circuit as compassable blocks in different scale, it is just like what we have in programming languages, like subroutines, or functions.

In fact, programming language people have been working on this, in 2013, the quantum programming language Quipper defined this representation in their paper. With this design, they were able to represent very complicated large quantum circuits with a very sparse representation as an abstract syntax tree (aka AST), so it is like a DSL for quantum circuits.

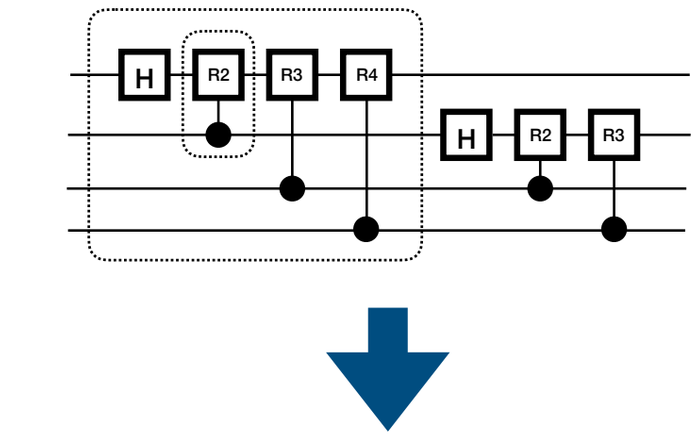
They can box the quantum circuit and re-use it in others, which very convenient and saves time, moreover, it is completely hardware free, we don’t need to care about the registers!

The Quantum Block IR



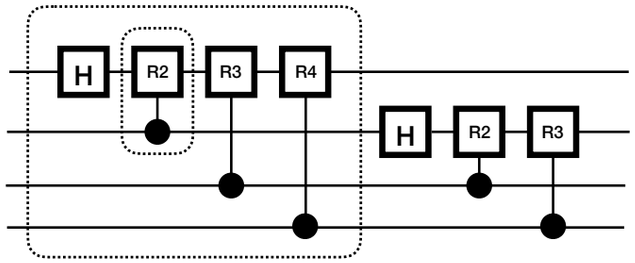
We basically use the same intuition but to make it more Julian, we embedded this into Julia's type system instead of just functions/subroutines, or a complete frontend. Our representation defined a symbolic system inside Julia's type system.

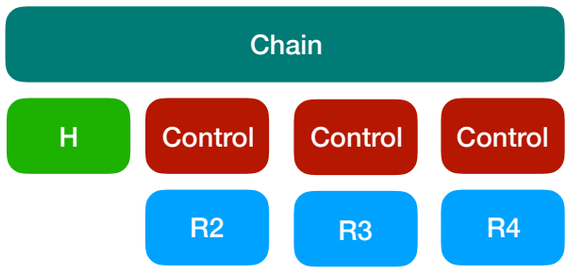
Block Tree Intermediate Representation (IR)

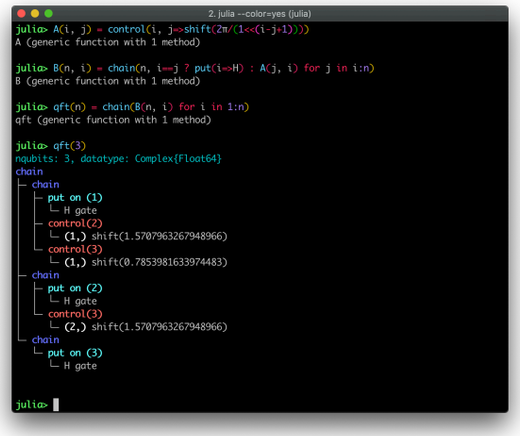


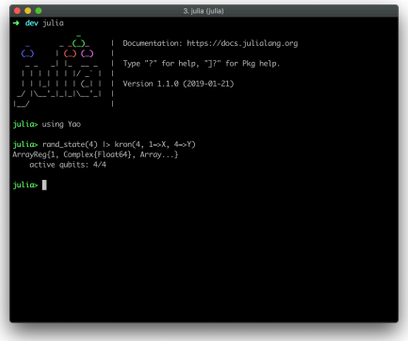
In this way, you could define the quantum circuit in a tree, or more precisely a Direct Acyclic Graph, aka DAG, since we allow you put the same children to different parents. And then we define some building keywords which are the very basic building blocks in our component package YaoBlocks, they are composite blocks like chain and kron, their functionality are similar to vertical and horizontal for loops.

Block Tree Intermediate Representation (IR)



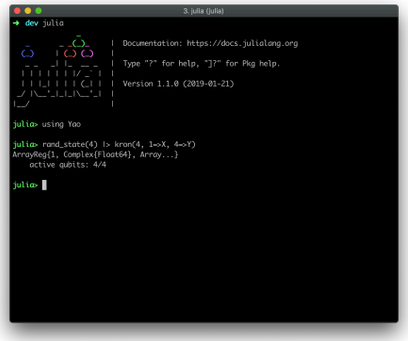




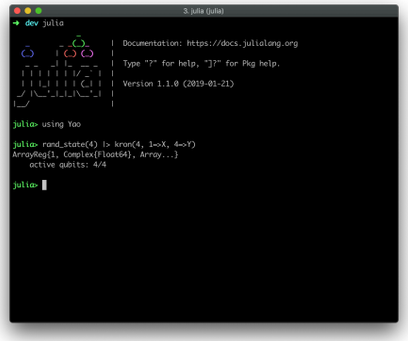


So you may wonder why can’t we just use functions and closures? Because Julia can do multiple dispatch, we could combine the type with many different methods to provide:

1. Aggressively specialization for simulation



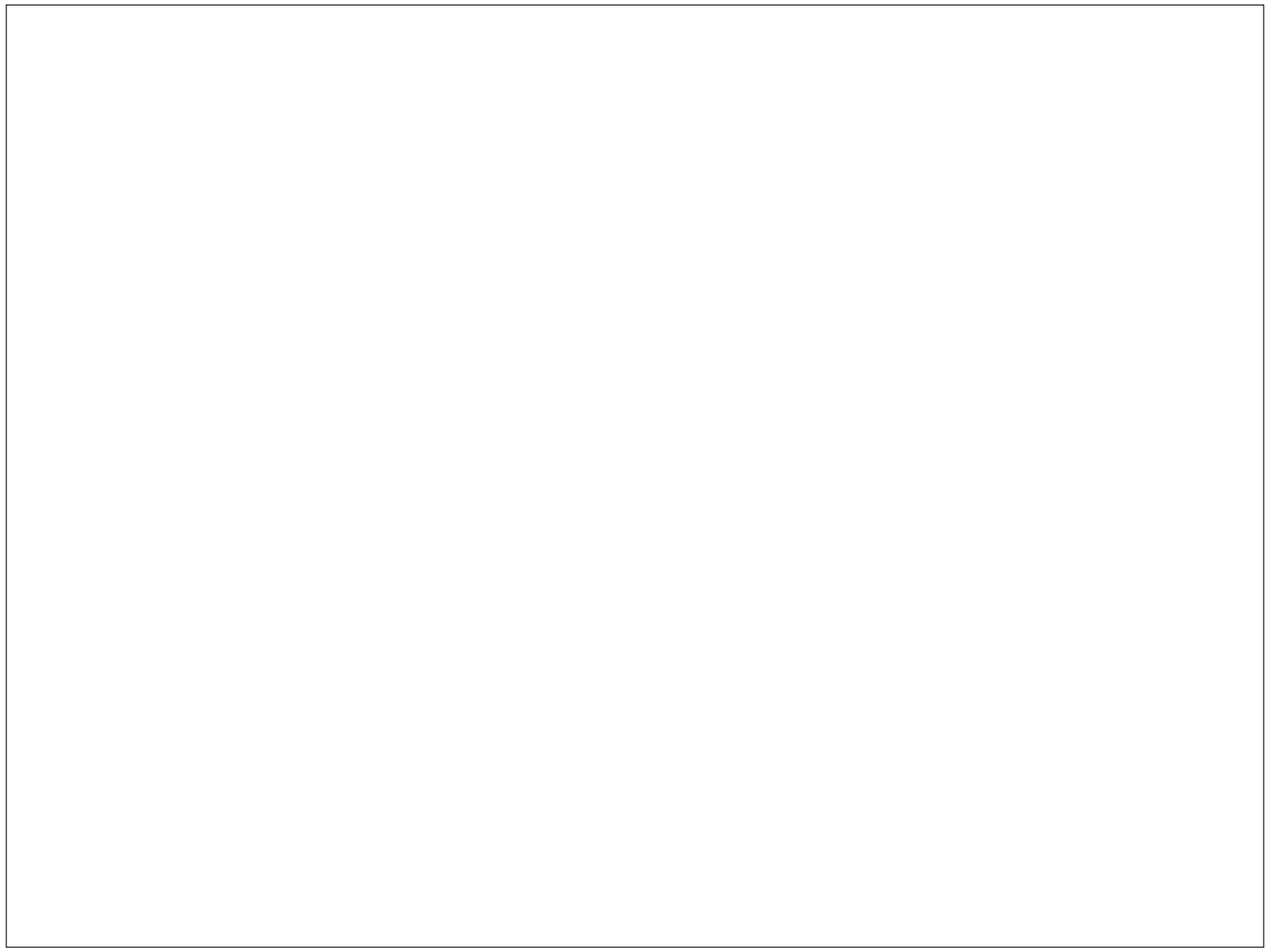

```
function apply!(r::ArrayReg, k::KronBlock)
    _check_size(r, k)
    for (locs, block) in zip(k.locs, k.blocks)
        _instruct!(state(r), block, Tuple(locs:locs+nqubits(block)-1))
    end
    return r
end
```



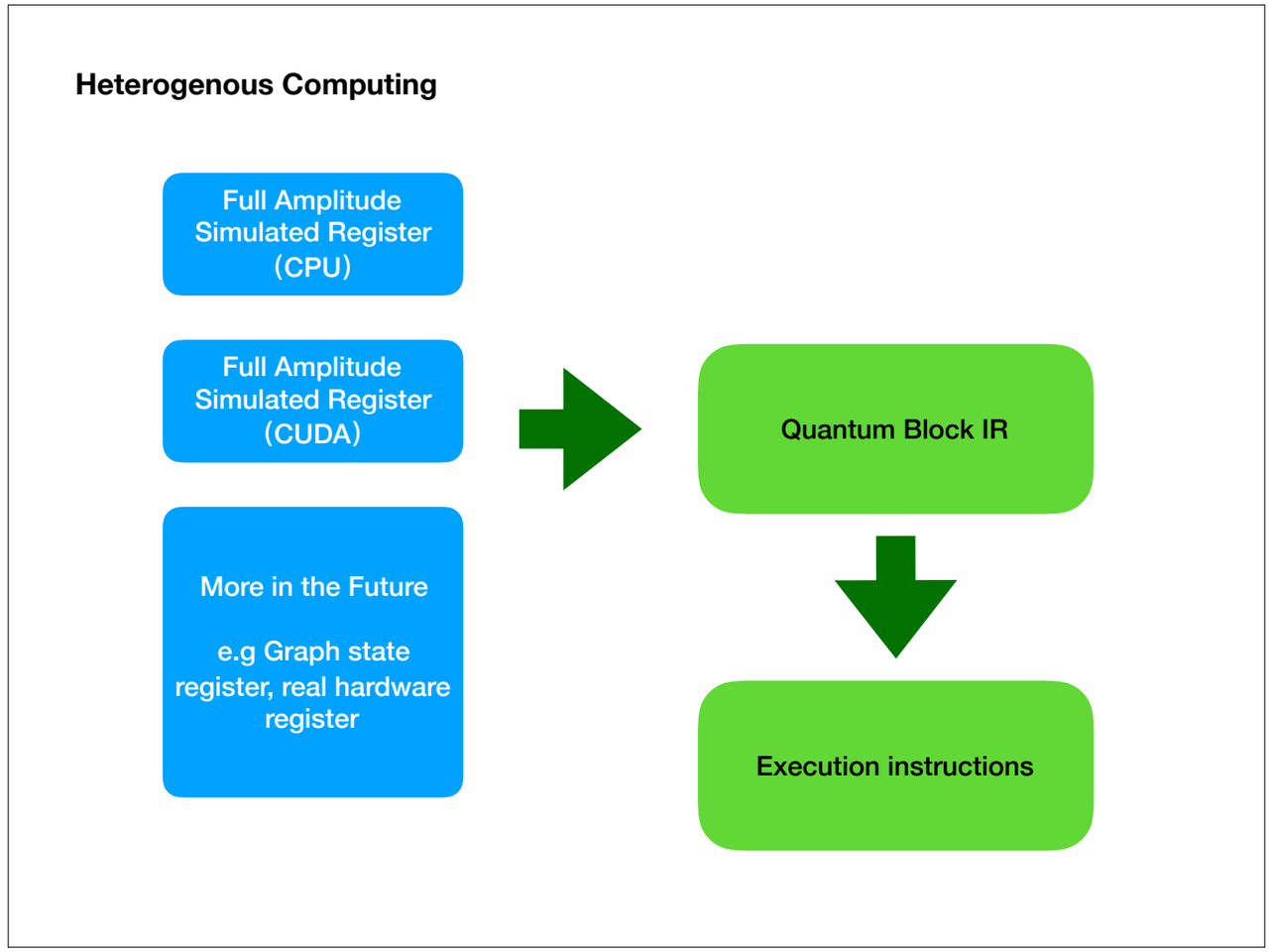

```julia
function apply!(r::ArrayReg, k::KronBlock)
    _check_size(r, k)
    for (locs, block) in zip(k.locs, k.blocks)
        _instruct!(state(r), block, Tuple(locs:locs+nqubits(block)-1))
    end
    return r
end
```

```julia
# specialization
# paulis
function YaoBase.instruct!(state::AbstractVecOrMat, ::Val{:X}, locs::NTuple{N, Int}) where N
    mask = bmask(locs)
    do_mask = bmask(first(locs))
    for b in basis(state)
        @inbounds if anyone(b, do_mask)
            i = b+1
            i_ = flip(b, mask) + 1
            swaprows!(state, i, i_)
        end
    end
    return state
end
```

1. Analysis the quantum circuit and transform it like a symbolic system



```
# Inspired by MasonPotter/Symbolics.jl
"""
    simplify(block[; rules=__default_simplification_rules__])

Simplify a block tree accroding to given rules, default to use
[`__default_simplification_rules__`](@ref).
"""
function simplify(ex::AbstractBlock; rules=__default_simplification_rules__)
    out1 = simplify_pass(rules, ex)
    out2 = simplify_pass(rules, out1)
    counter = 1
    while (out1 isa AbstractBlock) && (out2 isa AbstractBlock) && (out2 != out1)
        out1 = simplify_pass(rules, out2)
        out2 = simplify_pass(rules, out1)
        counter += 1
        if counter > 1000
            @warn "possible infinite loop in simplification rules. Breaking"
            return out2
        end
    end
    return out2
end
```



Moreover, because it is device free, we could do heterogenous computing with a simple switch on the register type just like many other Julia programs.

we currently have experimental CUDA support in CuYao.jl based on Julia's native CUDA programming stack, CUDAnative, etc. By experimental, means this feature is not stable as other parts, it may have some glitches.

Heterogenous Computing

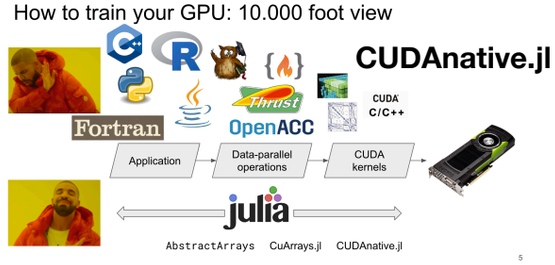
Full Amplitude Simulated Register (CPU)

Full Amplitude Simulated Register (CUDA)

More in the Future

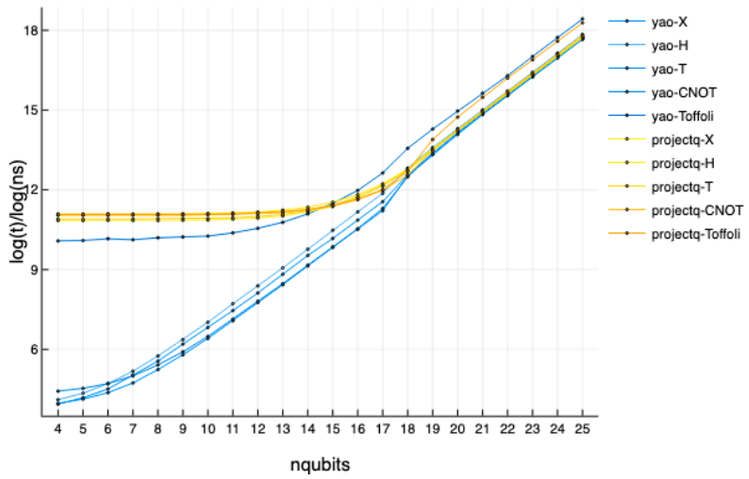
e.g Graph state register, real hardware register

How to train your GPU: 10.000 foot view



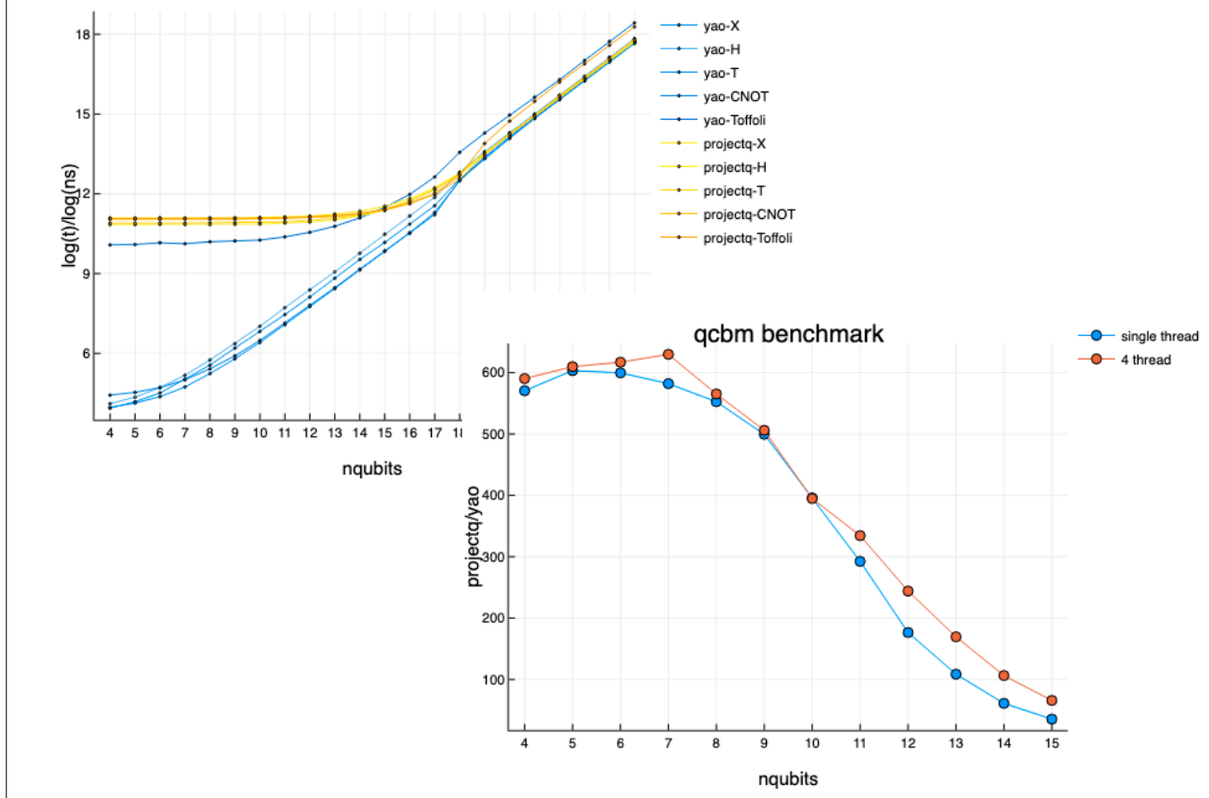
Quantum Block IR

Execution instructions



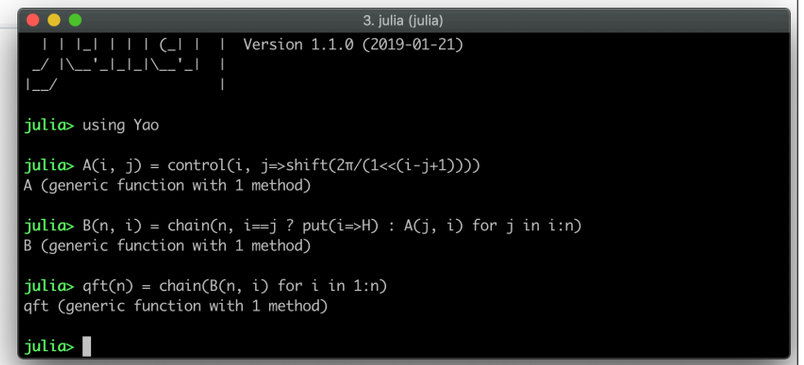
With these design, we are able to dispatch special implementation for each different gate, which results in a pretty nice benchmark, even with abstractions, we have way better performance than others.

Which is like over 600x times faster at some cases.


log(t)/log(ns)
nqubits
yao-X
yao-H
yao-T
yao-CNOT
yao-Toffoli
projectq-X
projectq-H
projectq-T
projectq-CNOT
projectq-Toffoli
qcbm benchmark
projectq/yao
nqubits
single thread
4 thread

```qasm
// quantum Fourier transform
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
creg c[4];
x q[0];
x q[2];
barrier q;
h q[0];
cu1(pi/2) q[1],q[0];
h q[1];
cu1(pi/4) q[2],q[0];
cu1(pi/2) q[2],q[1];
h q[2];
cu1(pi/8) q[3],q[0];
cu1(pi/4) q[3],q[1];
cu1(pi/2) q[3],q[2];
h q[3];
measure q -> c;
```

QASM

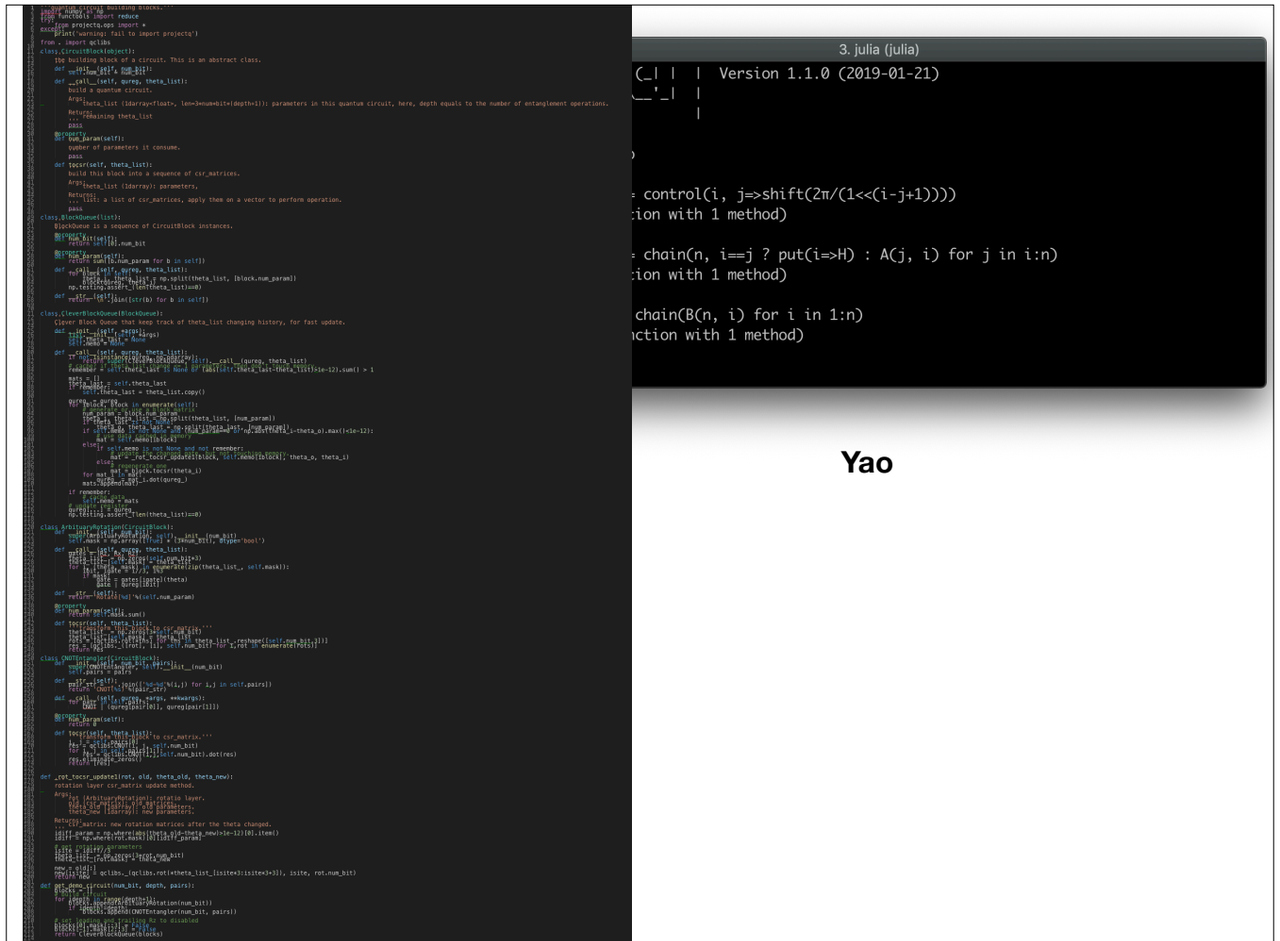


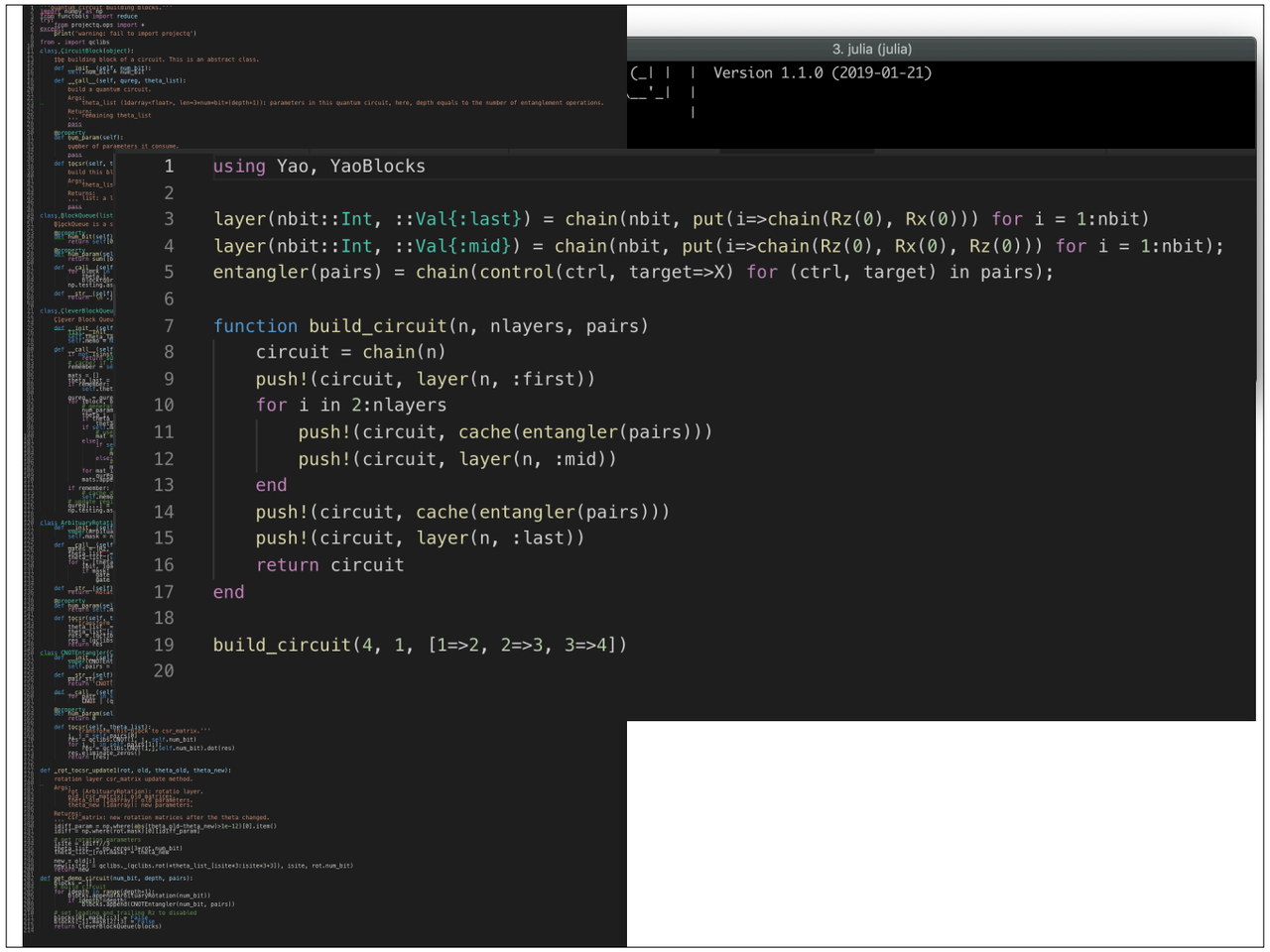
Yao

While the noise of defining a quantum circuit is reduced significantly, this is a QFT in QASM, the right hand is Yao.

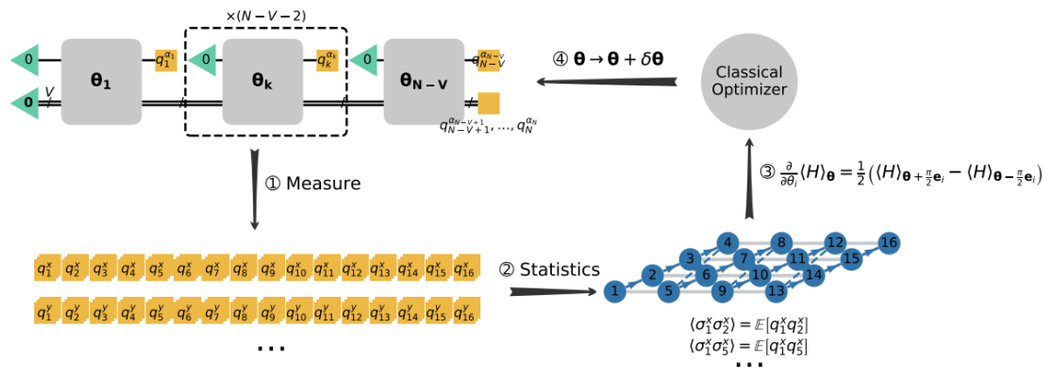
This is part of the QCBM circuit definition with ProjectQ, this is Yao.

Although, I should disclaim here that the ProjectQ's implementation is very long because this tree structure is necessary for dispatching parameters, thus we need to define anyway in ProjectQ, but Yao provides this structure already, but I would suggest you try them out both, and I'm sure you will find Yao is still simpler in many cases.


3. julia (julia)
(_| | | Version 1.1.0 (2019-01-21)
control(i, j=>shift(2π/(1<<(i-j+1))))
chain(n, i==j ? put(i=>H) : A(j, i) for j in i:n)
chain(B(n, i) for i in 1:n)
Yao


3. julia (julia)
(_| | | Version 1.1.0 (2019-01-21)
__'_| |
       |
using Yao, YaoBlocks
layer(nbit::Int, ::Val{:last}) = chain(nbit, put(i=>chain(Rz(0), Rx(0))) for i = 1:nbit)
layer(nbit::Int, ::Val{:mid}) = chain(nbit, put(i=>chain(Rz(0), Rx(0), Rz(0))) for i = 1:nbit);
entangler(pairs) = chain(control(ctrl, target=>X) for (ctrl, target) in pairs);
function build_circuit(n, nlayers, pairs)
circuit = chain(n)
push!(circuit, layer(n, :first))
for i in 2:nlayers
push!(circuit, cache(entangler(pairs)))
push!(circuit, layer(n, :mid))
end
push!(circuit, cache(entangler(pairs)))
push!(circuit, layer(n, :last))
return circuit
end
build_circuit(4, 1, [1=>2, 2=>3, 3=>4])

Variational Quantum Eigensolver with Fewer Qubits
arXiv: 1902.02663



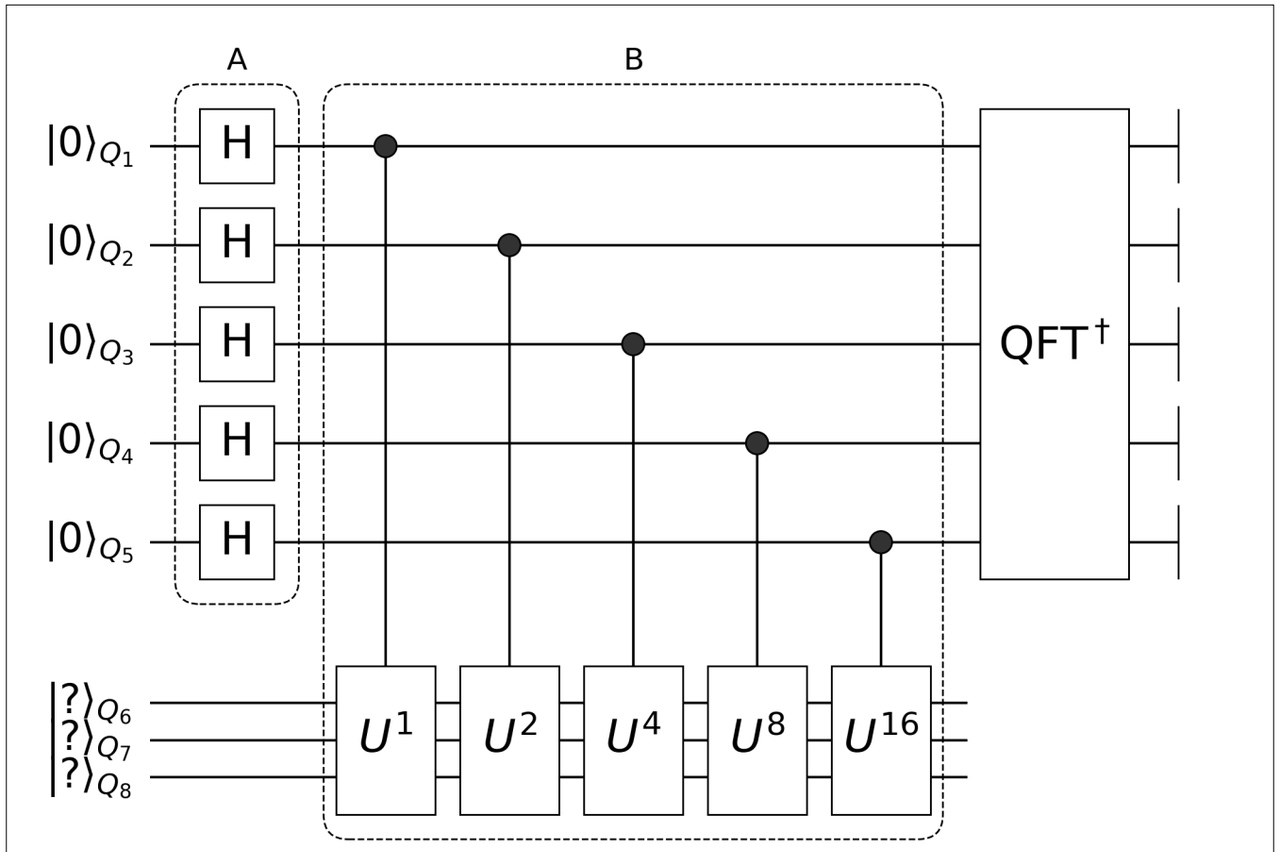
However, using multi-threading won't actually help in a general quantum circuit simulation, everyone know it is very hard and theoretically impossible for large size, since it grows exponentially anyway, and for deep small quantum circuits, the operations have orders, thus parallel won't help.

So we actually supports to define a batch of registers in the framework, and allows you to put them on GPU for the simulation.

But for some certain task, like this one, which requires a lot measurement in the middle, can be accelerate a lot by using a batch of registers, which is inspired by today's deep learning frameworks: if you try to divide your task into batch of operations, it can be accelerate by CUDA a lot.

```
julia> @benchmark zero_state(n, 1000) |> cu |> $(qcbm.circuit) seconds = 2
BenchmarkTools.Trial:
  memory estimate:  16.70 MiB
  allocs estimate:  7278
  --------------
  minimum time:     4.623 ms (0.00% GC)
  median time:      10.226 ms (8.24% GC)
  mean time:        11.168 ms (9.86% GC)
  maximum time:     81.029 ms (89.50% GC)
  --------------
  samples:          180
  evals/sample:     1

julia> @benchmark zero_state(n, 1000) |> $(cqcbm.circuit) seconds = 2
BenchmarkTools.Trial:
  memory estimate:  8.02 MiB
  allocs estimate:  2478
  --------------
  minimum time:     345.571 ms (0.00% GC)
  median time:      360.031 ms (0.00% GC)
  mean time:        358.910 ms (0.70% GC)
  maximum time:     369.374 ms (4.10% GC)
  --------------
  samples:          6
  evals/sample:     1
```

Which looks like this, this is a benchmark on batched register on qcbm on a V100 machine, you can use a single function cu to turn the register to GPU memory and it make the qcbm about 70 times faster.



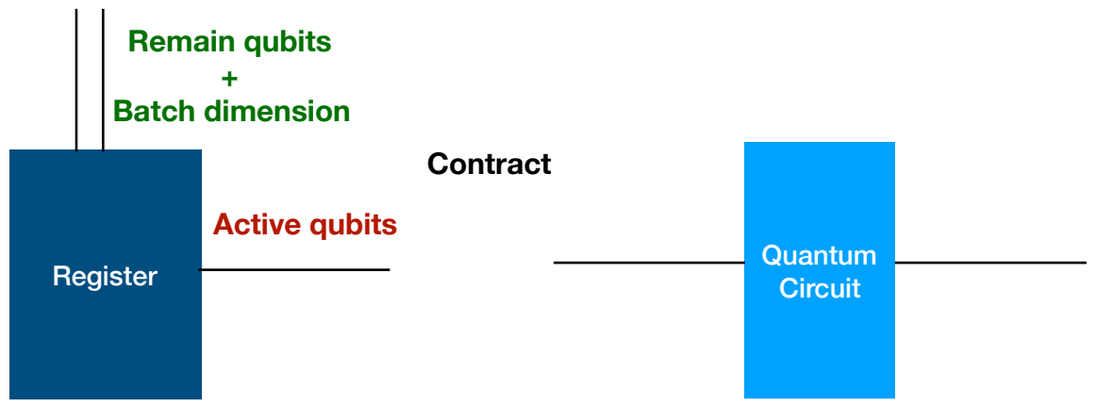
Quantum Circuit for Phase Estimation

However, previous design won't work well if we want to define things like phase estimation, imagine if we have defined our own QFT block and want to share it with Viral, and Viral doesn't know anything about the implementation details and he doesn't want to. But we need to apply this block defined on n qubits to n+m qubits.

How do we do this? We make qubit scopes, some qubits are activated in current scope, some are not, we call them the activated qubits.

But how can we just apply the block defined on n qubits

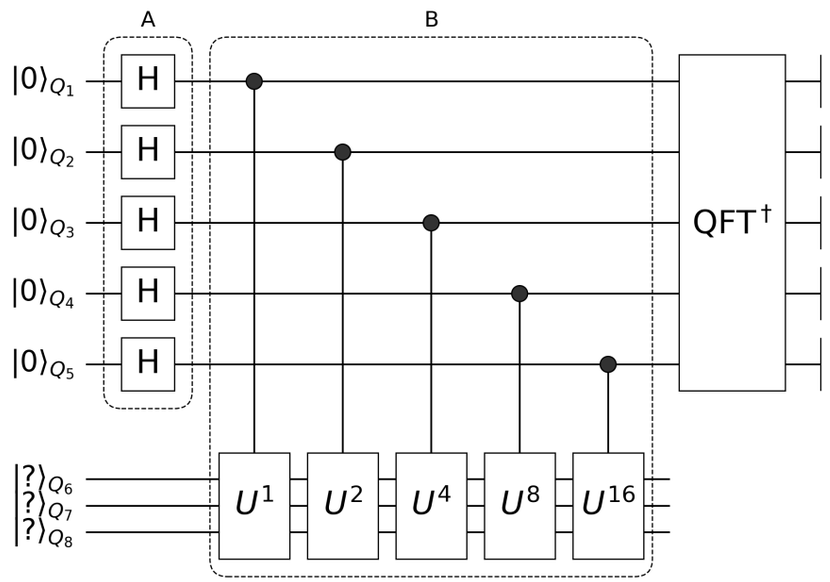
$$|\psi_k\rangle = \sum_{x,y} L(x, y, \dots)\,|j\rangle\,|i\rangle$$



Our register actually has 3 dimension in principal, which are

1. The activated qubit dimension
2. The remain qubit dimension
3. The batch dimension

It always contract with the quantum circuit which is a large matrix on the active qubit dimension.



Quantum Circuit for Phase Estimation

```
julia> PE(n, m, U) =
           chain(n+m, # total number of the qubits
               concentrate(Hadamards(n), 1:n), # apply H in local scope
               ControlU(n, m, U),
               concentrate(QFT(n)', 1:n))
```

So we could just add an extra semantic in our IR, which is the concentrate, which provide a scope that inside it is a smaller scope on the register and outside it is the original scope it stays in, then only a few line of code, we can make this Phase estimation circuit.

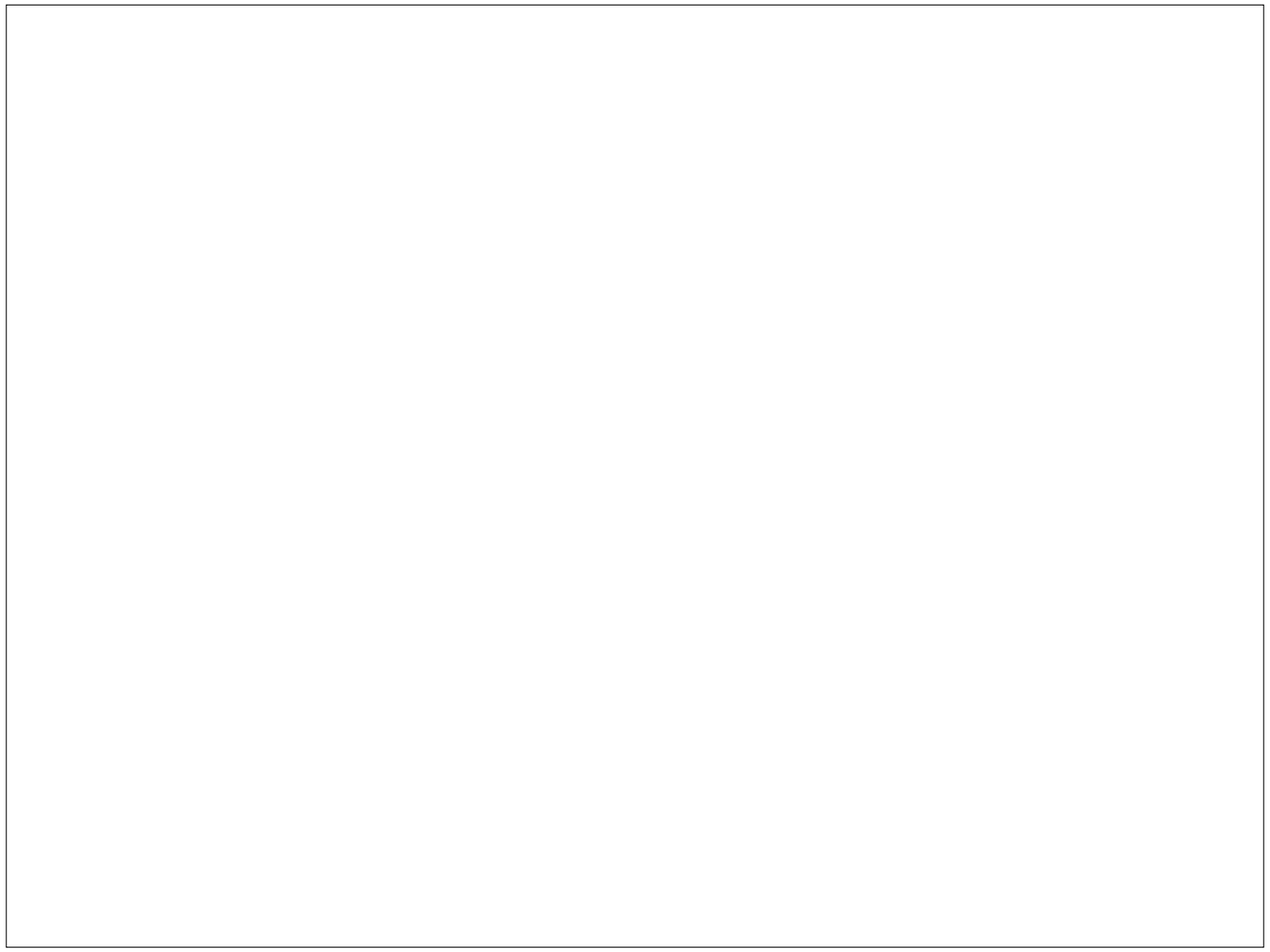
Let’s try it out

So now, we could try it out, if you happen to bring your own laptop, I will demonstrate a very simple quantum circuit born machine demo which fits a gaussian distribution. You can also find the demo in our documentation.

Yao's Mission

So after talking about all these, what is Yao? Why we are developing it? And What kind of thing we want it to be? Before talking about this, I want to have a brief overview on what other people is working on.

Low level instructions



```
// Repetition code syndrome measurement
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
qreg a[2];
creg c[3];
creg syn[2];
gate syndrome d1,d2,d3,a1,a2
{
  cx d1,a1; cx d2,a1;
  cx d2,a2; cx d3,a2;
}
x q[0]; // error
barrier q;
syndrome q[0],q[1],q[2],a[0],a[1];
measure a -> syn;
if(syn==1) x q[0];
if(syn==2) x q[2];
if(syn==3) x q[1];
measure q -> c;
```

QASM

```
// Repetition code syndrome measurement
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
qreg a[2];
creg c[3];
creg syn[2];
gate syndrome d1,d2,d3,a1,a2
{
  cx d1,a1; cx d2,a1;
  cx d2,a2; cx d3,a2;
}
x q[0]; // error
barrier q;
syndrome q[0],q[1],q[2],a[0],a[1];
measure a -> syn;
if(syn==1) x q[0];
if(syn==2) x q[2];
if(syn==3) x q[1];
measure q -> c;
```

QASM

Quil

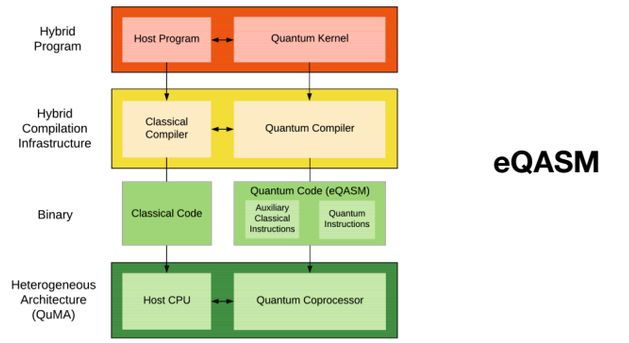
```
DEFCIRCUIT FOO:
    LABEL @FOO_A
    JUMP @GLOBAL   # valid, global label
    JUMP @FOO_A    # valid, local to FOO
    JUMP @BAR_A    # invalid

DEFCIRCUIT BAR:
    LABEL @BAR_A
    JUMP @FOO_A    # invalid

LABEL @GLOBAL
FOO
BAR
JUMP @FOO_A        # invalid
JUMP @BAR_A        # invalid
```

```
if (*x) {
    // instrA...
} else {
    // instrB...
}
```

This can be translated into Quil in the following way:

```
JUMP-WHEN @THEN [x]
# instrB...
JUMP @END
LABEL @THEN
# instrA...
LABEL @END
```

```
// Repetition code syndrome measurement
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
qreg a[2];
creg c[3];
creg syn[2];
gate syndrome d1,d2,d3,a1,a2
{
  cx d1,a1; cx d2,a1;
  cx d2,a2; cx d3,a2;
}
x q[0]; // error
barrier q;
syndrome q[0],q[1],q[2],a[0],a[1];
measure a -> syn;
if(syn==1) x q[0];
if(syn==2) x q[2];
if(syn==3) x q[1];
measure q -> c;
```



eQASM

QASM

```
DEFCIRCUIT FOO:
    LABEL @FOO_A
    JUMP @GLOBAL    # valid, global label
    JUMP @FOO_A     # valid, local to FOO
    JUMP @BAR_A     # invalid

DEFCIRCUIT BAR:
    LABEL @BAR_A
    JUMP @FOO_A     # invalid

LABEL @GLOBAL
FOO
BAR
JUMP @FOO_A         # invalid
JUMP @BAR_A         # invalid
```

Quil

```
if (*x) {
    // instrA...
} else {
    // instrB...
}
```

This can be translated into Quil in the following way:

```
JUMP-WHEN @THEN [x]
# instrB...
JUMP @END
LABEL @THEN
# instrA...
LABEL @END
```

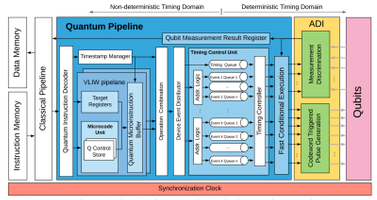


Fig. 9. Quantum microarchitecture implementing the instantiated eQASM for the seven-qubit superconducting quantum processor.

Languages
(Frontends)

Imperative

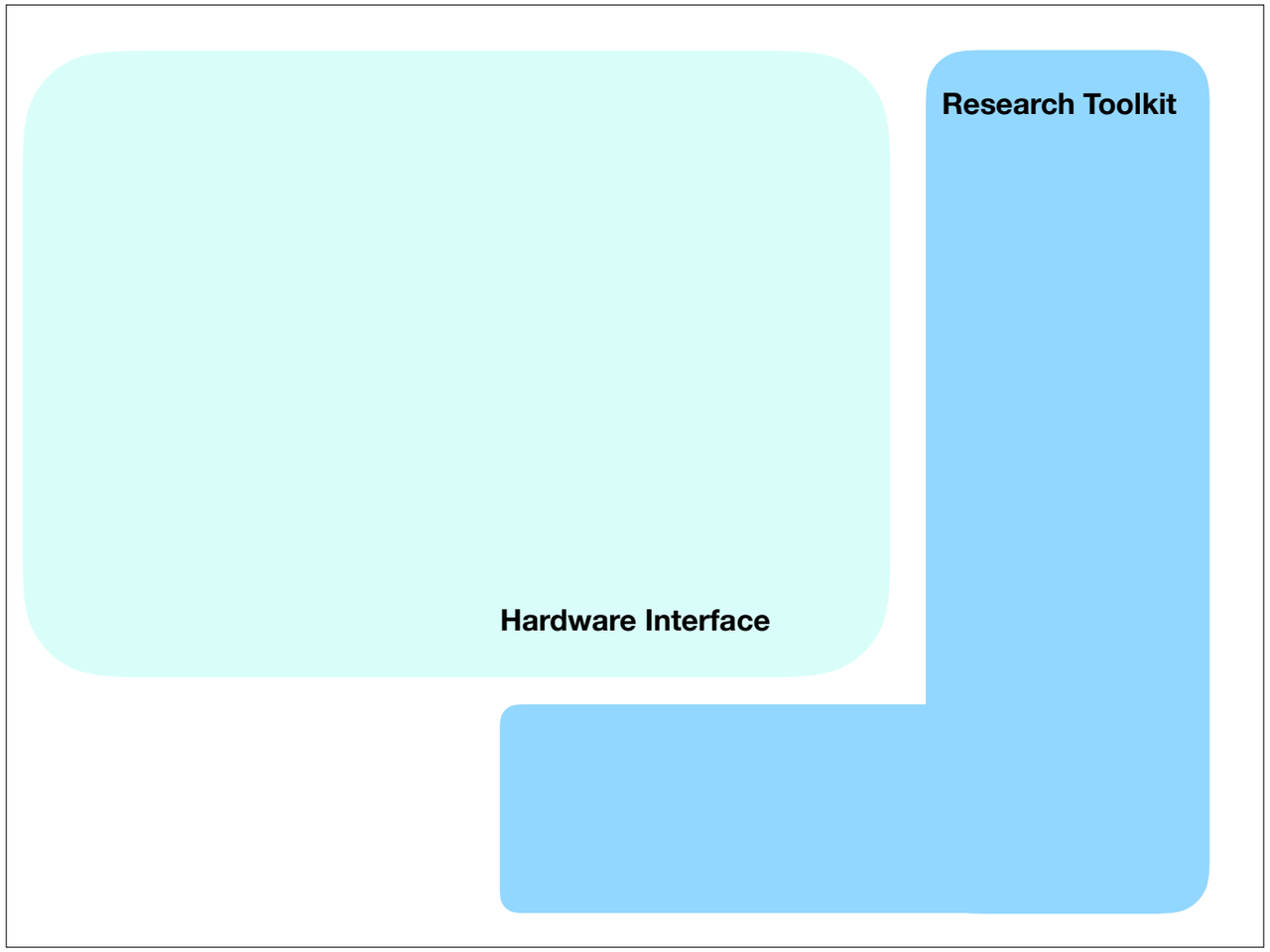
1. QCL
2. Q#
3. Quantum Pseudocode
4. QISI>
5. qGCL

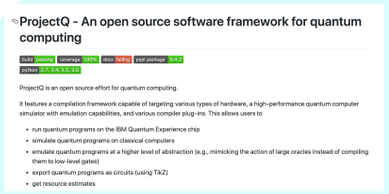
Functional

1. QFC/QPL
2. QML
3. LiQuil>
4. Quantum Lambda calculus
5. Quipper

SDKs

(Software Development Kits/Framework)


Research Toolkit
Hardware Interface



ProjectQ - An open source software framework for quantum computing

build passing coverage 100% docs failing pypi package 0.4.2 python 2.7, 3.4, 3.5, 3.6

ProjectQ is an open source effort for quantum computing.

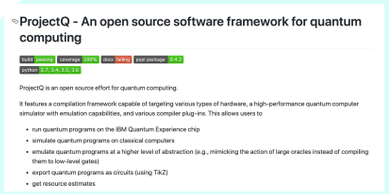
It features a compilation framework capable of targeting various types of hardware, a high-performance quantum computer simulator with emulation capabilities, and various compiler plug-ins. This allows users to

- run quantum programs on the IBM Quantum Experience chip
- simulate quantum programs on classical computers
- emulate quantum programs at a higher level of abstraction (e.g., mimicking the action of large oracles instead of compiling them to low-level gates)
- export quantum programs as circuits (using TikZ)
- get resource estimates

ProjectQ

Hardware Interface

Research Toolkit



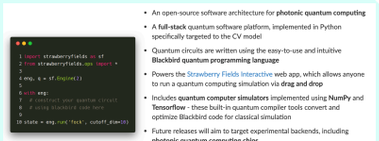
ProjectQ - An open source software framework for quantum computing

ProjectQ is an open source effort for quantum computing.

It features a compilation framework capable of targeting various types of hardware, a high-performance quantum computer simulator with emulation capabilities, and various compiler plug-ins. This allows users to

- run quantum programs on the IBM Quantum Experience chip
- simulate quantum programs on classical computers
- emulate quantum programs at a higher level of abstraction (e.g., mimicking the action of large oracles instead of compiling them to low-level gates)
- export quantum programs as circuits (using TikZ)
- get resource estimates

ProjectQ

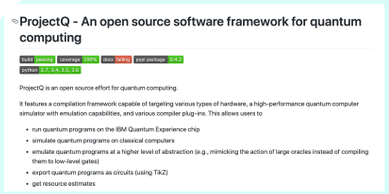


- An open-source software architecture for **photonic quantum computing**
- A **full-stack** quantum software platform, implemented in Python specifically targeted to the CV model
- Quantum circuits are written using the easy-to-use and intuitive **Blackbird quantum programming language**
- Powers the Strawberry Fields Interactive web app, which allows anyone to run a quantum computing simulation via **drag and drop**
- Includes **quantum computer simulators** implemented using **NumPy** and **Tensorflow** - these built-in quantum compiler tools convert and optimize Blackbird code for classical simulation
- Future releases will aim to target experimental backends, including **photonic quantum computing chips**

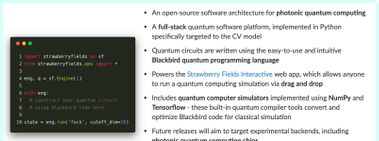
Strawberry Fields

Hardware Interface

Research Toolkit



ProjectQ



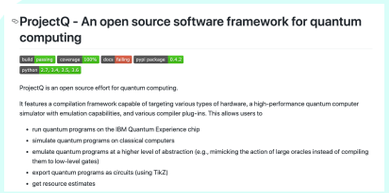
- An open-source software architecture for **photonic quantum computing**
- A **full-stack** quantum software platform, implemented in Python specifically targeted to the CV model
- Quantum circuits are written using the easy-to-use and intuitive **Blackbird quantum programming language**
- Powers the Strawberry Fields Interactive web app, which allows anyone to run a quantum computing simulation via **drag and drop**
- Includes **quantum computer simulators** implemented using **NumPy** and **Tensorflow** - these built-in quantum compiler tools convert and optimize Blackbird code for classical simulation
- Future releases will aim to target experimental backends, including **photonic quantum computing chips**

Strawberry Fields

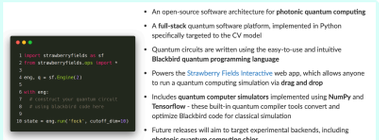
Forest

Hardware Interface

Research Toolkit

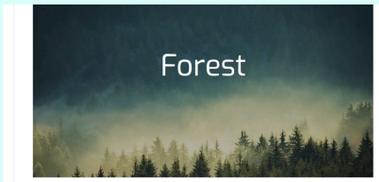


ProjectQ



- An open-source software architecture for **photonic quantum computing**
- A **full-stack** quantum software platform, implemented in Python specifically targeted to the CV model
- Quantum circuits are written using the easy-to-use and intuitive **Blackbird quantum programming language**
- Powers the Strawberry Fields Interactive web app, which allows anyone to run a quantum computing simulation via **drag and drop**
- Includes **quantum computer simulators** implemented using **NumPy** and **Tensorflow** - these built-in quantum compiler tools convert and optimize Blackbird code for classical simulation
- Future releases will aim to target experimental backends, including **photonic quantum computing chips**

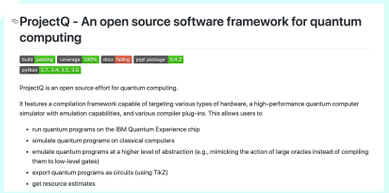
Strawberry Fields



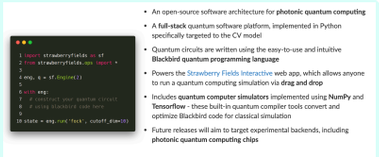
Forest

Hardware Interface

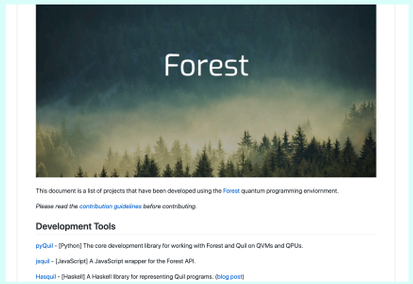
Research Toolkit



ProjectQ



Strawberry Fields

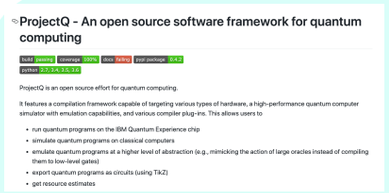


Forest

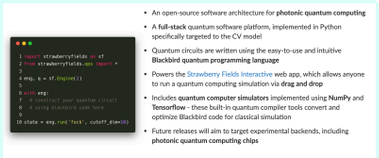
Hardware Interface

Research Toolkit

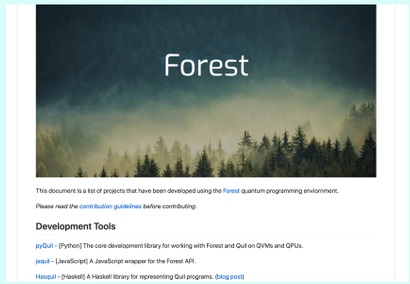
Qiskit



ProjectQ



Strawberry Fields

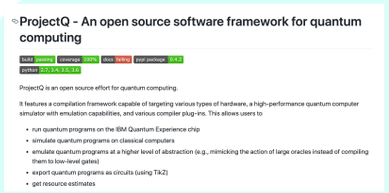


Forest

Hardware Interface

Research Toolkit

Qiskit



ProjectQ - An open source software framework for quantum computing

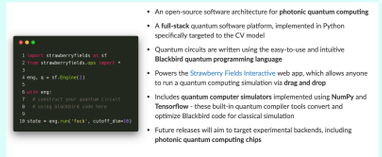
build passing coverage 100% docs failing pypi package 0.4.2 python 2.7,3.4,3.5,3.6

ProjectQ is an open source effort for quantum computing.

It features a compilation framework capable of targeting various types of hardware, a high-performance quantum computer simulator with emulation capabilities, and various compiler plug-ins. This allows users to

- run quantum programs on the IBM Quantum Experience chip
- simulate quantum programs on classical computers
- emulate quantum programs at a higher level of abstraction (e.g., mimicking the action of large oracles instead of compiling them to low-level gates)
- export quantum programs as circuits (using TikZ)
- get resource estimates

ProjectQ



- An open-source software architecture for **photonic quantum computing**
- A **full-stack** quantum software platform, implemented in Python specifically targeted to the CV model
- Quantum circuits are written using the easy-to-use and intuitive **Blackbird quantum programming language**
- Powers the Strawberry Fields Interactive web app, which allows anyone to run a quantum computing simulation via **drag and drop**
- Includes **quantum computer simulators** implemented using **NumPy** and **Tensorflow** - these built-in quantum compiler tools convert and optimize Blackbird code for classical simulation
- Future releases will aim to target experimental backends, including **photonic quantum computing chips**

Strawberry Fields

This document is a list of projects that have been developed using the Forest quantum programming enviornment.

Please read the contribution guidelines before contributing.

Development Tools

pyQuil - [Python] The core development library for working with Forest and Quil on QVMs and QPUs.

jaquil - [JavaScript] A JavaScript wrapper for the Forest API.

Hasquil - [Haskell] A Haskell library for representing Quil programs. (blog post)

Forest

Hardware Interface

Research Toolkit

Qiskit

The mission of Yao

Build and Test Quantum Algorithms

We don’t know whether there will be a practical quantum algorithm shows the quantum advantage, but as researcher, we want to explore this by designing more and more algorithms. But as long as the algorithm become more and more complicated we need tools to ease our brain.

As what I just introduced, every feature in Yao has a practical algorithm paper behind it, we implement it because it is useful in implementing the paper. And in fact you can find our playground of algorithms in QuAlgorithmZoo, it is a bit messy, but there are quite a few new algorithms implemented with Yao there.

So what kind of feature do we care about? I list several.

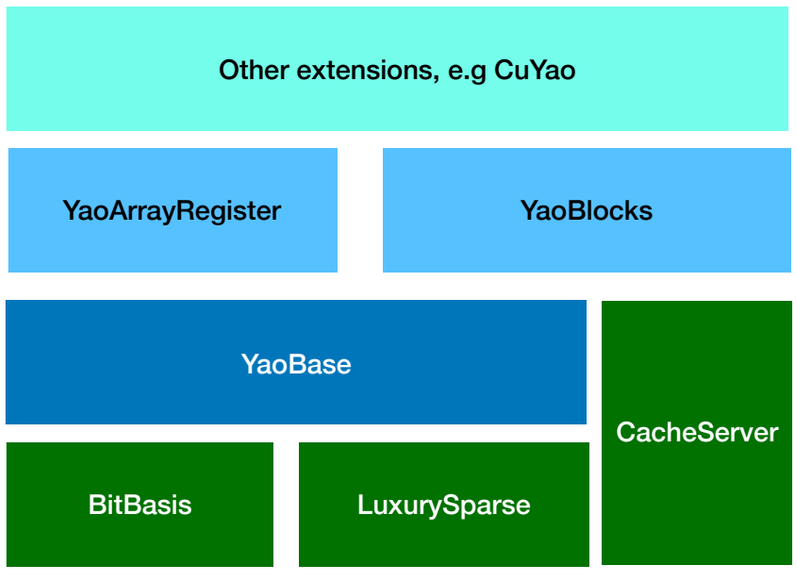
Build and Test Quantum Algorithms

- Simulation on laptop matters: 5~20 qubits
- Abstraction of quantum circuits
- Extensibility to support new algorithms
- Readability: easy to understand what's happening (no black box abstractions)
- Share: it should be easy to share, like a package

(read the first 3 features)

So the first 3 features have been solved by the solution I talked about, what about the rest?

Architecture



To make it easier for sharing, we divide the whole framework into a few component packages, which means the package Yao itself is just a meta package, there is nothing in it. And if you want to make something of your own, you don’t have to open a PR to the core repo, you can just make your own package, and let it depend on the component package you need to.

And the Julia package manager will take care the rest.

(explain which package is for what here)

```
julia> using Yao

julia> apply!(rand_state(2), put(1=>X))
ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 2/2

julia> @edit apply!(rand_state(2), put(1=>X))

julia>
```

Since Yao is implemented in pure Yao with a very hierarchical APIs, it is completely transparent, which means you can inspect any piece of code from the highest level to the lowest level.

For example we could check how X gate is applied to the first qubit of this random state, we only need an edit macro which ships with Julia itself, and we can dig into the implementation, where apply is a middle level API defined in YaoBlocks, and the instruct function inside is a low level API defined in YaoArrayRegister for simulation.

```
julia> using Yao

julia> apply!(rand_state(2), put(1=>X))
ArrayReg{1, Complex{Float64}, Array...}
    active qubits: 2/2

julia> @edit apply!(rand_state(2), put(1=>X))

julia>
```

```
function apply!(r::ArrayReg{B, T}, pb::PutBlock{N}) where {B, T, N}
    _check_size(r, pb)
    instruct!(matvec(r.state), mat(T, pb.content), pb.locs)
    return r
end
```

```
julia> using Yao

julia> apply!(rand_state(2
ArrayReg{1, Complex{Float6
    active qubits: 2/2

julia> @edit apply!(rand_s

julia>
```

```
function YaoBase.instruct!(state::AbstractVecOrMat, ::Val{:X}, locs::NTuple{N, Int}) where N
    mask = bmask(locs)
    do_mask = bmask(first(locs))
    for b in basis(state)
        @inbounds if anyone(b, do_mask)
            i = b+1
            i_ = flip(b, mask) + 1
            swaprows!(state, i, i_)
        end
    end
    return state
end
```

```
function apply!(r::ArrayReg{B, T}, pb::PutBlock{N}) where {B, T, N}
    _check_size(r, pb)
    instruct!(matvec(r.state), mat(T, pb.content), pb.locs)
    return r
end
```

On going projects

We also have some on going projects that you might be interested in expecting

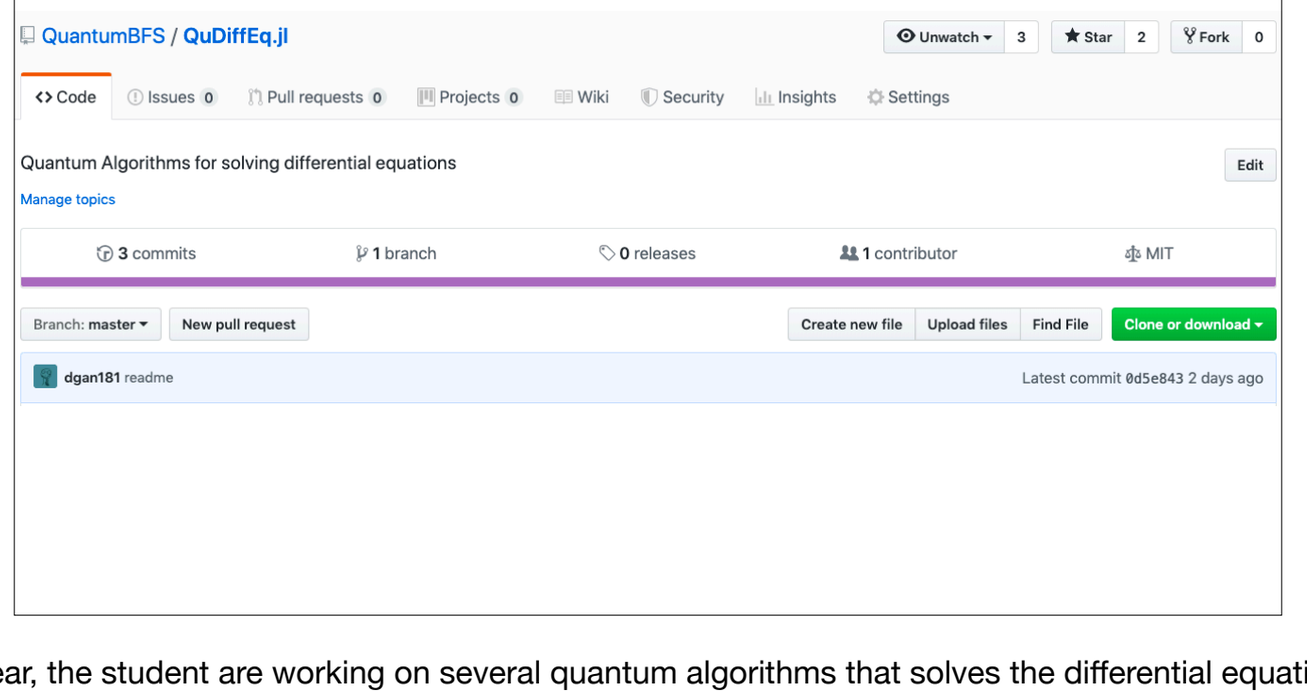
Full AD support on Quantum Circuit

```
using Yao, Zygote, Flux.Optimise
# make a one qubit GHZ state as learning target
# NOTE: this can be an arbitrary one qubit state
t = ArrayReg(bit"0") + ArrayReg(bit"1")
normalize!(t)

# calculate the fidelity with GHZ state
function fid(xs)
    r = zero_state(1)
    U = mat(chain(Rx(xs[1]), Rz(xs[2])))
    return abs(statevec(t)' * U * statevec(r))
end

# simply tell Zygote to get the gradient and start training
function train()
    opt = ADAM()
    xs = rand(2)
    for _ in 1:1000
        println(fid(xs))
        Optimise.update!(opt, xs, -fid'(xs))
    end
    return xs
end

train()
```

We just made two proof of principle demo to the Zygote paper. So Zygote is a source to source automatic differentiation package, it can take the source code and analysis the program and generate the adjoint program, which is the gradient. In this demo, we didn't define anything specific of the gradient, just some patches to workaround current Zygote bugs. And it just work.

We are planning to integrate with Zygote, so we could have full featured AD on any object in Yao, which means things like quantum architecture search, integration with neural networks like Flux can be much easier than before.

JSoC 2019: Quantum ODE solver for DiffEq.jl



This is a summer of code project this year, the student are working on several quantum algorithms that solves the differential equation on quantum circuits. It might not be very useful for now, but it is cool, and it make it possible to do more complicated ideas to make the research on this direction to conduct new results easier.

Roadmap

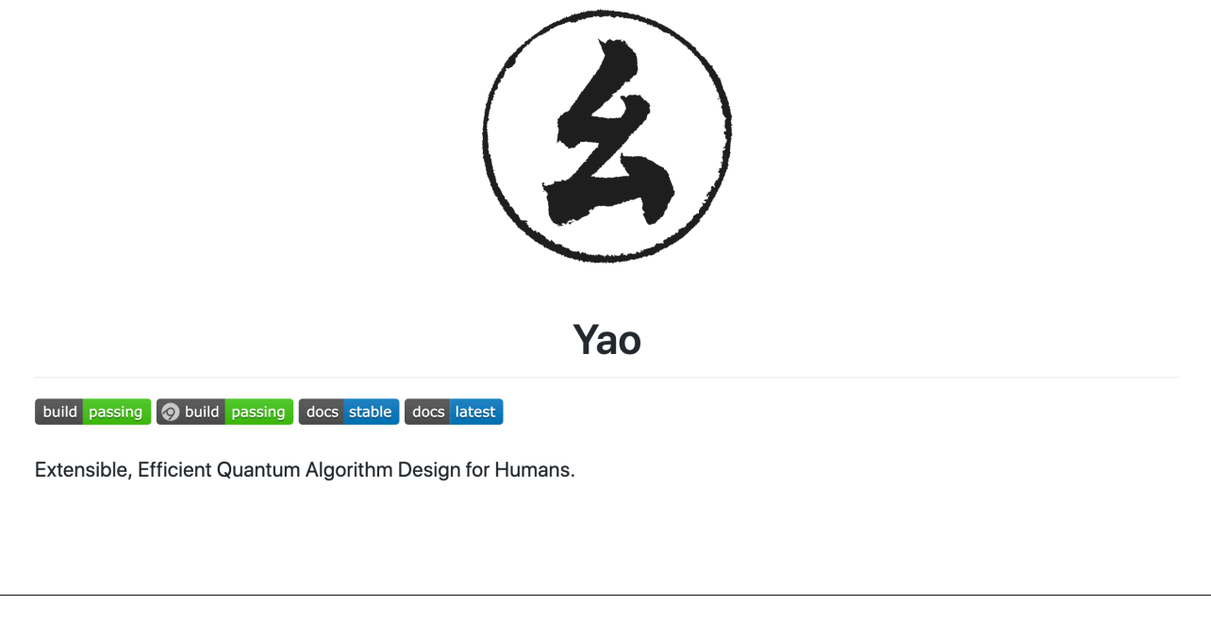
- Quantum circuit architecture optimization
- Tensor Network support
- Experiment validation
- Circuit Plotting & visualization

Ordered by priority

So, Yao is an open source project to bind people together to share your work and your idea, and we would like to have a community around it, and we want it benefits real practical research rather than just commercial advertisement. As long as someone finds it useful for their own research than it worth the effort.

And it always lack of people

Star us
Github: QuantumBFS/Yao.jl



Yao

Extensible, Efficient Quantum Algorithm Design for Humans.

If you find it useful, could star us on github!