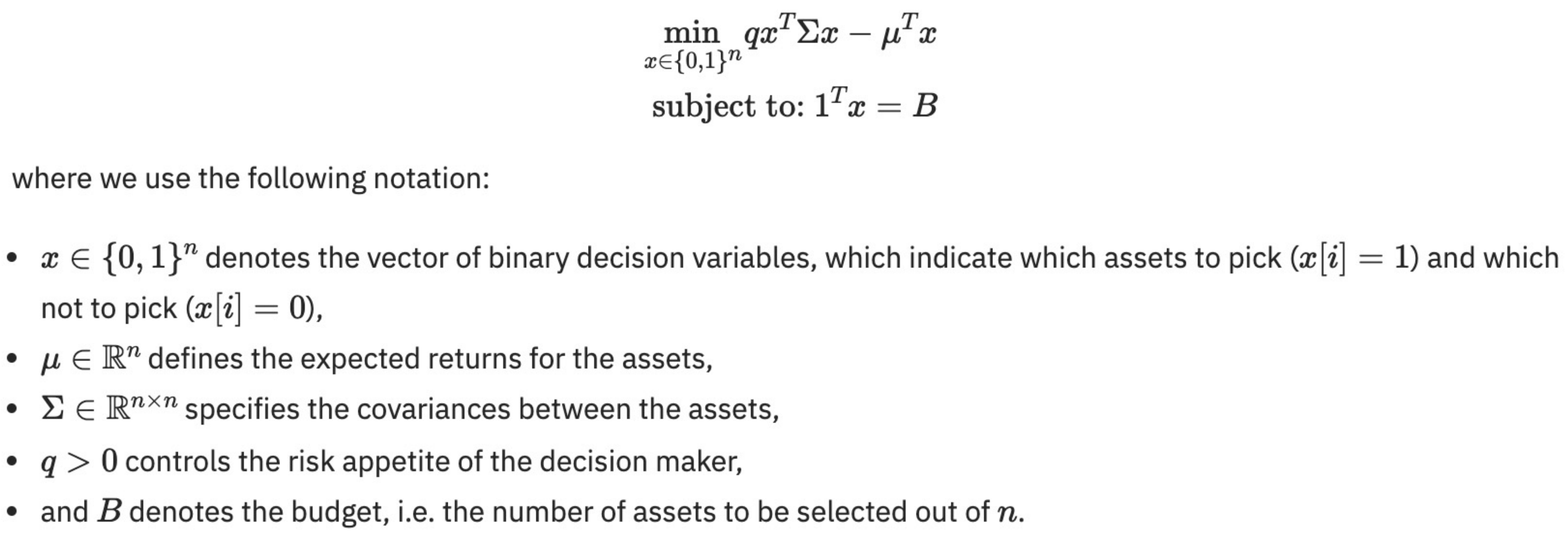
Portfolio Optimization Based on QAOA

QHack Open Hackathon Project

cyx

1. Object of Project

We aim to solve the portfolio optimization problem which is one of the most popular topics in the area of finance. The problem can be formulated as a combinatorial optimization problem subject to certain constraints provided below.



$$\min_{x\in\{0,1\}^n} qx^T\Sigma x - \mu^T x$$

$$\text{subject to: } 1^T x = B$$

where we use the following notation:

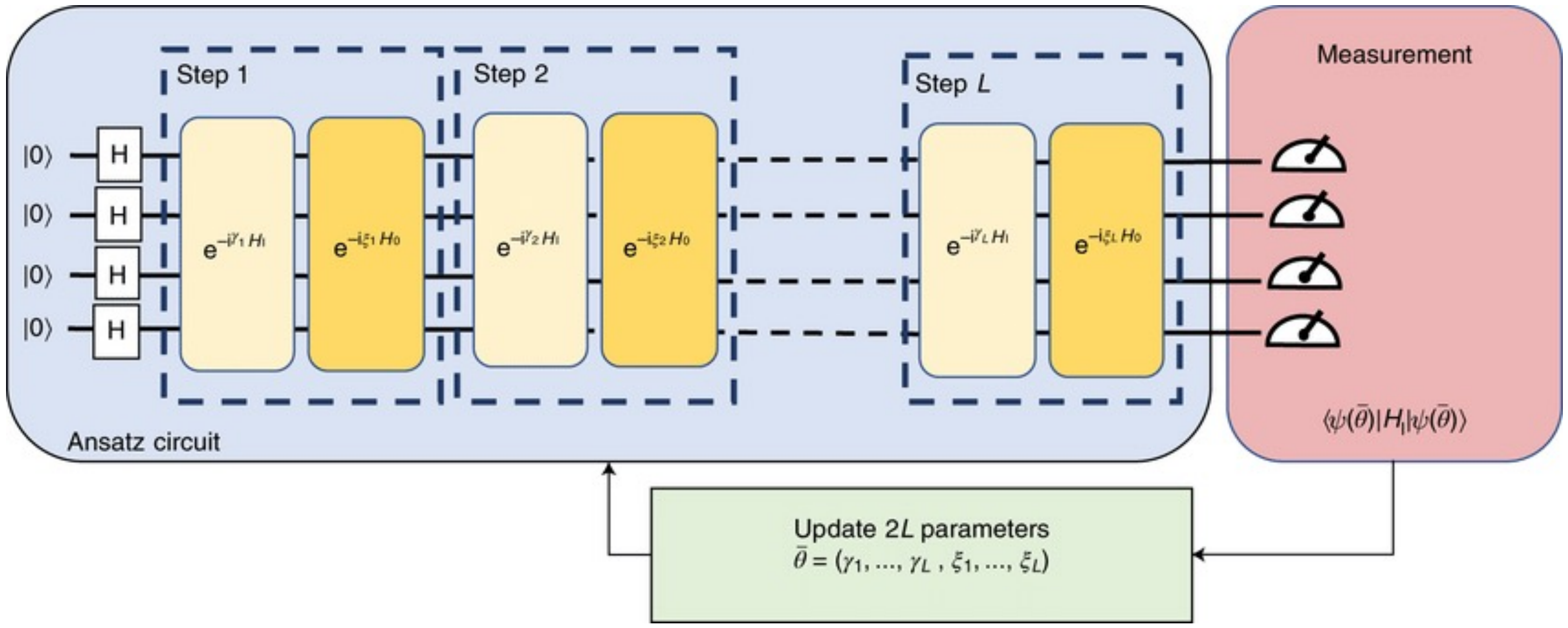
- $x \in \{0,1\}^n$ denotes the vector of binary decision variables, which indicate which assets to pick ($x[i] = 1$) and which not to pick ($x[i] = 0$),
- $\mu \in \mathbb{R}^n$ defines the expected returns for the assets,
- $\Sigma \in \mathbb{R}^{n\times n}$ specifies the covariances between the assets,
- $q > 0$ controls the risk appetite of the decision maker,
- and B denotes the budget, i.e. the number of assets to be selected out of n.

By treating the constraint as a penalty term in the cost function, this problem can be reformulated as a quadratic unconstrained binary optimization (QUBO) problem.

2. Method

- We have exact optimization methods (e.g. Minimum Eigen solver) for solving this type of QUBO problems. But the running times of these method usually grow exponentially with the problem size and thus it would be very expensive and impractical to implement them when dealing with real-world portfolio optimization problems involving a large number of assets.

- In this project, we employ the well-known quantum approximate optimization algorithm (QAOA) to solve portfolio optimization problems and attempt to find the quantum advantage over classical methods. The structure of QAOA is shown below.



3. Experiments

We perform three experiments for this project. In the first two experiments, we implement the QAOA algorithm with Qiskit and PennyLane respectively. In the last experiment, we select the NumPyMinimumEigensolver as the benchmark method to compare with the QAOA. To make a fair comparison, we use the same server environment and data for all experiments. We provide detailed information of our experiments below.

Experiment	Framework	Optimizer	Backend	Circuit Depth	Max_Iteration	Num_assets	Budget	Penalty
A	Qiskit	QAOA with COBYLA	qasm_simulator (CPU) aer_simulator_statevector (GPU)	3	1000	2-28	num_assets /2	num_assets
B	PennyLane	QAOA with ADAM	default.qubit	3	1000	2-17	num_assets /2	num_assets
C	Numpy	NumPyMinimumEigensolver	None	None	None	2-28	num_assets /2	num_assets

Note:

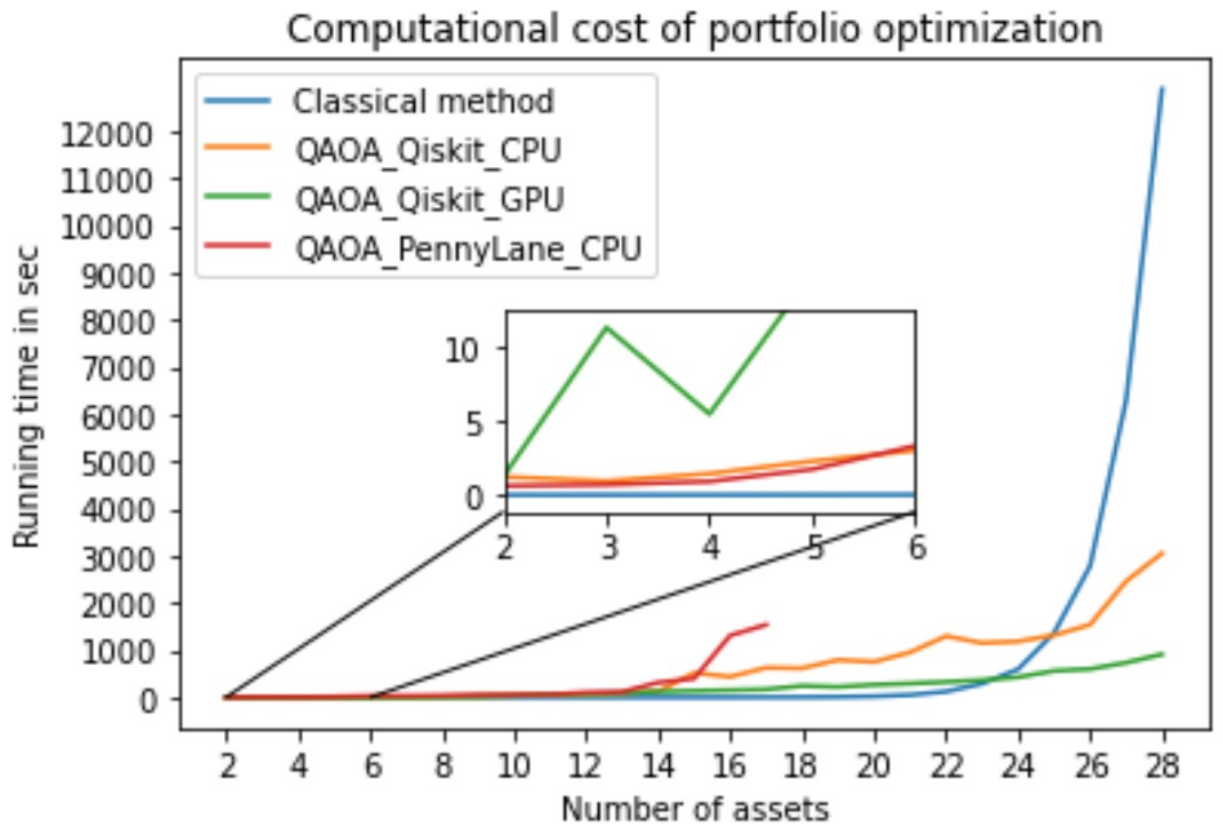
- In Experiment A, we attempted to use Adam optimizer to optimize parameters of QAOA. But the program got stuck when the number of assets is 22. So we used Cobyla optimizer instead for this experiment.

- In Experiment A, we initially used 'aer_simulator_statevector' for CPU simulation. But using it become very expensive when the number of assets is bigger than 16. So we replaced it with 'qasm_simulator' which supports Aer's built-in fast Pauli Expectation.

- In Experiment B, we used backends 'lightening.qubit' , 'qiskit.aer' (CPU and GPU), and 'qulacs.simulator' (CPU and GPU) to implement QAOA. But all of them led to much longer running times even for small problem size (e.g. num_assets=2), compared to the 'default.qubit' simulator. So we finally selected 'default.qubit' as the simulator for this experiment.

- In Experiment B, even with the 'default.qubit' simulator, we got extremely long running time when the number of assets become bigger than 17. Thus, we only ran the QAOA program for problems with num_assets ranging from 2 to 17.

4. First-phase Outcomes



Method	Backend	Accuracy*
Classical method	None	100%
QAOA_Qiskit_CPU	qasm_simulator	62.5%
QAOA_Qiskit_GPU	aer_simulator_statevector	62.5%
QAOA_PennyLane_CPU	default.qubit	93.75%

*The accuracy in the table is based on problems with number of assets ranging from 2 to 17