Julia in Qiskit

Sahar Ben Rached, Rafał Pracht

Mentored by John Lapeyre and Jim Garrison

Qiskit

Julia

Qiskit

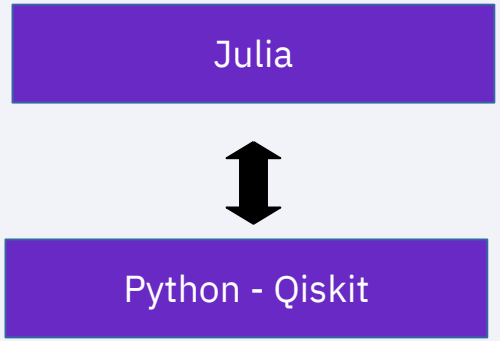
Julia is a high-level, high-performance, dynamic programming language. While it is a general-purpose language and can be used to write any application, many of its features are well suited for numerical analysis and computational science.

Julia achieved "peak performance of 1.54 petaFLOPS using 1.3 million threads" on 9300 Knights Landing (KNL) nodes of the Cori II (Cray XC40) supercomputer. **Julia** thus joins **C**, **C++**, and **Fortran** as high-level languages in which petaFLOPS computations have been achieved.

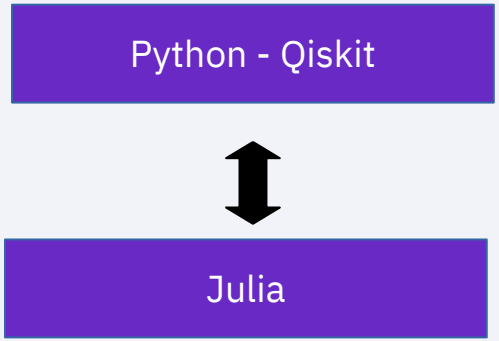
Possibility of use Julia in Qiskit

Qiskit

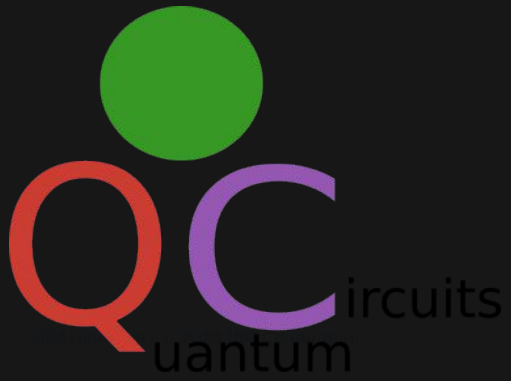
QuantumCircuits



Qiskit-Alt



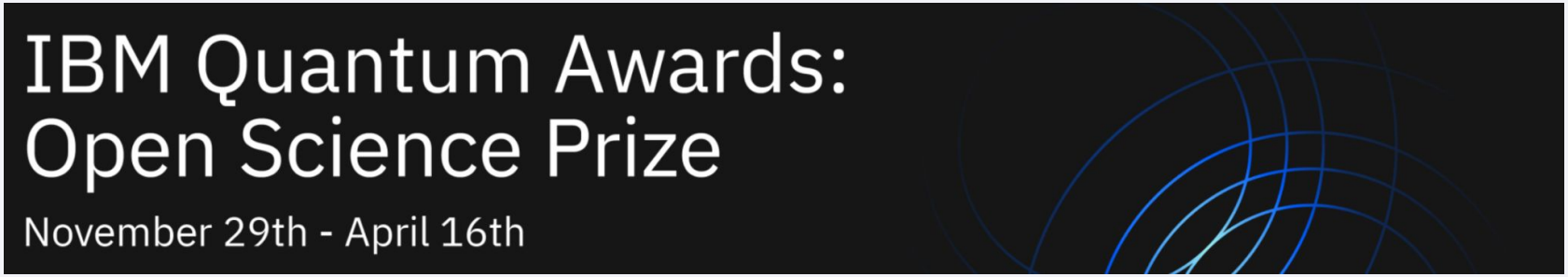
QuantumCircuits library

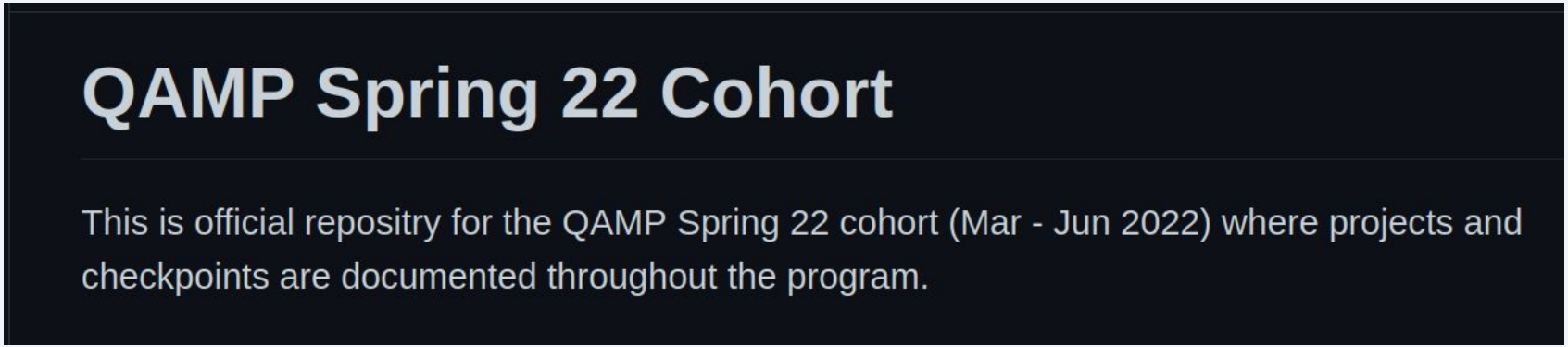


Qiskit

QuantumCircuits Library

Qiskit





QuantumCircuits Library

open-source - Apache License 2.0

src:

Language	files	blank	comment	code
Julia	21	729	896	2414
SUM:	21	729	896	2414

examples:

Language	files	blank	comment	code
Julia	3	101	131	176
SUM:	3	101	131	176

test:

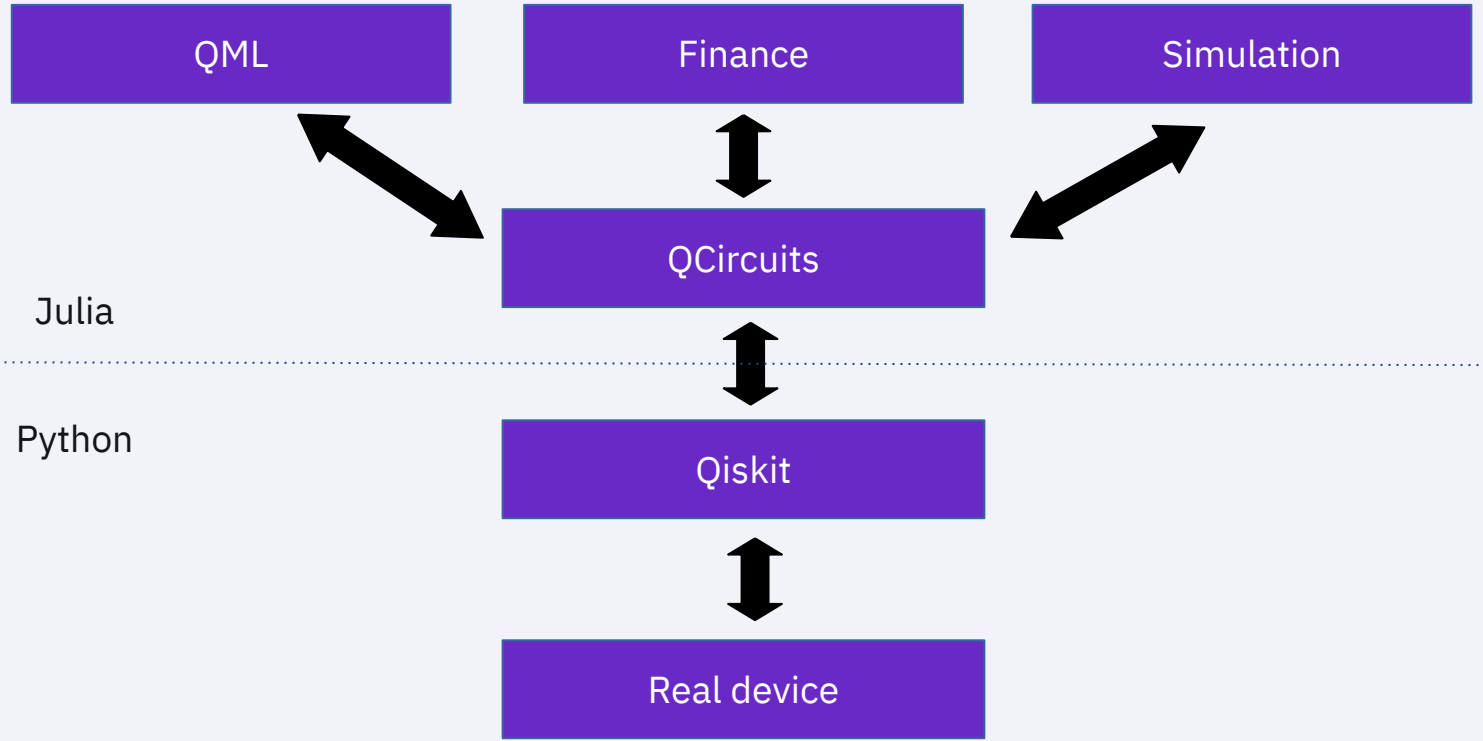
Language	files	blank	comment	code
Julia	24	436	461	1471
Lisp	1	8	0	31
TOML	1	0	0	4
SUM:	26	444	461	1506

docs:

Language	files	blank	comment	code
CSS	2	685	49	14706
Markdown	23	189	0	1071
HTML	23	54	0	800
JavaScript	5	52	82	575
TOML	2	85	1	329
Julia	1	4	4	54
SUM:	56	1069	136	17535

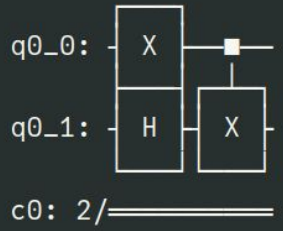
QuantumCircuits Architecture

Qiskit

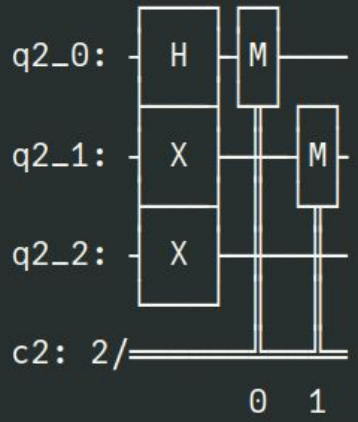


```
# We use the simulator written in Julia
const backend = QuantumSimulator()

# Let's create an example circuit.
qc1 = QCircuit(2)
qc1.x(0)
qc1.h(1)
qc1.cx(0, 1)
qc1
```




```
qr = QuantumRegister(3)
cr = ClassicalRegister(2)
qc = QCircuit(qr, cr)
qc.h(0)
qc.x(1)
qc.x(2)
qc.measure([0, 1], [0, 1])
qc
```



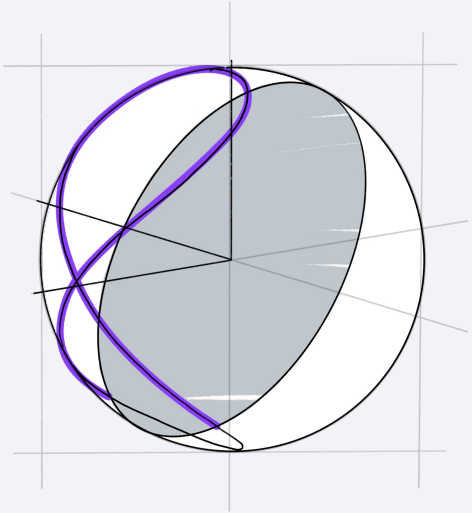
Qiskit

see: https://adgnitio.github.io/QuantumCircuits.jl/dev/

QML

Qiskit

Using a parameter-shift rule to calculate the derivatives is slow compared to automate gradient calculations. In QuantumCircuits I use the Zygote library to calculate the gradient of the circuits. Thanks to this, the QML algorithms run much faster on the simulator.



Currently, according to my knowledge, Qiskit doesn’t allows using AD to calculate the derivatives.

And, I don't know if technically is possible to automate calculate gradient in python from C++ written symulator.

Quantum Monte Carlo

In Quantum Monte Carlo, the algorithm used by quantum finance to derivative pricing we need to prepare the quantum state where we encode the probabilities of the underlying asset prices.

But to prepare a quantum state in general, an exponential number of gates, O(2^n), are needed.

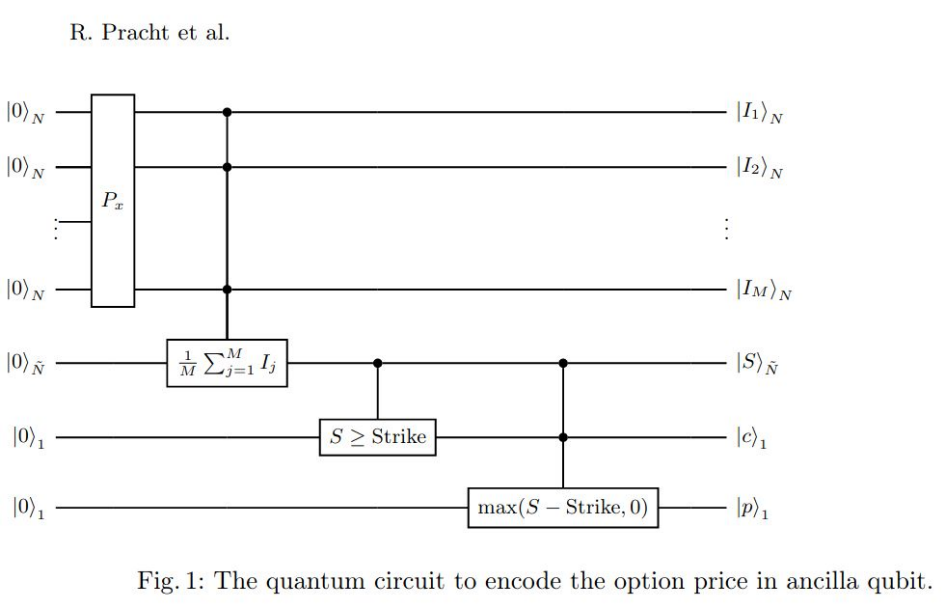


Fig. 1: The quantum circuit to encode the option price in ancilla qubit.

see: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4137397

Log-Normal state preparation

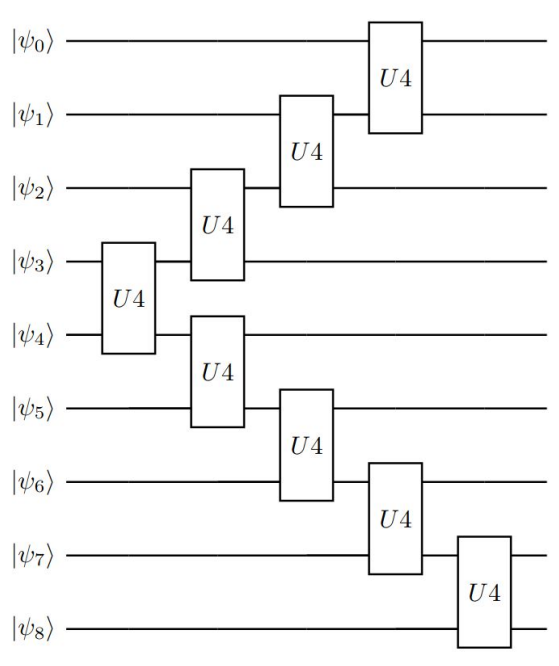
Qiskit

```
# Generate ansact
qr = QuantumRegister(N)
cr = ClassicalRegister(N)
qc = QCircuit(qr, cr)
#add_ent_gate(i, j) = qc.cx(i, j)
add_ent_gate(i, j) = qc.u4(i, j)

#qc.x(4)
for i in 0:3
    add_ent_gate(4-i, 4-i-1)
    add_ent_gate(4+i, 4+i+1)
end

# decompose
qc = decompose(qc)
qc.measure(0:N-1, 0:N-1)

# Random parameter
params = getRandParameters(qc)
setparameters!(qc, params)
```




```
val, x, itr = gradientDescent(loss_stage1, loss_stage1', params, α=0.0001, maxItr=500, debug=tru
                              useBigValInc=true, argsArePeriodic=true)
```

see: https://github.com/Adgnitio/QuantumCircuits.jl/blob/main/examples/lognormal_state_preparation_simple.jl

Zygote vs parameter-shift rule time comparison.

Qiskit

The calculation of derivative for loss function in Quantum Circuits took **1.37** seconds.

The same using Qiskit QasmSimulator took **8** seconds (shots=8192).

Julia calculates the derivative 6 times faster than the parameter-shift rule.

```
# number of parameters
length(params) # 99 | 99
# time of julia derivatives
@time loss'(params) | 99-element Vector{Float64}:
# 1.371532 seconds (61.62 k allocations: 1.857 GiB, 6.16% gc time)

@time qderivative(qiskitBackendSim, qc, mse_error_loss, params, corrMes=false)
# Job ID: 2374456f-ddda-4d4a-a8de-5f29a55d998c run 199 circuits.
# Job Status: job has successfully run
# Job 2374456f-ddda-4d4a-a8de-5f29a55d998c took 8 078 milliseconds.

8.078 / 1.371532 | 5.889764146953917
# 5.889764146953917 times faster
```

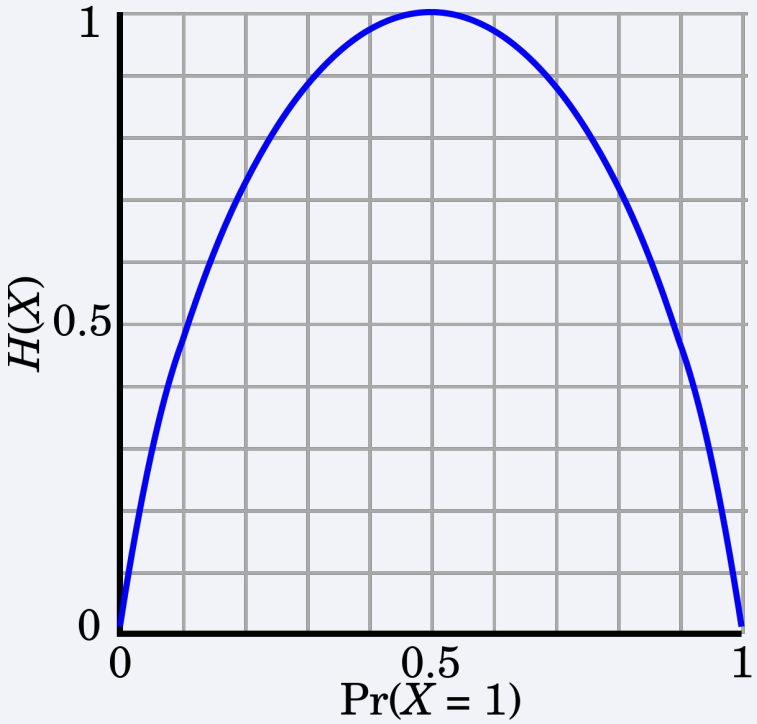
see: https://github.com/Adgnitio/QuantumCircuits.jl/blob/main/examples/lognormal_state_preparation_simple.jl

Stability.

$$\nabla_\theta f(x;\theta) = \frac{1}{2}\left[f(x;\theta+\frac{\pi}{2}) - f(x;\theta-\frac{\pi}{2})\right]$$

In my QML
problem for IBM Quantum Awards, Open Science Prize, I need about execute the job **10** or **20** times, to have the needed stability of the derivatives.

For my problem, Julia code is 60-120 times faster than the parameter-shift rule.



QML

The next use case is using the QML to find the parameters of Anzac that minimize the error of the difference between the unitary of the expected circuit and the Anzac.

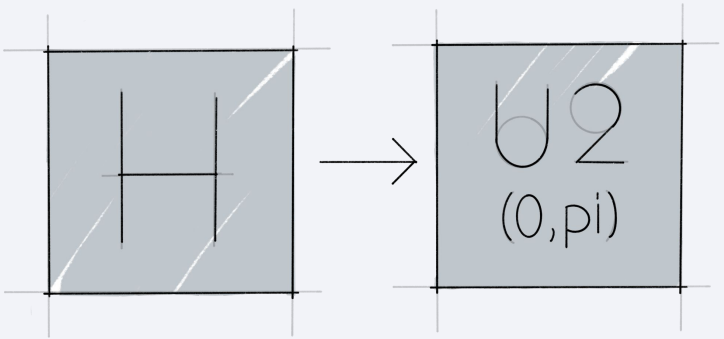
- We minimize the unitary error, so the Anzac will work in the same way as the expected circuits for any input state.
- This can be useful if we can replace some part of the algorithm with the Anzac with the same unitary but with less depth.

```
"Calculate the unitary matrix error"
unitary_error(exp_unit, mat_unit) = _unitary_error(exp_unit, mat_unit)
function unitary_error(exp_unit, mat_unit, exp_scale, mat_scale)
    return _unitary_error(exp_unit, mat_unit, x -> x * exp_scale, x -> x * mat_scale)
end
function _unitary_error(exp_unit, mat_unit, fa=identity, fb=identity)
    return sum(((a, b),) -> abs2(fa(a) - fb(b)), zip(exp_unit, mat_unit))
end

"Calculate the observe unitary matrix error, the error which we can observe
from quantum state"
function observe_unitary_error(exp_unit, mat_unit, index=1)
    # Remove phase
    exp_scale = cis(-angle(exp_unit[index]))
    mat_scale = cis(-angle(mat_unit[index]))

    return unitary_error(exp_unit, mat_unit, exp_scale, mat_scale)
end

"Calculate the minimum observe unitary matrix error."
function min_observe_unitary_error(exp_unit, mat_unit)
    l = length(exp_unit)
    return minimum(i -> observe_unitary_error(exp_unit, mat_unit, i), 1:l)
```



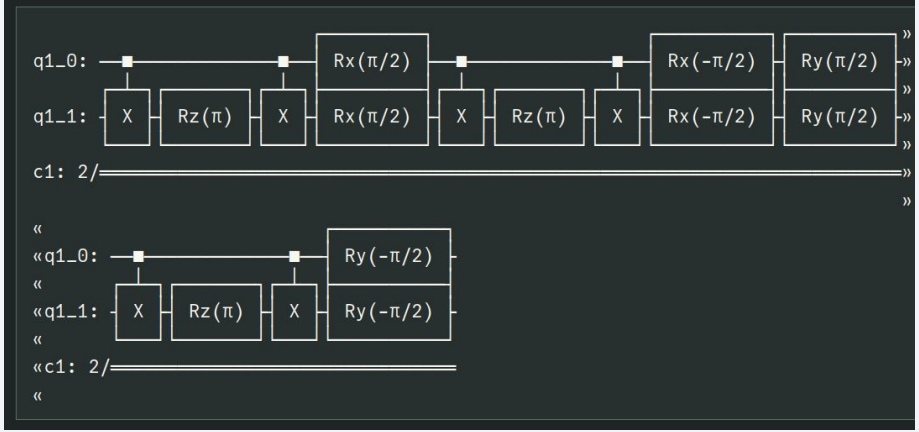
According to my knowledge, in Qiskit, there is no easy method to define loss function comparing two unitaries.

Cartan's KAK decomposition

Qiskit

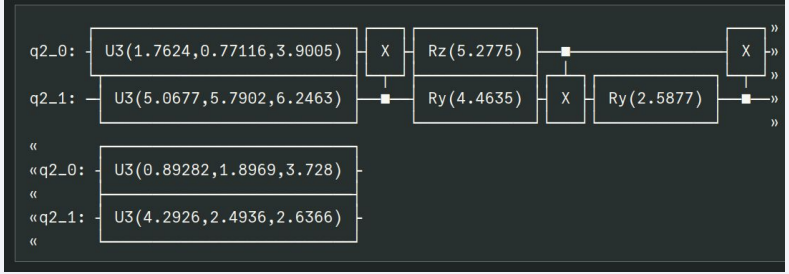
```
t = π/2

qc = QCircuit(2)
ZZ(qc, 0, 1, t)
YY(qc, 0, 1, t)
XX(qc, 0, 1, t)
expmat = tomatrix(qc)
qc
```




```
qr = QuantumRegister(2)
qc = QCircuit(qr)
qc.u4(qr[0], qr[1])

params = getRandParameters(qc)
setparameters!(qc, params)
qc = decompose(qc)
```



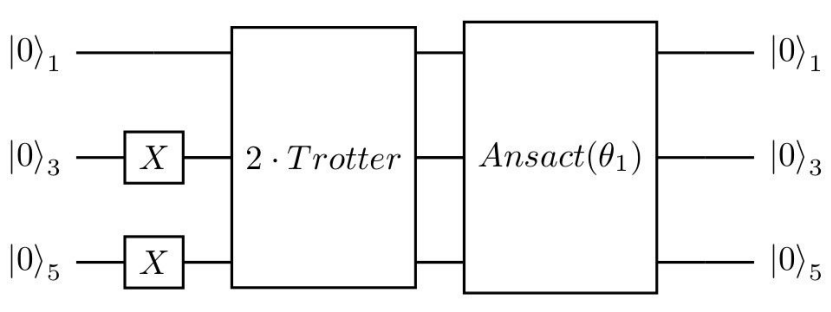

```
params, _, err, _  = findparam(expmat, qc, debug=false, trystandard=false)
err
```

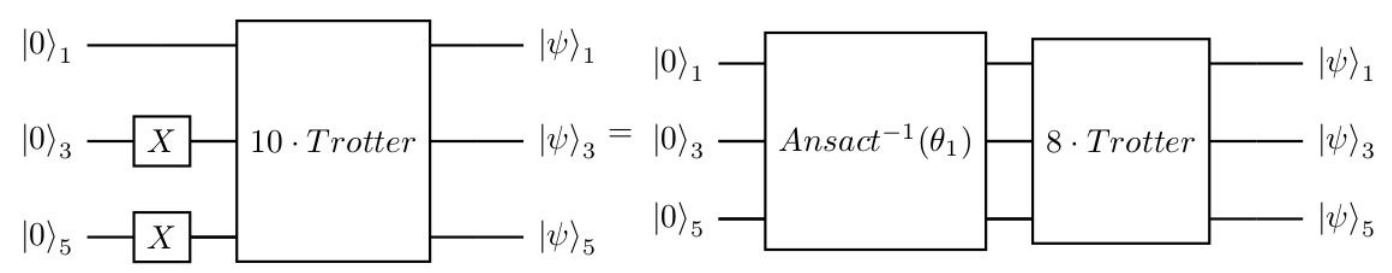
```
8.988143676440324e-8
```

see: https://adgnitio.github.io/QuantumCircuits.jl/dev/Simulation/U4_Cartan_decomposition/

Trotterization

Qiskit

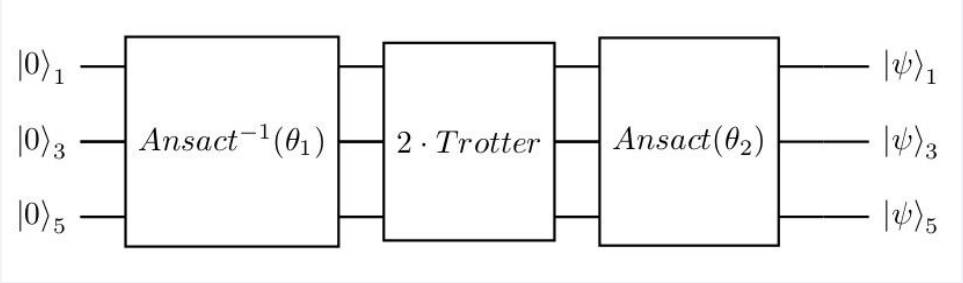


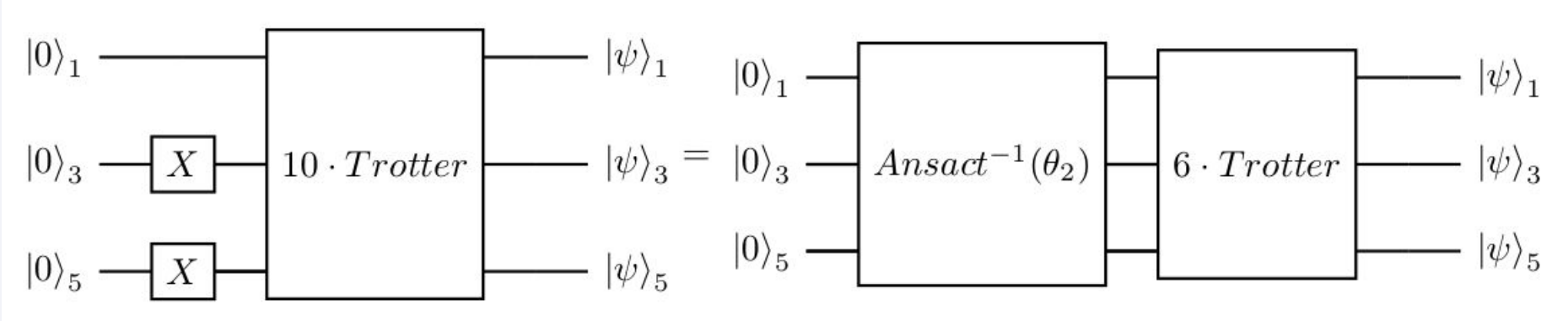


see: https://adgnitio.github.io/QuantumCircuits.jl/dev/Simulation/Trotterization/
https://github.com/rafal-pracht/OpenScience21.jl/blob/main/01_solution.ipynb

Trotterization

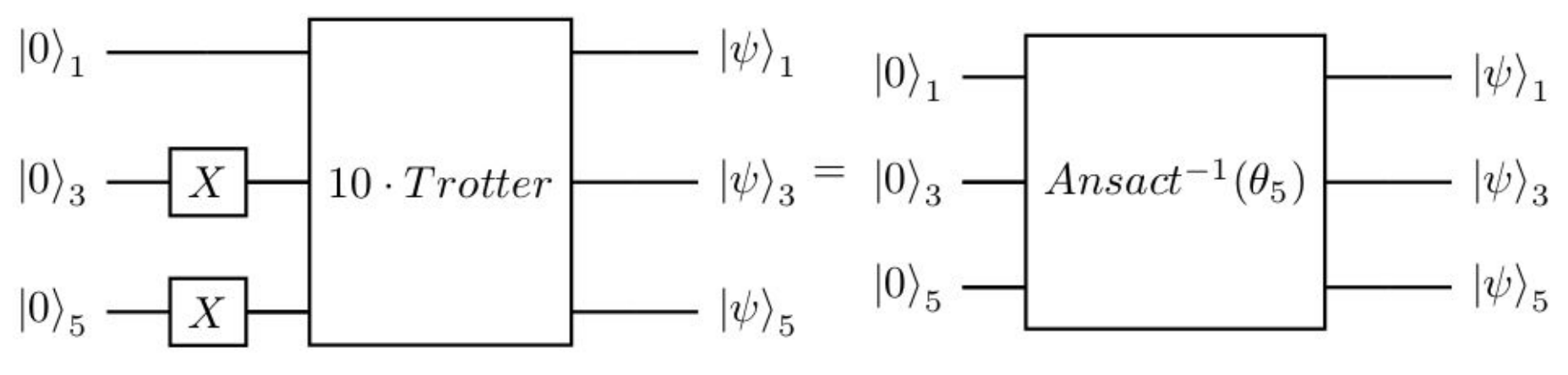
Qiskit



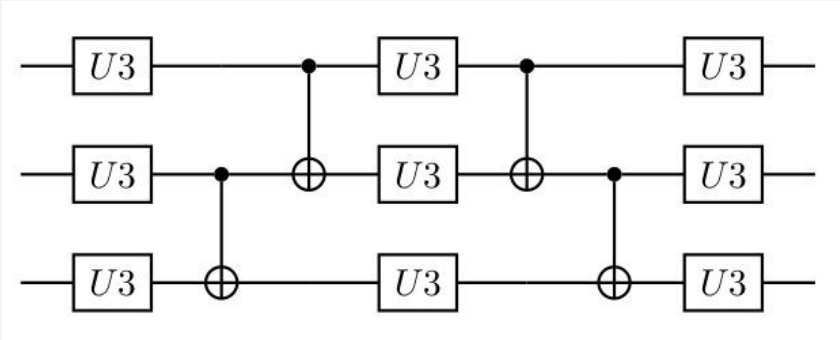


see: https://adgnitio.github.io/QuantumCircuits.jl/dev/Simulation/Trotterization/
https://github.com/rafal-pracht/OpenScience21.jl/blob/main/01_solution.ipynb

Trotterization



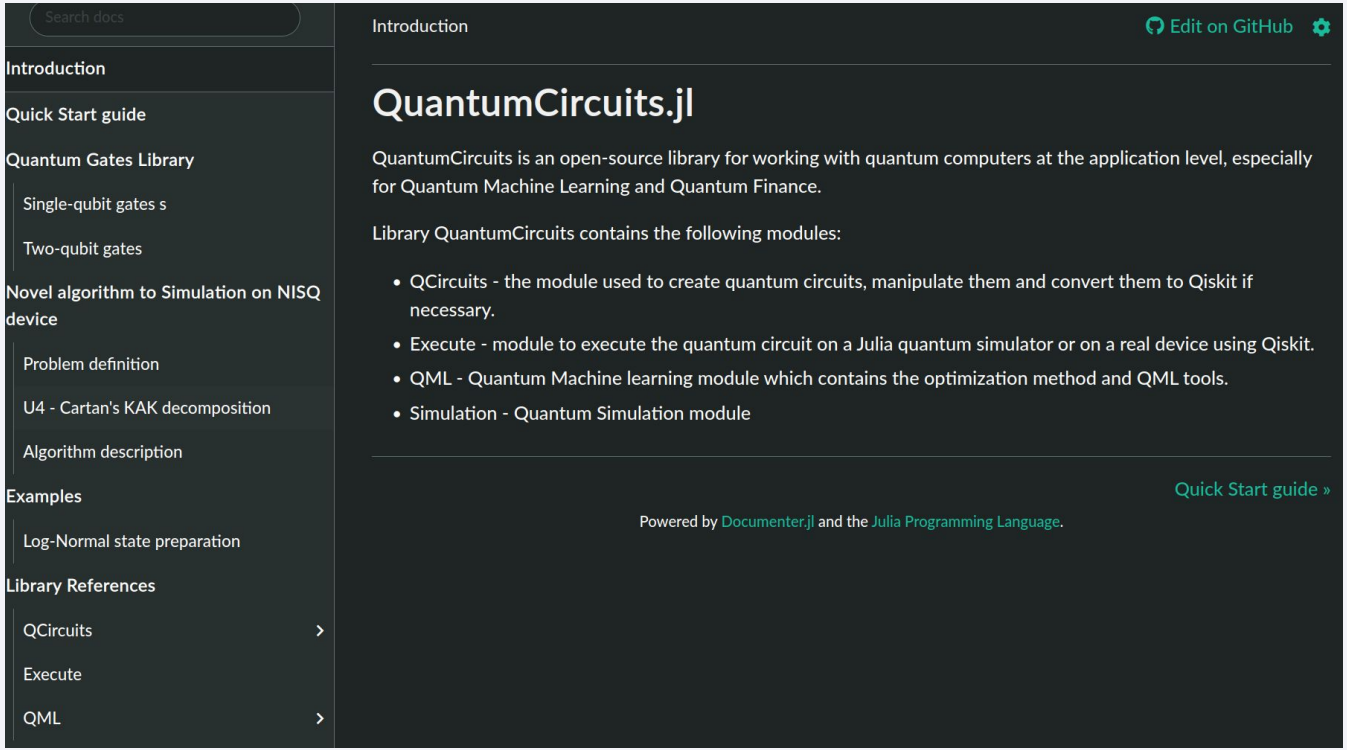
Anzac:



see: https://adgnitio.github.io/QuantumCircuits.jl/dev/Simulation/Trotterization/
https://github.com/rafal-pracht/OpenScience21.jl/blob/main/01_solution.ipynb

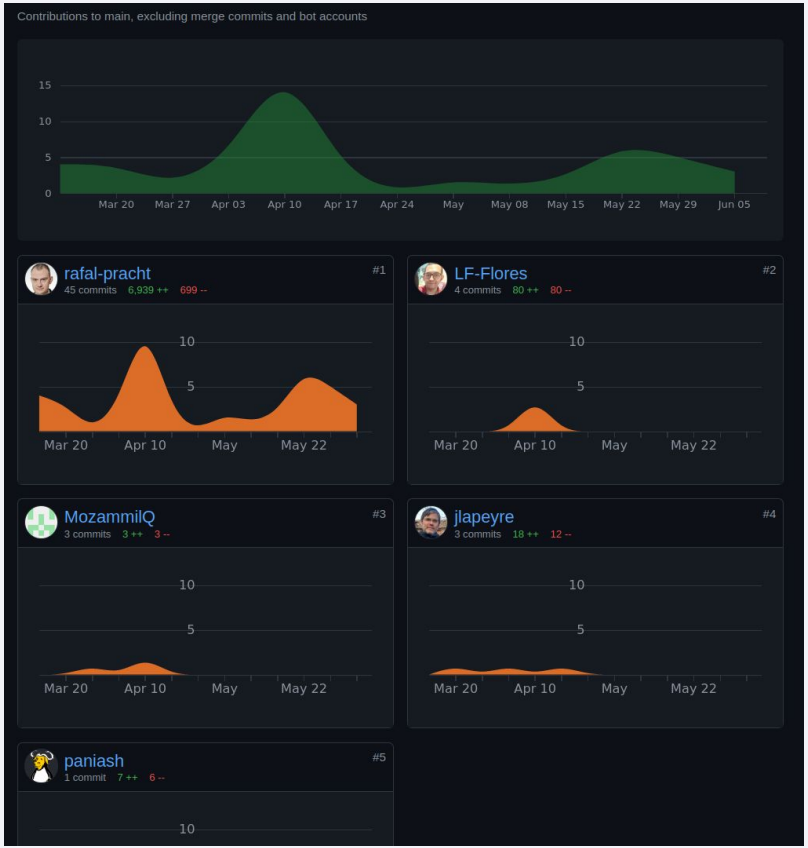
Documentation

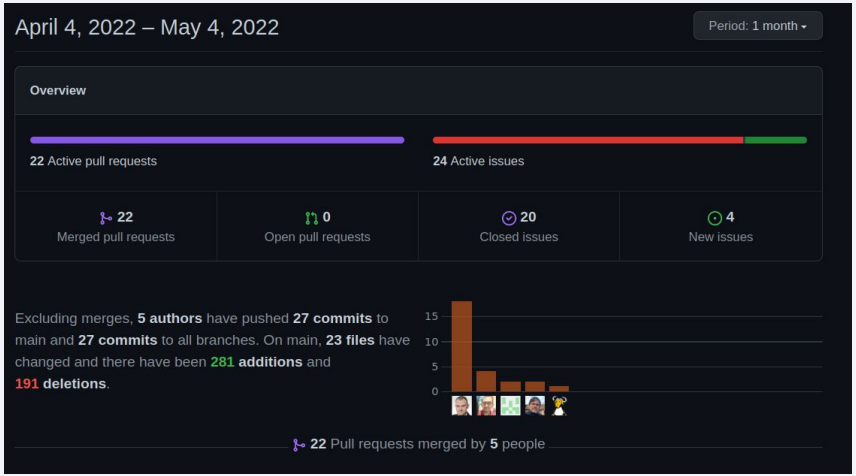
Qiskit

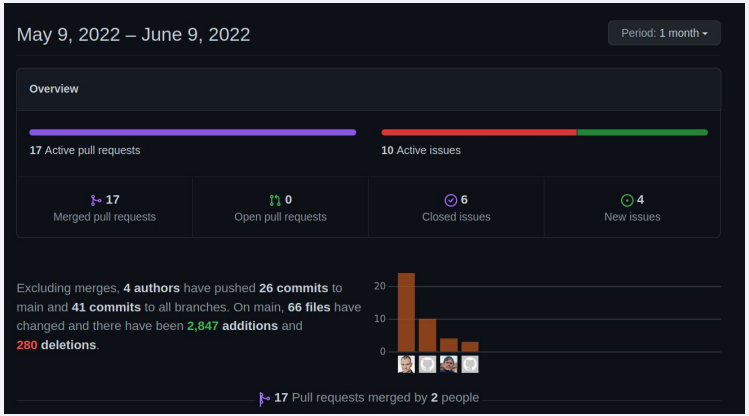


see: https://adgnitio.github.io/QuantumCircuits.jl/dev/

Collaboration





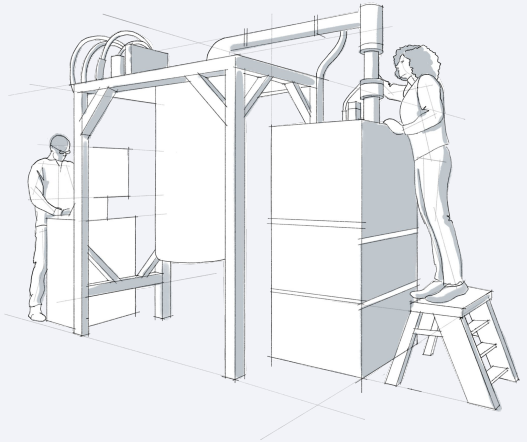


see: https://adgnitio.github.io/QuantumCircuits.jl/dev/

Future works

I would like to implement in the library the algorithm for derivative pricing using Quantum Monte Carlo.

Avoiding the "two languages" issue.



Fx Asian Option Pricing using Quantum Computers

Rafał Pracht[1], Adam Ryterski[1], Julia Plewa, and Marek Stefaniak[1]

BNP Paribas, ul. Kasprzaka 2; 01-211 Warszawa
{rafal.pracht,adam.ryterski,marek.stefaniak}@bnpparibas.pl

see: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4137397

Qiskit-Alt

Qiskit

Qiskit-Alt workflow

Qiskit

Molecular Simulations

Qiskit Nature Workflow

Defining the Molecular Geometry

Using qiskit-alt

Julia backend

Computing the Qubit Hamiltonian

Computing the Fermionic operator

Constructing the Fermionic Hamiltonian

Jordan-Wigner transformation

Convert to a Qubit Hamiltonian

Running molecular simulation algorithm

Computing the Qubit Hamiltonian: H_2 Molecule

Qiskit

Qiskit Nature

```
es_problem = ElectronicStructureProblem(driver)
second_q_op = es_problem.second_q_ops()
qubit_converter = QubitConverter(mapper=JordanWignerMapper())
qubit_op = qubit_converter.convert(second_q_op[0])
qubit_op.primitive
```

```
SparsePauliOp(['IIII', 'ZIII', 'IZII', 'ZZII', 'IIZI', 'ZIZI', 'IZZI', 'IIIZ', 'ZIIZ', 'IZIZ', 'IIZZ', 'XXXX', 'YYXX', 'XXYY', 'YYYY'],
              coeffs=[-0.81054798+0.j, -0.22575349+0.j,  0.17218393+0.j,  0.12091263+0.j,
 -0.22575349+0.j,  0.17464343+0.j,  0.16614543+0.j,  0.17218393+0.j,
  0.16614543+0.j,  0.16892754+0.j,  0.12091263+0.j,  0.0452328 +0.j,
  0.0452328 +0.j,  0.0452328 +0.j,  0.0452328 +0.j])
```

Qiskit Alt

```
# Compute the Fermionic operator of the molecule
fermi_op = qiskit_alt.electronic_structure.fermionic_hamiltonian(geometry, basis)

# Convert the Fermionic operator to a Pauli operator using the Jordan-Wigner transform
pauli_op = qiskit_alt.electronic_structure.jordan_wigner(fermi_op);

# Convert the Pauli operator into a sum of Pauli operators
pauli_sum_op = PauliSumOp(pauli_op)

# Print the PauliSumOp operator, which will be the input to the VQE algorithm to compute the minimum eigenvalue
print(pauli_sum_op)

# Print the SparsePauliOp operator ~ Fermionic operator computed with qiskit-alt
pauli_op.simplify()
```

```
-0.090578986088348 * IIII
- 0.22575349222402383 * ZIII
+ 0.17218393261915532 * IZII
+ 0.12091263261776633 * ZZII
- 0.22575349222402383 * IIZI
+ 0.17464343068300456 * ZIZI
+ 0.16614543256382416 * IZZI
+ 0.04523279994605783 * XXXX
+ 0.04523279994605783 * YYXX
+ 0.04523279994605783 * XXYY
+ 0.04523279994605783 * YYYY
+ 0.17218393261915553 * IIIZ
+ 0.16614543256382416 * ZIIZ
+ 0.16892753870087907 * IZIZ
+ 0.12091263261776633 * IIZZ
```

➡ **Similar generated Hamiltonian**

Running molecular simulation algorithm

Qiskit

Computing the ground state energy

- Ansatz: TwoLocal
- Optimizer: COBYLA
- Backend: Statevector Simulator

Results

Qiskit Nature

```
# Compute the ground-state energy of the molecule
nature_result = vqe.compute_minimum_eigenvalue(operator=qubit_op)
nature_energy = nature_result.eigenvalue.real + nuclear_energy_term # Include constant term that is normally thrown away
print("The ground-state energy of the Hydrogen molecule is {} Hatree".format(round(nature_energy,3)))
```

```
The ground-state energy of the Hydrogen molecule is -1.117 Hatree
```

Qiskit Alt

```
# Convert the Pauli operator into a sum of Pauli operators, the required input format
pauli_sum_op = PauliSumOp(pauli_op)
```

```
# Compute the ground-state energy of the molecule
alt_result = vqe.compute_minimum_eigenvalue(operator=pauli_sum_op)
print("The ground-state energy of the Hydrogen molecule is {} Hatree".format(round(alt_result.eigenvalue.real,3)))
```

```
The ground-state energy of the Hydrogen molecule is -1.117 Hatree
```

✓ Time advantage

✓ Results accuracy

Project plan

Objective: Integrate qiskit-alt in qiskit nature workflow

- ✓ Generate Hamiltonian operators with qiskit-alt
- ✓ VQE tutorial

- ✓ Features of Qiskit Nature
- ✓ Testing molecular simulation algorithms
- ✓ Testing performance

- ✓ Release a plugin to use qiksit-alt in qiskit nature workflow

Thank you for your attention

Qiskit