Quantum Walk Mixer Based QAOA for Portfolio Optimization

Quantathon 2.0

Team: FSociety

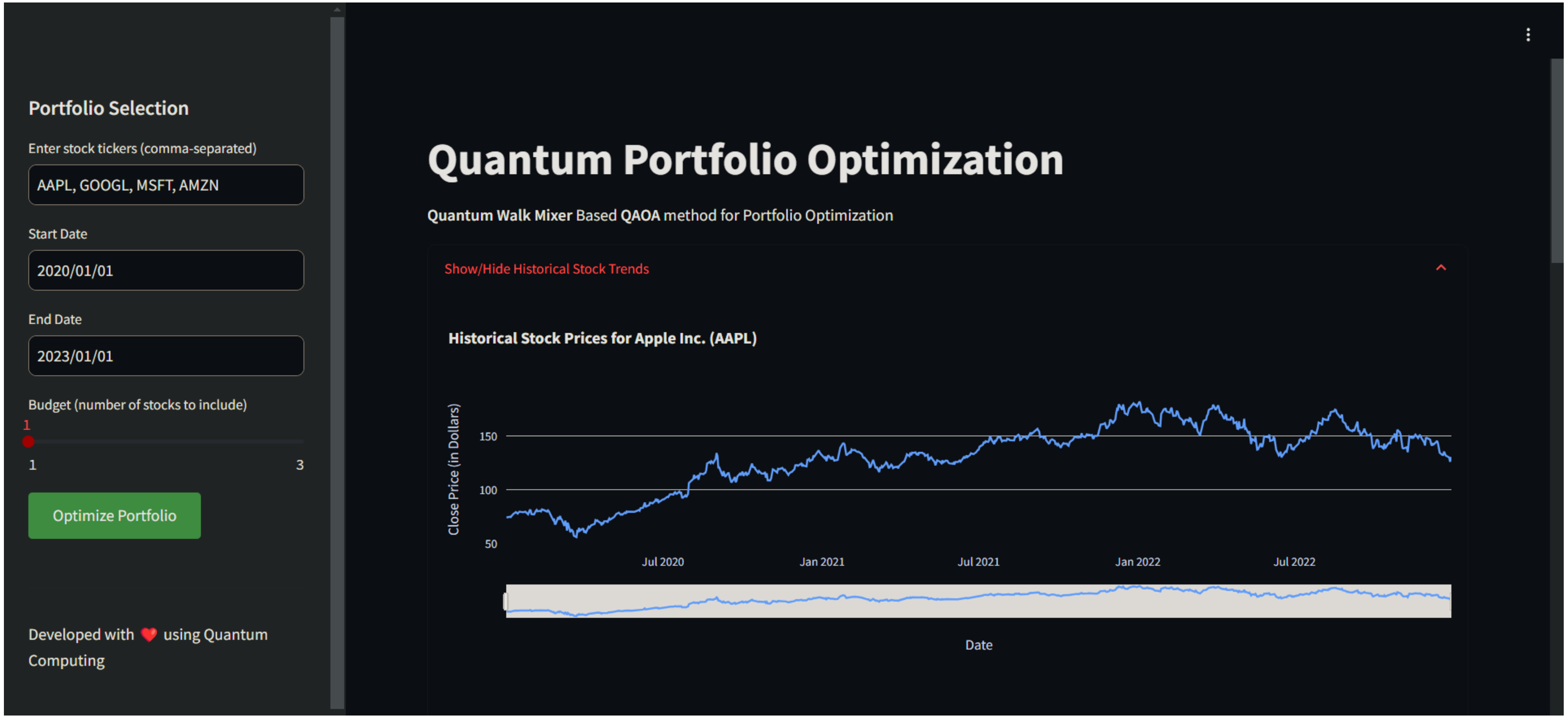
Adhithyan VP, Jyotiraj Nath

The Problem Statement

“Portfolio Optimization Based on Quantum Walks”

(as given by the SQCC Hacktathon 2.0)

The Interface (switching to the deployed site)



Portfolio Selection

Enter stock tickers (comma-separated)

AAPL, GOOGL, MSFT, AMZN

Start Date

2020/01/01

End Date

2023/01/01

Budget (number of stocks to include)

1

1 3

Optimize Portfolio

Developed with ❤️ using Quantum Computing

Quantum Portfolio Optimization

Quantum Walk Mixer Based **QAOA** method for Portfolio Optimization

Show/Hide Historical Stock Trends

Historical Stock Prices for Apple Inc. (AAPL)

Experimenting with 4 stocks and 2 budget stocks

Chosen Stocks :

APPL (Apple)

GOOGL (Google)

MSFT (Microsoft)

AMZN (AMAZON)

Experimenting with 4 stocks and 2 budget stocks

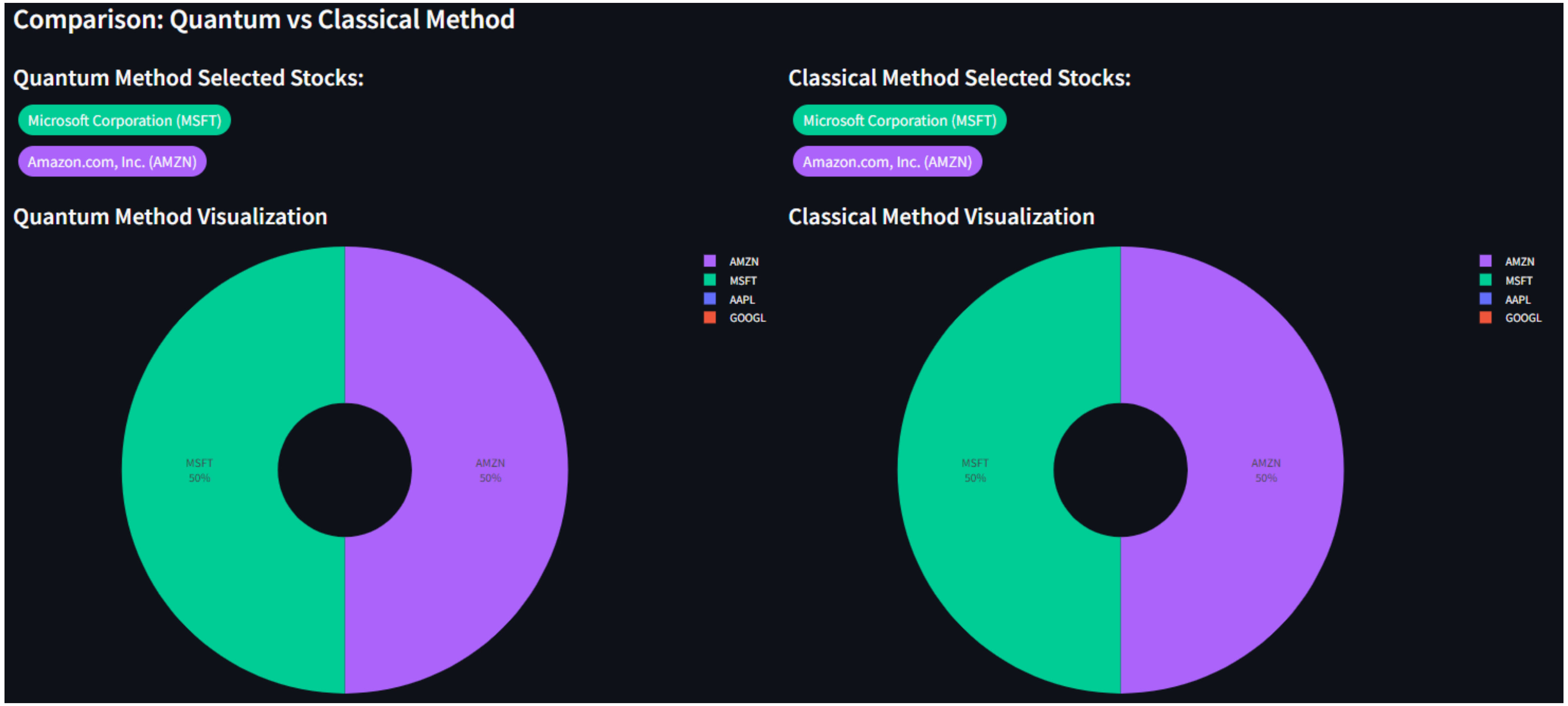
Chosen Stocks :

APPL (Apple)

GOOGL (Google)

MSFT (Microsoft)

AMZN (AMAZON)



Comparison: Quantum vs Classical Method

Quantum Method Selected Stocks:

Microsoft Corporation (MSFT)

Amazon.com, Inc. (AMZN)

Quantum Method Visualization

Classical Method Selected Stocks:

Microsoft Corporation (MSFT)

Amazon.com, Inc. (AMZN)

Classical Method Visualization

Experimenting with 4 stocks and 2 budget stocks

Chosen Stocks :

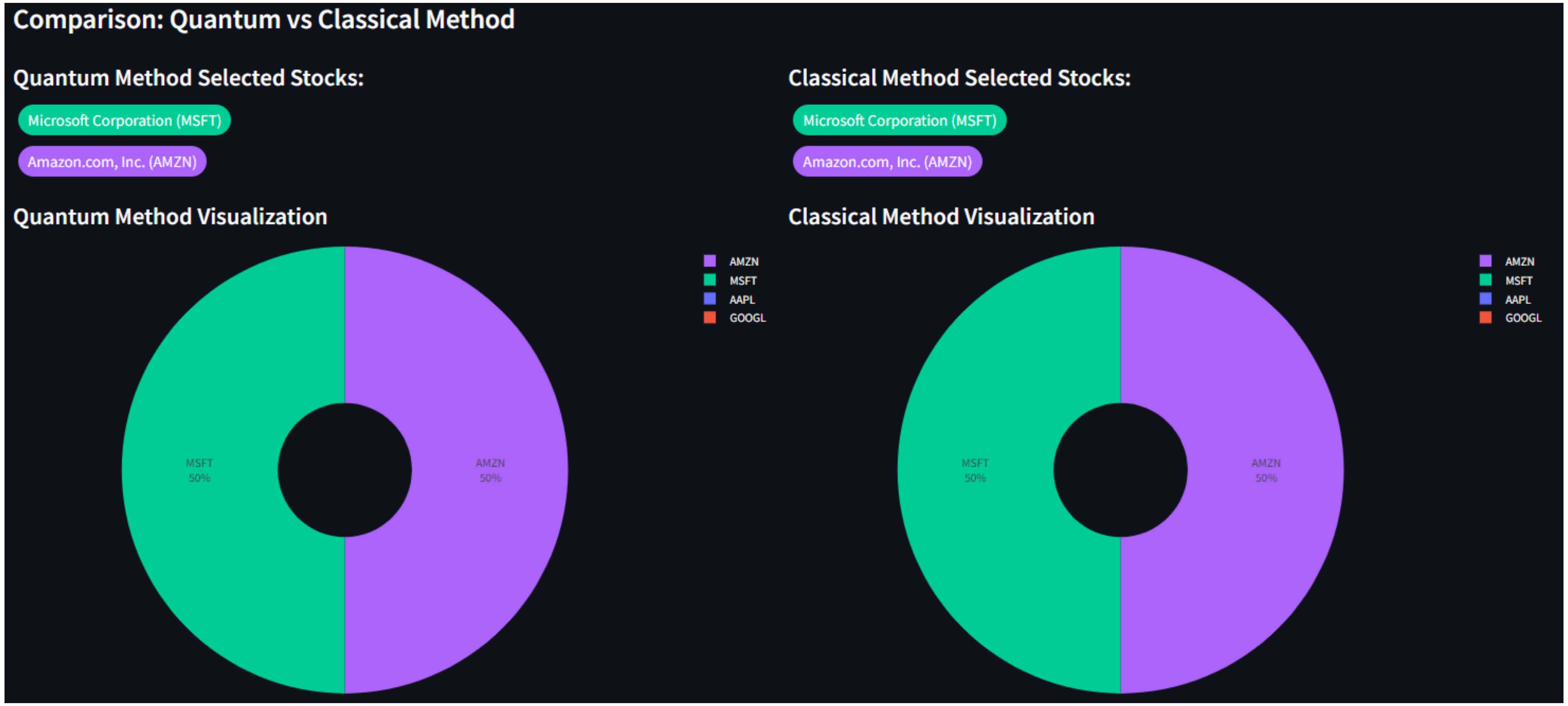
APPL (Apple)

GOOGL (Google)

MSFT (Microsoft)

AMZN (AMAZON)

"Nothing to worry About as both Quantum And Classical Results Are same"



Comparison: Quantum vs Classical Method

Quantum Method Selected Stocks:

Microsoft Corporation (MSFT)

Amazon.com, Inc. (AMZN)

Quantum Method Visualization

Classical Method Selected Stocks:

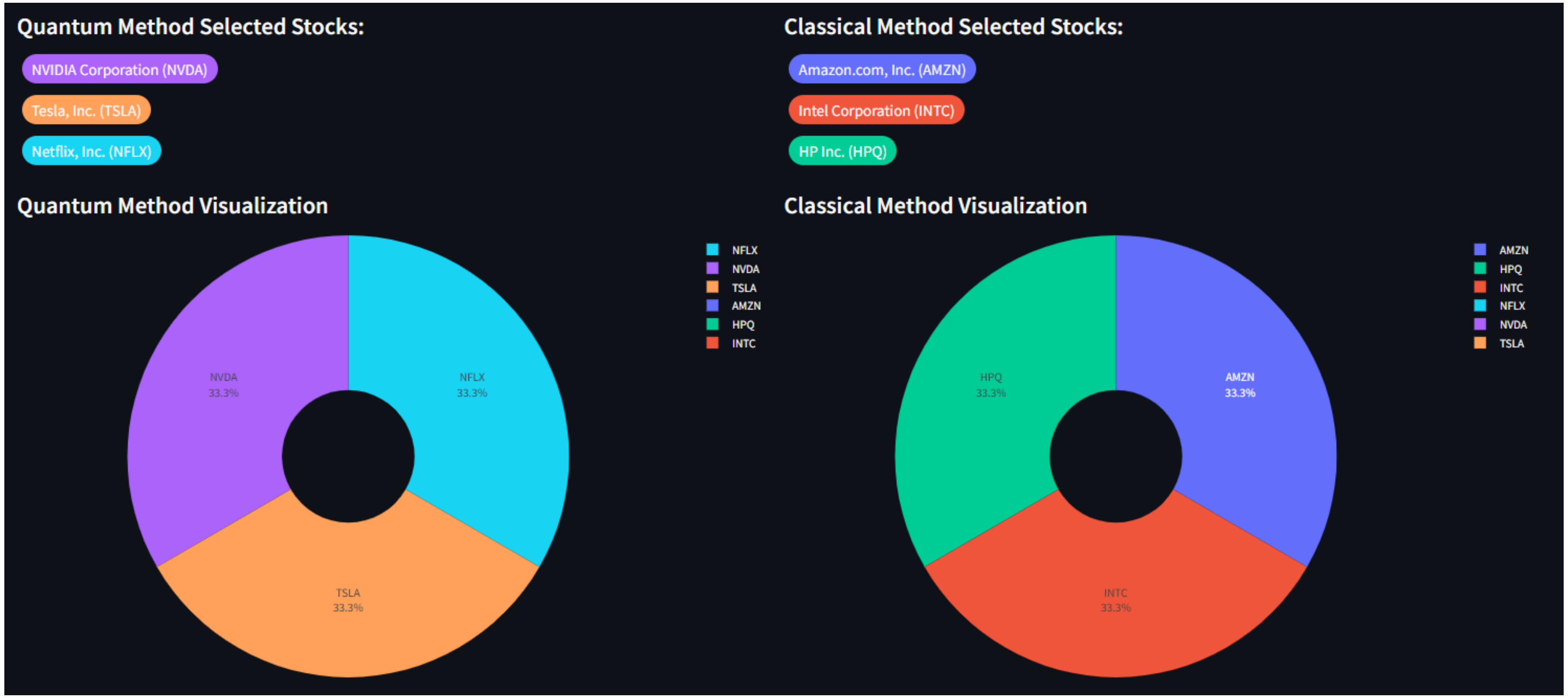
Microsoft Corporation (MSFT)

Amazon.com, Inc. (AMZN)

Classical Method Visualization

Experimenting with 6 stocks and 3 budget stocks

Chosen Stocks : AMZN, INTC, HPQ, NVDA, TSLA, NFLX



Quantum Method Selected Stocks:

- NVIDIA Corporation (NVDA)
- Tesla, Inc. (TSLA)
- Netflix, Inc. (NFLX)

Quantum Method Visualization

Classical Method Selected Stocks:

- Amazon.com, Inc. (AMZN)
- Intel Corporation (INTC)
- HP Inc. (HPQ)

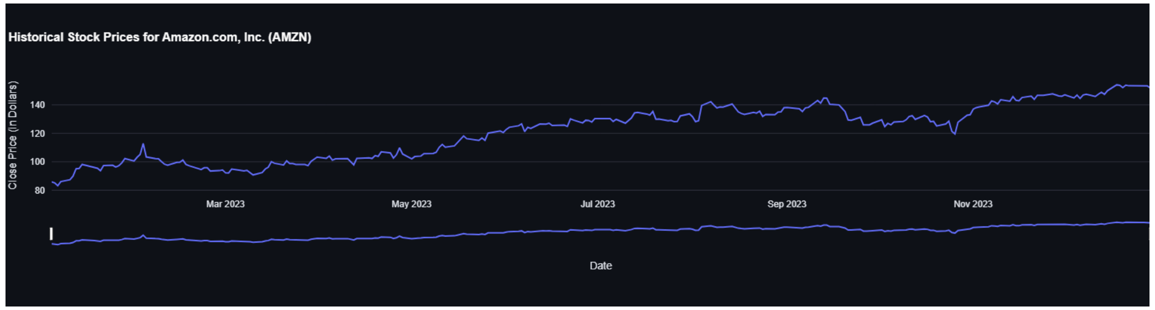
Classical Method Visualization

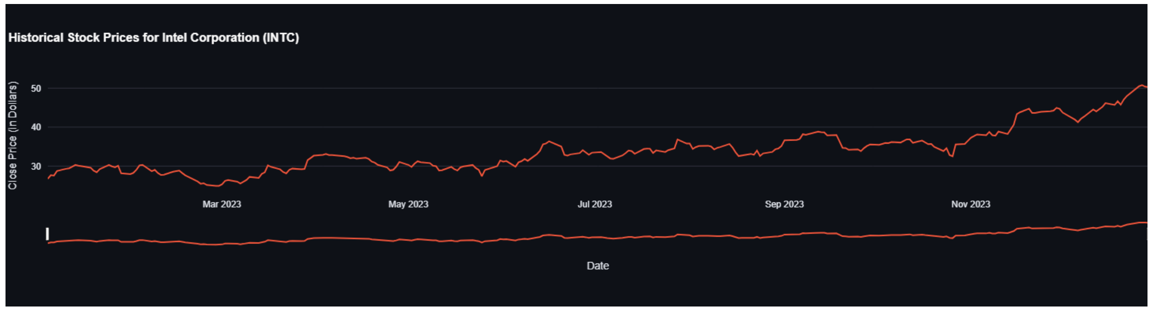
Approximation Ratio ~ 31%

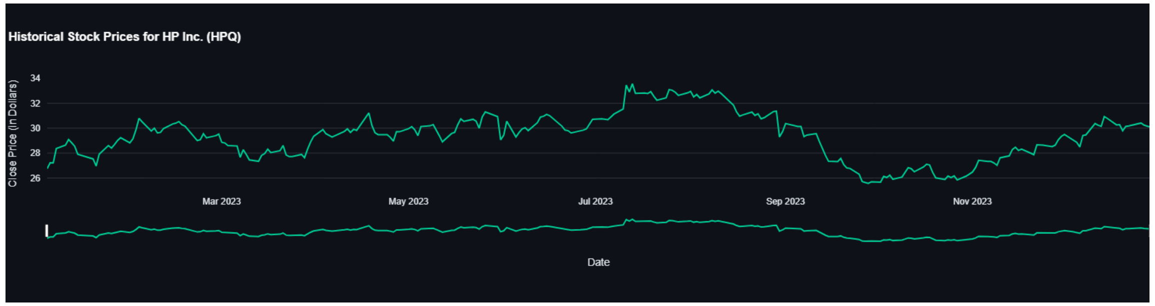
Is it really that bad and our method is completely offset from the classical results?

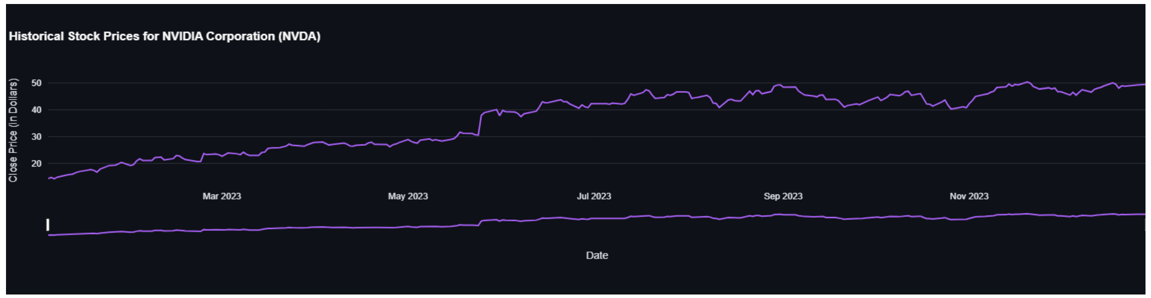
We go beyond the results

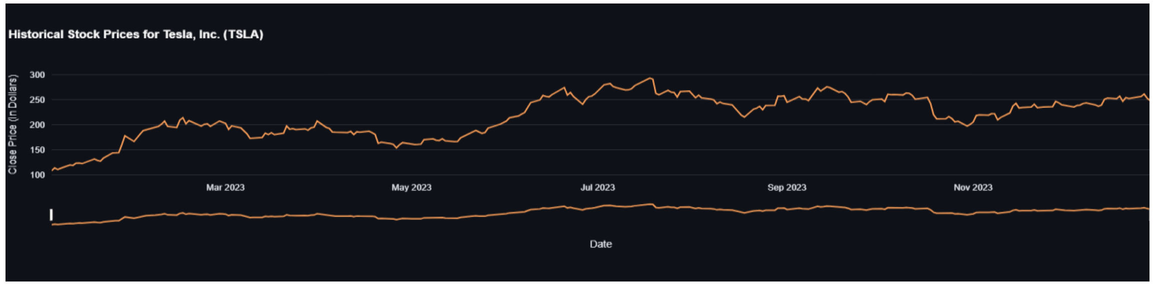
Chosen Stocks : AMZN, INTC, HPQ, NVDA, TSLA, NFLX

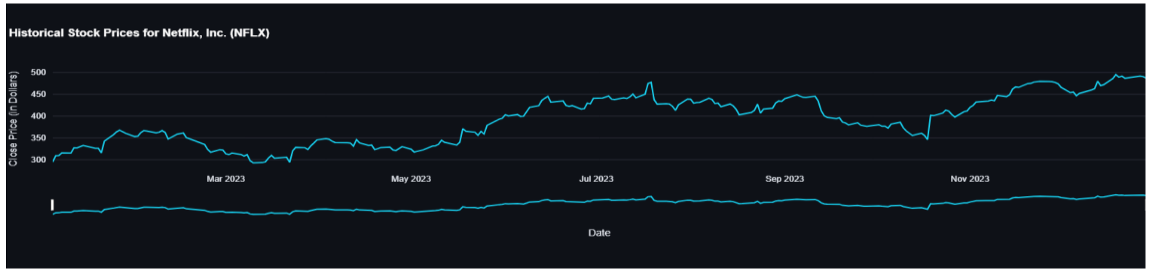












We go beyond the results

Chosen Stocks : AMZN, INTC, HPQ, NVDA, TSLA, NFLX

Quantum Method said:

TSLA:	101%
NFLX:	59%
NVDA:	239%

Classical Method said:

AMZN:	50%
INTC:	25%
HPQ:	12%

We go beyond the results

Chosen Stocks : AMZN, INTC, HPQ, NVDA, TSLA, NFLX

Quantum Method said:

TSLA: 101%

NFLX: 59%

NVDA: 239%

Classical Method said:

AMZN: 50%

INTC: 25%

HPQ: 12%

If we invested in $100 in each Company in that time period

Quantum Method would've become:
$699 from the $300 investment

Classical Method would've become:
$387 from the $300 investment

We go beyond the results

Chosen Stocks : AMZN, INTC, HPQ, NVDA, TSLA, NFLX

Quantum Method said:

TSLA: 101%

NFLX: 59%

NVDA: 239%

Classical Method said:

AMZN: 50%

INTC: 25%

HPQ: 12%

If we invested in $100 in each Company in that time period

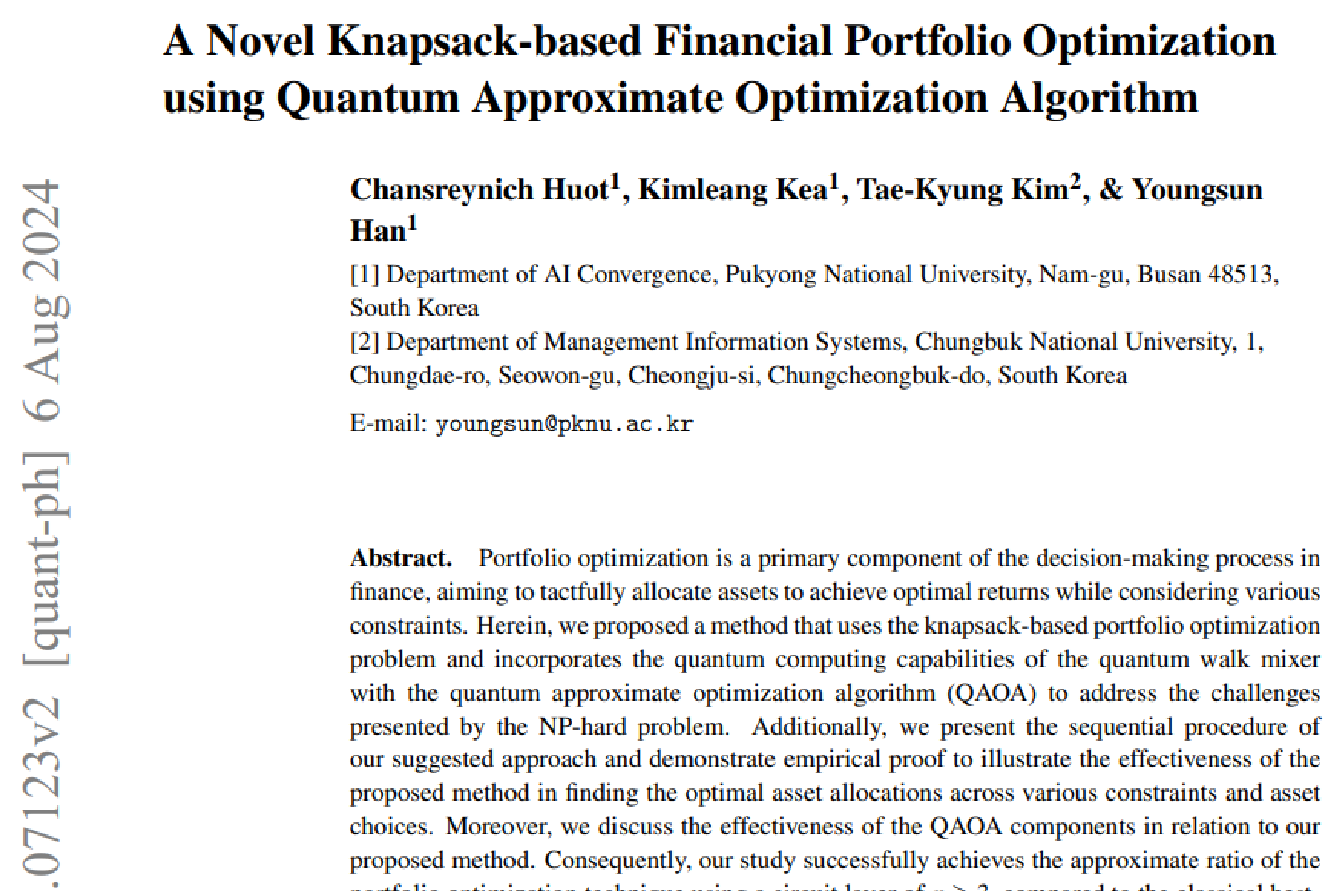
Quantum Method would've become:
$699 from the $300 investment

Classical Method would've become:
$387 from the $300 investment

Quantum Method: 133% return
Classical Method: 29% return

What is going on behind the hood

Credit is due where it is due: https://arxiv.org/pdf/2402.07123



A Novel Knapsack-based Financial Portfolio Optimization using Quantum Approximate Optimization Algorithm

Chansreynich Huot[1], **Kimleang Kea**[1], **Tae-Kyung Kim**[2], **& Youngsun Han**[1]

[1] Department of AI Convergence, Pukyong National University, Nam-gu, Busan 48513, South Korea

[2] Department of Management Information Systems, Chungbuk National University, 1, Chungdae-ro, Seowon-gu, Cheongju-si, Chungcheongbuk-do, South Korea

E-mail: `youngsun@pknu.ac.kr`

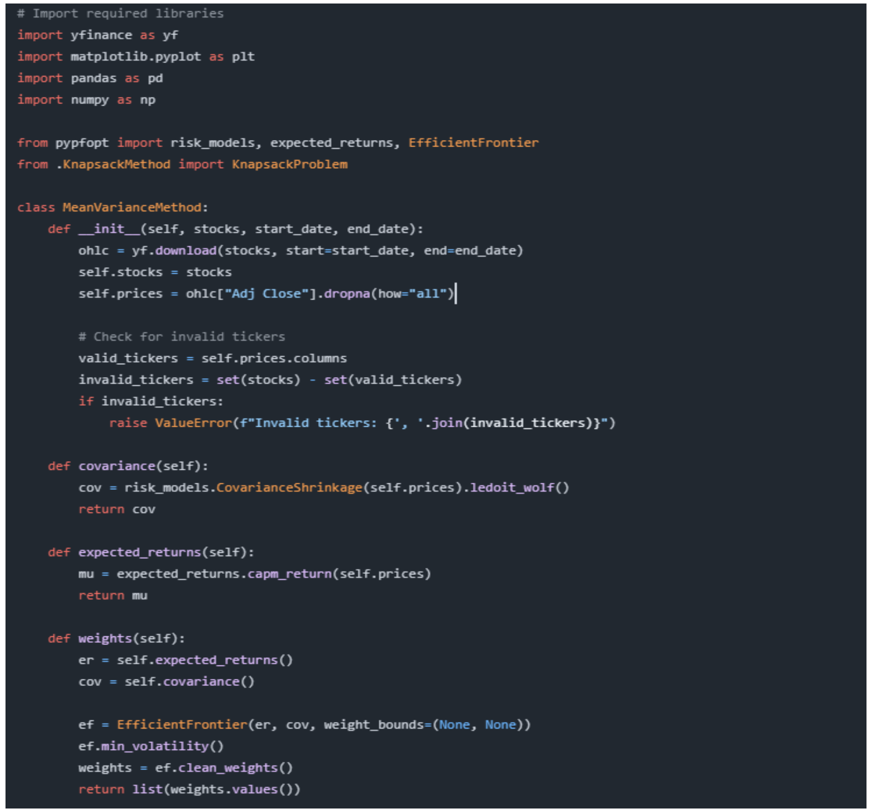
Abstract. Portfolio optimization is a primary component of the decision-making process in finance, aiming to tactfully allocate assets to achieve optimal returns while considering various constraints. Herein, we proposed a method that uses the knapsack-based portfolio optimization problem and incorporates the quantum computing capabilities of the quantum walk mixer with the quantum approximate optimization algorithm (QAOA) to address the challenges presented by the NP-hard problem. Additionally, we present the sequential procedure of our suggested approach and demonstrate empirical proof to illustrate the effectiveness of the proposed method in finding the optimal asset allocations across various constraints and asset choices. Moreover, we discuss the effectiveness of the QAOA components in relation to our proposed method. Consequently, our study successfully achieves the approximate ratio of the

The Historical Returns and the Knapsack Method

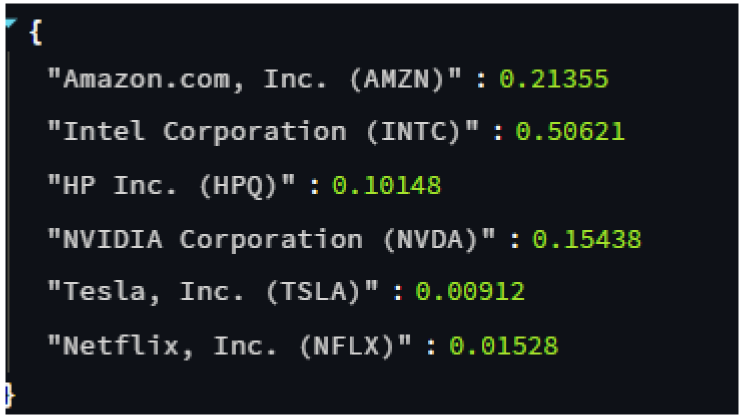
We use the Historical return based on the Markowitz Model of Mean Variance

“Maximum expected yield by allocating funds into stocks considering minimization risk”

(A classical Method)



```python
# Import required libraries
import yfinance as yf
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from pypfopt import risk_models, expected_returns, EfficientFrontier
from .KnapsackMethod import KnapsackProblem

class MeanVarianceMethod:
    def __init__(self, stocks, start_date, end_date):
        ohlc = yf.download(stocks, start=start_date, end=end_date)
        self.stocks = stocks
        self.prices = ohlc["Adj Close"].dropna(how="all")

        # Check for invalid tickers
        valid_tickers = self.prices.columns
        invalid_tickers = set(stocks) - set(valid_tickers)
        if invalid_tickers:
            raise ValueError(f"Invalid tickers: {', '.join(invalid_tickers)}")

    def covariance(self):
        cov = risk_models.CovarianceShrinkage(self.prices).ledoit_wolf()
        return cov

    def expected_returns(self):
        mu = expected_returns.capm_return(self.prices)
        return mu

    def weights(self):
        er = self.expected_returns()
        cov = self.covariance()

        ef = EfficientFrontier(er, cov, weight_bounds=(None, None))
        ef.min_volatility()
        weights = ef.clean_weights()
        return list(weights.values())
```



```
{
  "Amazon.com, Inc. (AMZN)" : 0.21355
  "Intel Corporation (INTC)" : 0.50621
  "HP Inc. (HPQ)" : 0.10148
  "NVIDIA Corporation (NVDA)" : 0.15438
  "Tesla, Inc. (TSLA)" : 0.00912
  "Netflix, Inc. (NFLX)" : 0.01528
}
```

The Historical Returns and the Knapsack Method

We got the expected return E(Ri) for ith stock

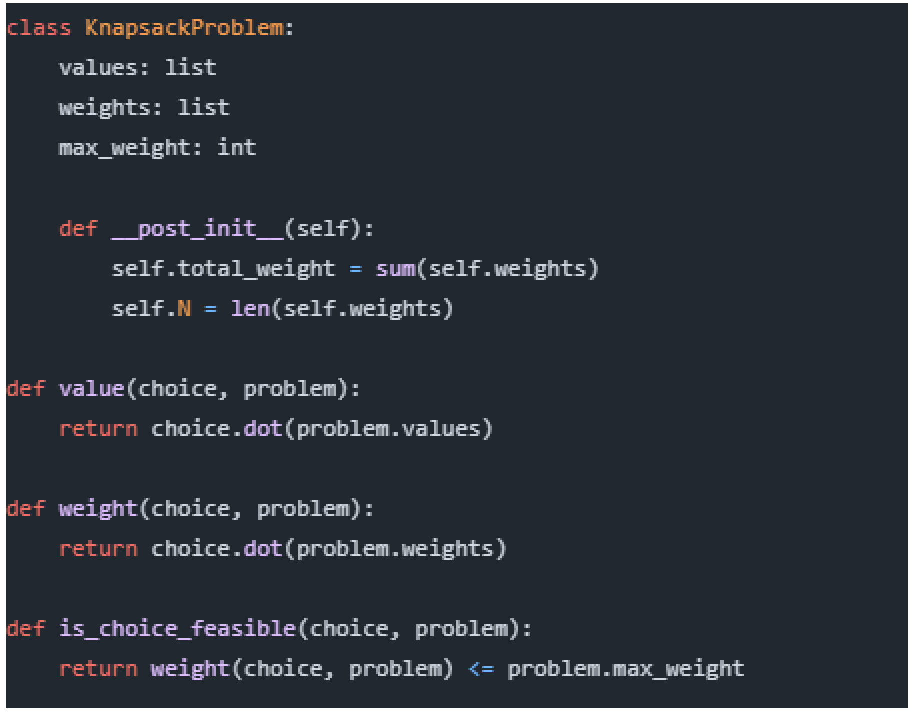
Which now need to be packed into a Knapsack

$$\text{maximize} \quad R(x) = \sum_{i=1}^{n} x_i E(R_i)$$

$$\text{subject to} \quad \sum_{i=1}^{n} x_i w_i \leq C,$$

xi- is a binary string which says, whether to include the stock and wi is the risk associated with it

So, the Knapsack problem becomes: Knapsack(E(Ri), wi, C)



```
class KnapsackProblem:
    values: list
    weights: list
    max_weight: int

    def __post_init__(self):
        self.total_weight = sum(self.weights)
        self.N = len(self.weights)

def value(choice, problem):
    return choice.dot(problem.values)

def weight(choice, problem):
    return choice.dot(problem.weights)

def is_choice_feasible(choice, problem):
    return weight(choice, problem) <= problem.max_weight
```

The Feasibility Oracle(Uf)

To check whether a proposed solution to the knapsack problem violates any constraints

We defined the Knapsack as K(N) = $(0,1)^N$ set of all possible bitstring of length N for portfolio choices

x included in the K(N). We create a subset of feasible Solution F

$$f : K(N) \to \{0,1\}, \quad x \mapsto f(x) = \begin{cases} 1, & \text{if } x \in F, \\ 0, & \text{otherwise.} \end{cases}$$

$$U_f \left|x,y\right\rangle = \left|x, y \oplus w(x)\right\rangle, \quad \forall x \in K(N), \quad y \in K(1)$$

y is a flag qubit based on the feasibility of the state of x

And we have all the x included in F iff total weight w(x)

We would need some ancillary qubits for

- Formulated Knapsack choices (S)
- Weights of the Item Choices K_w (All the item weights addition w, controlled by corresponding bit in register S)
- Flag qubit indicating feasibility of state K_F

The first Oracle U$_1$

Augments K$_w$ with all the weights based on the Knapsack Choices of S (Facilitates the binary representation)

- First we initialize the weight register K$_w$ 0> state
- To represent the weight in a distributed amplitude we apply the QFT

$$QFT\,|K_w\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i \cdot 0 \cdot k/2^n} |k\rangle$$

- Now, we add the weights conditioned on the bitstring x representing the selection of choices with Cphase Operations

$$|K_w\rangle = \begin{cases} |K_w + w_i\rangle\,, & \text{if } x_i = 1, \\ |K_w\rangle\,, & \text{otherwise.} \end{cases}$$

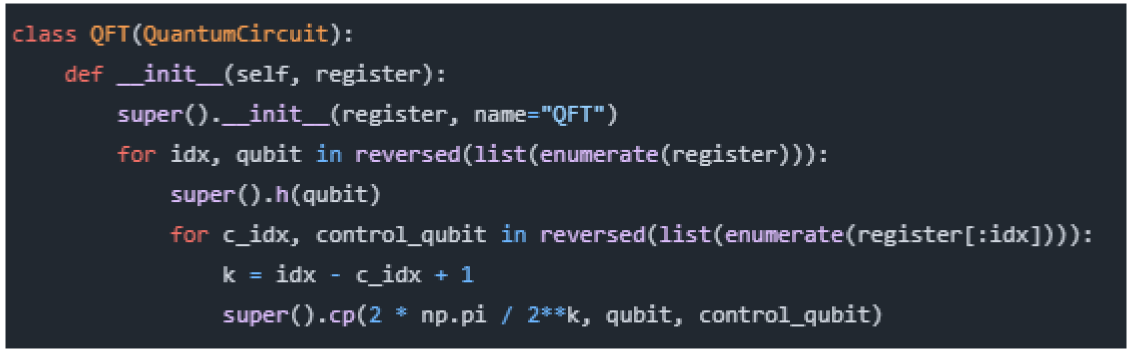
- Now to change back the phase information to a computational basis using an uncompute step QFT^{-1}

$$QFT^{-1}\,|K_w\rangle$$

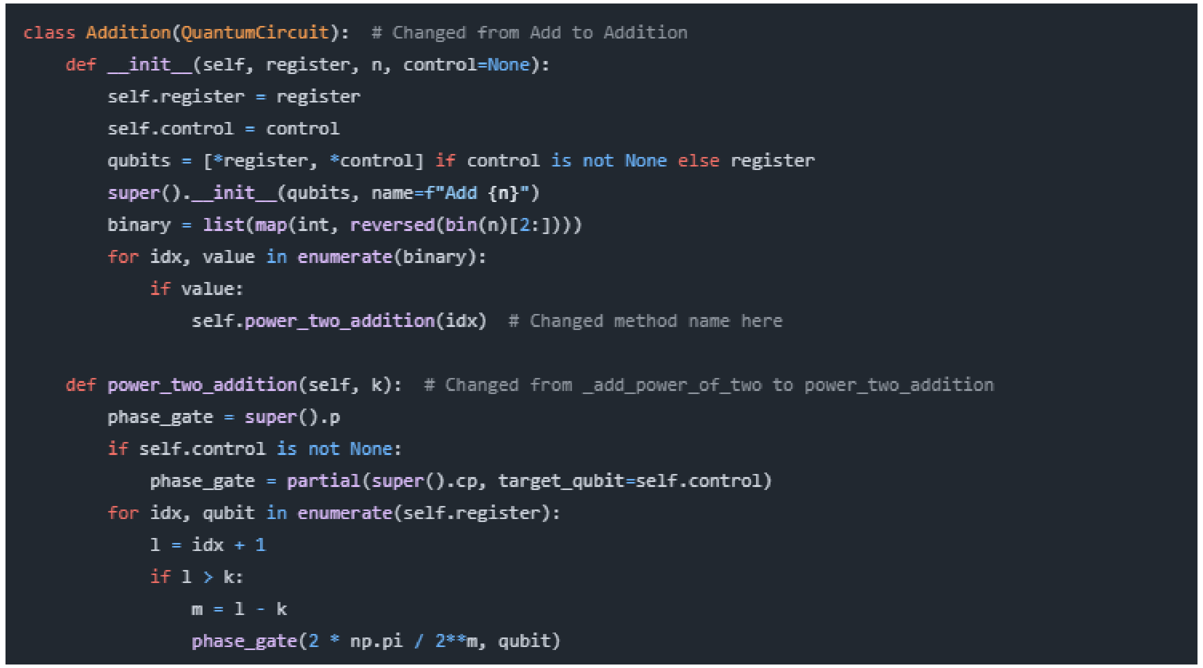
- Now it is a binary number of the sum of the weights

The first Oracle U_1

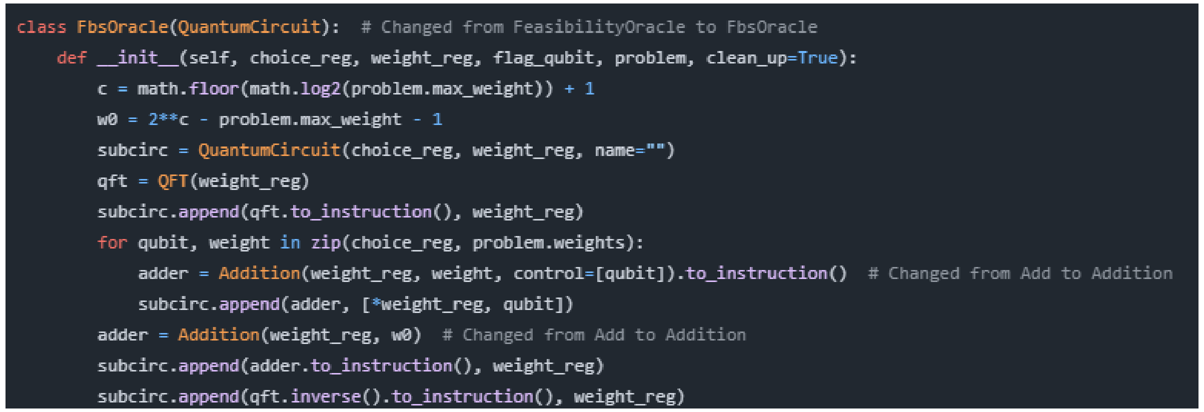
Circuits.py file



```python
class QFT(QuantumCircuit):
    def __init__(self, register):
        super().__init__(register, name="QFT")
        for idx, qubit in reversed(list(enumerate(register))):
            super().h(qubit)
            for c_idx, control_qubit in reversed(list(enumerate(register[:idx]))):
                k = idx - c_idx + 1
                super().cp(2 * np.pi / 2**k, qubit, control_qubit)
```



```python
class Addition(QuantumCircuit):  # Changed from Add to Addition
    def __init__(self, register, n, control=None):
        self.register = register
        self.control = control
        qubits = [*register, *control] if control is not None else register
        super().__init__(qubits, name=f"Add {n}")
        binary = list(map(int, reversed(bin(n)[2:])))
        for idx, value in enumerate(binary):
            if value:
                self.power_two_addition(idx)  # Changed method name here

    def power_two_addition(self, k):  # Changed from _add_power_of_two to power_two_addition
        phase_gate = super().p
        if self.control is not None:
            phase_gate = partial(super().cp, target_qubit=self.control)
        for idx, qubit in enumerate(self.register):
            l = idx + 1
            if l > k:
                m = l - k
                phase_gate(2 * np.pi / 2**m, qubit)
```



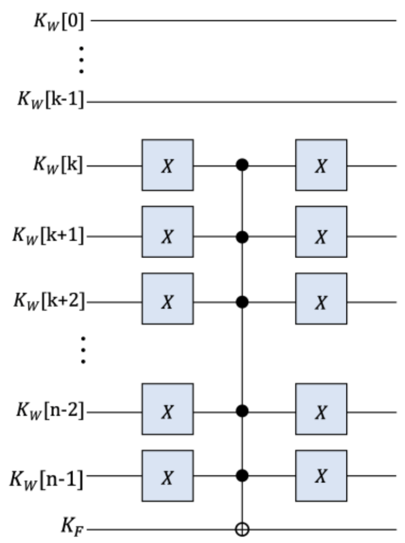
```python
class FbsOracle(QuantumCircuit):  # Changed from FeasibilityOracle to FbsOracle
    def __init__(self, choice_reg, weight_reg, flag_qubit, problem, clean_up=True):
        c = math.floor(math.log2(problem.max_weight)) + 1
        w0 = 2**c - problem.max_weight - 1
        subcirc = QuantumCircuit(choice_reg, weight_reg, name="")
        qft = QFT(weight_reg)
        subcirc.append(qft.to_instruction(), weight_reg)
        for qubit, weight in zip(choice_reg, problem.weights):
            adder = Addition(weight_reg, weight, control=[qubit]).to_instruction()  # Changed from Add to Addition
            subcirc.append(adder, [*weight_reg, qubit])
        adder = Addition(weight_reg, w0)  # Changed from Add to Addition
        subcirc.append(adder.to_instruction(), weight_reg)
        subcirc.append(qft.inverse().to_instruction(), weight_reg)
```

The second Oracle U_2

Here, we need to satisfy the condition $w(x) \leq C \Leftrightarrow w(x) + C_0 < C + C_0 + 1$

We will conditionally modify the state of an ancillary qubit concerning the sum of weights in binary K_w

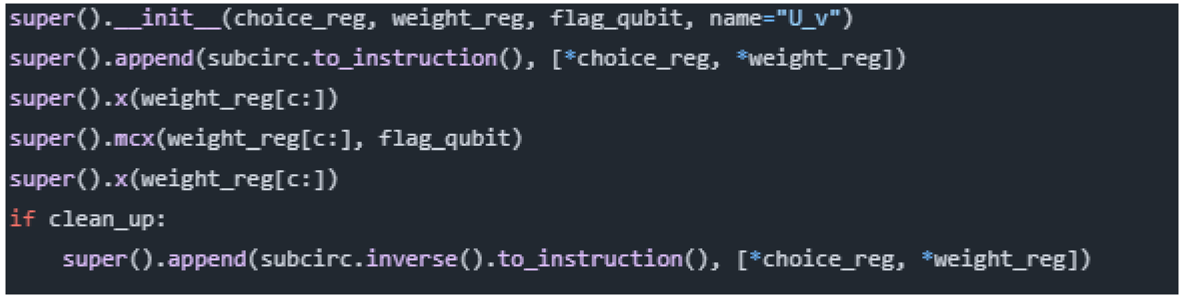
Those ancillary are the flag qubit K_F



After U_2 we get the feasible solution adhering to the Knapsack's capacity, it flips the ancillary qubit if the weight condition is satisfied

The second Oracle U_2

In the feasible oracle class of Circuits.py



```
super().__init__(choice_reg, weight_reg, flag_qubit, name="U_v")
super().append(subcirc.to_instruction(), [*choice_reg, *weight_reg])
super().x(weight_reg[c:])
super().mcx(weight_reg[c:], flag_qubit)
super().x(weight_reg[c:])
if clean_up:
    super().append(subcirc.inverse().to_instruction(), [*choice_reg, *weight_reg])
```

We now move forward towards the QWM-QAOA

The QAOA method is a Variational method especially used to solve the Combinatorial problem, where we do a minimization of the objective function

In our case the objective function is the Phase Hamiltonian H_c

First we initialize with a superposition state to get a distribution of all the states

Then we apply alternatively the Phase Evolution and the Mixing terms to get an ansatz state:

$$|\psi_p(\vec{\gamma},\vec{\beta})\rangle = e^{-i\beta_p B} e^{-i\gamma_p H_c} \dots e^{-i\beta_1 B} e^{-i\gamma_1 H_c} |+\rangle^{\otimes N}$$

The gamma and the beta are the variational parameters

Next step is to variationally determine the best result for

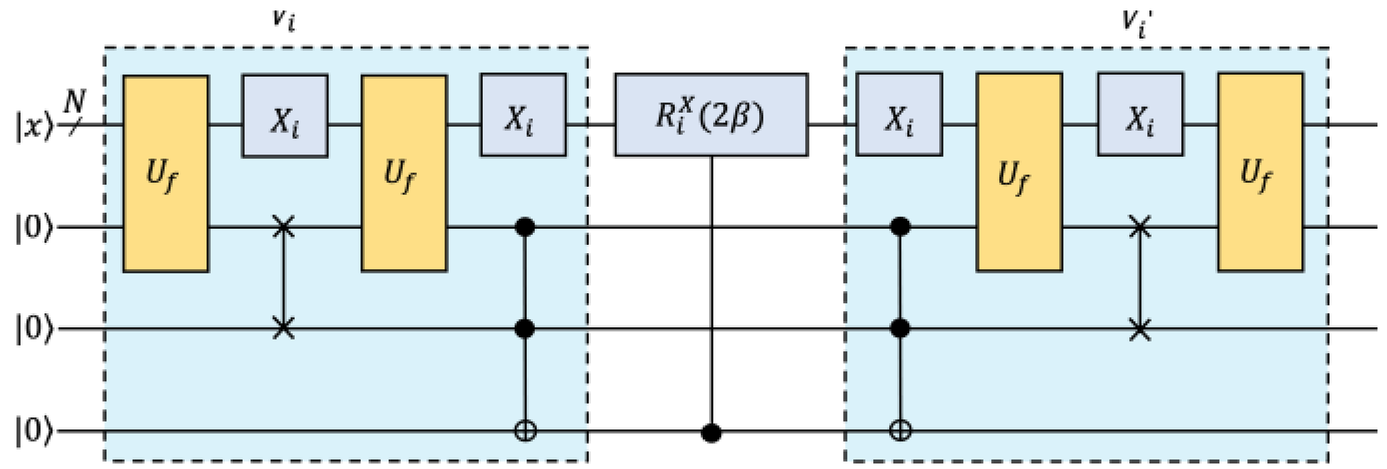
$$f_p(\vec{\gamma},\vec{\beta}) = \langle \psi_p(\vec{\gamma},\vec{\beta}) | H_c | \psi_p(\vec{\gamma},\vec{\beta}) \rangle$$

We now move forward towards the QWM-QAOA

Now for the Quantum Walk Mixer we use the Mixing Hamiltonian Tilda B

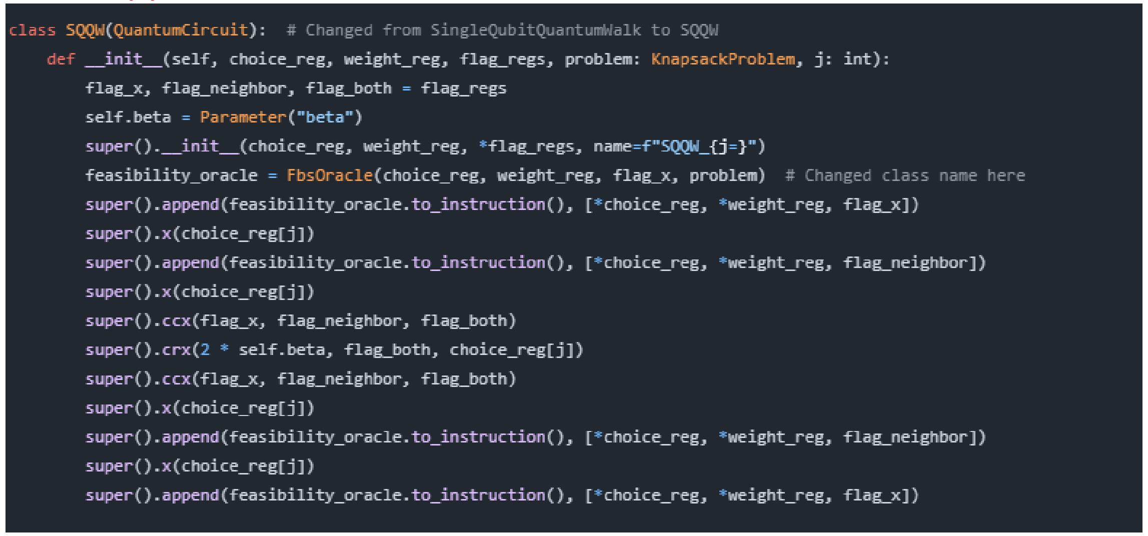
Which encodes the feasibility information of the neighbouring states into the ancillary qubits

It can be decomposed to two other Operations V and V†



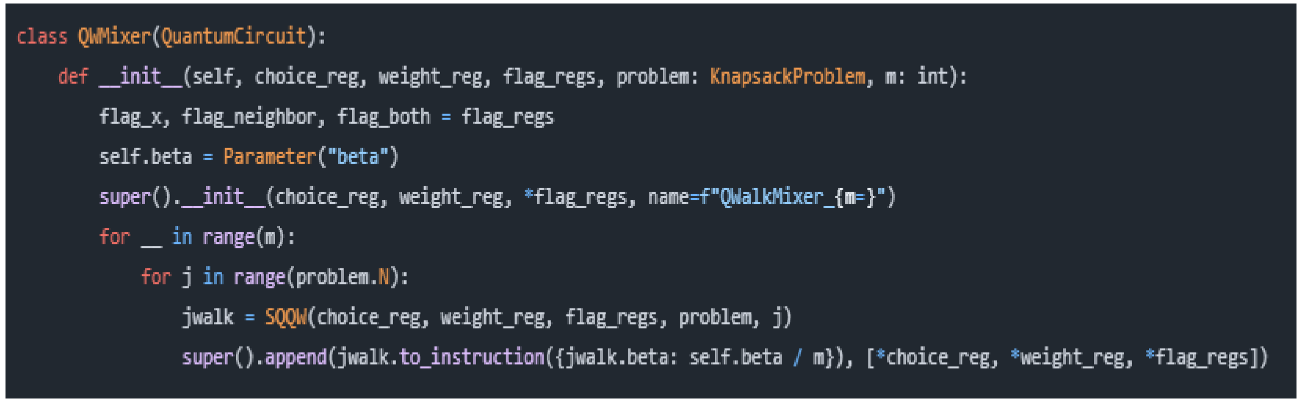
We now move forward towards the QWM-QAOA

Circuits.py



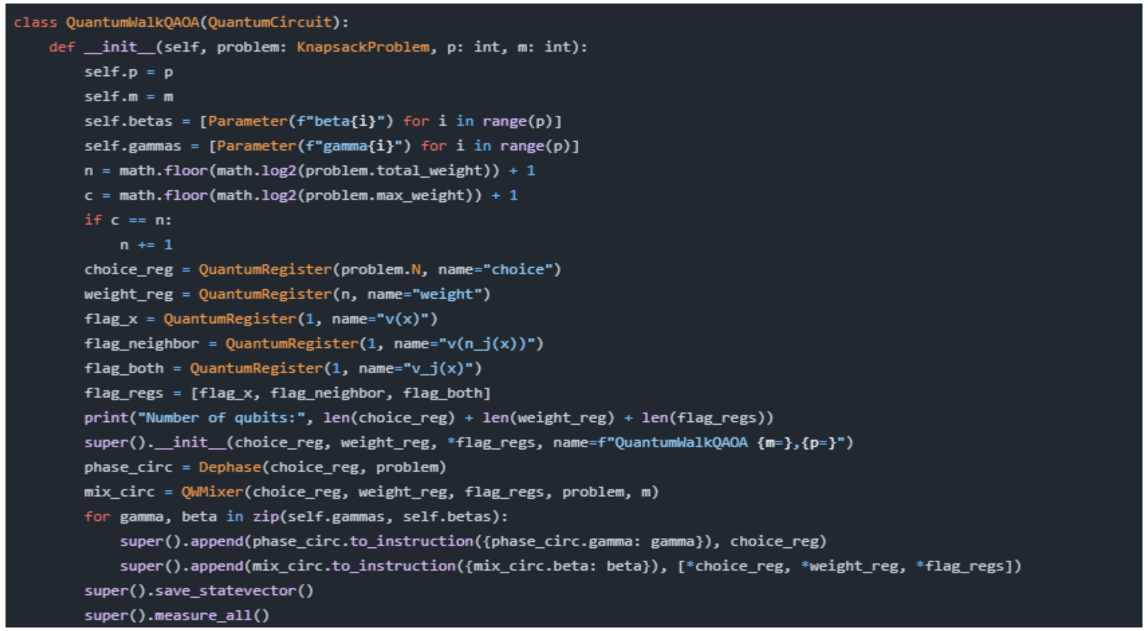
```python
class SQQW(QuantumCircuit):  # Changed from SingleQubitQuantumWalk to SQQW
    def __init__(self, choice_reg, weight_reg, flag_regs, problem: KnapsackProblem, j: int):
        flag_x, flag_neighbor, flag_both = flag_regs
        self.beta = Parameter("beta")
        super().__init__(choice_reg, weight_reg, *flag_regs, name=f"SQQW_{j=}")
        feasibility_oracle = FbsOracle(choice_reg, weight_reg, flag_x, problem)  # Changed class name here
        super().append(feasibility_oracle.to_instruction(), [*choice_reg, *weight_reg, flag_x])
        super().x(choice_reg[j])
        super().append(feasibility_oracle.to_instruction(), [*choice_reg, *weight_reg, flag_neighbor])
        super().x(choice_reg[j])
        super().ccx(flag_x, flag_neighbor, flag_both)
        super().crx(2 * self.beta, flag_both, choice_reg[j])
        super().ccx(flag_x, flag_neighbor, flag_both)
        super().x(choice_reg[j])
        super().append(feasibility_oracle.to_instruction(), [*choice_reg, *weight_reg, flag_neighbor])
        super().x(choice_reg[j])
        super().append(feasibility_oracle.to_instruction(), [*choice_reg, *weight_reg, flag_x])
```

Circuits.py



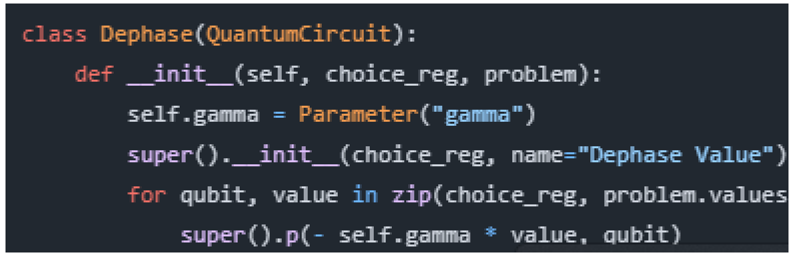
```python
class QWMixer(QuantumCircuit):
    def __init__(self, choice_reg, weight_reg, flag_regs, problem: KnapsackProblem, m: int):
        flag_x, flag_neighbor, flag_both = flag_regs
        self.beta = Parameter("beta")
        super().__init__(choice_reg, weight_reg, *flag_regs, name=f"QWalkMixer_{m=}")
        for _ in range(m):
            for j in range(problem.N):
                jwalk = SQQW(choice_reg, weight_reg, flag_regs, problem, j)
                super().append(jwalk.to_instruction({jwalk.beta: self.beta / m}), [*choice_reg, *weight_reg, *flag_regs])
```

QAOA.py



```python
class QuantumWalkQAOA(QuantumCircuit):
    def __init__(self, problem: KnapsackProblem, p: int, m: int):
        self.p = p
        self.m = m
        self.betas = [Parameter(f"beta{i}") for i in range(p)]
        self.gammas = [Parameter(f"gamma{i}") for i in range(p)]
        n = math.floor(math.log2(problem.total_weight)) + 1
        c = math.floor(math.log2(problem.max_weight)) + 1
        if c == n:
            n += 1
        choice_reg = QuantumRegister(problem.N, name="choice")
        weight_reg = QuantumRegister(n, name="weight")
        flag_x = QuantumRegister(1, name="v(x)")
        flag_neighbor = QuantumRegister(1, name="v(n_j(x))")
        flag_both = QuantumRegister(1, name="v_j(x)")
        flag_regs = [flag_x, flag_neighbor, flag_both]
        print("Number of qubits:", len(choice_reg) + len(weight_reg) + len(flag_regs))
        super().__init__(choice_reg, weight_reg, *flag_regs, name=f"QuantumWalkQAOA {m=},{p=}")
        phase_circ = Dephase(choice_reg, problem)
        mix_circ = QWMixer(choice_reg, weight_reg, flag_regs, problem, m)
        for gamma, beta in zip(self.gammas, self.betas):
            super().append(phase_circ.to_instruction({phase_circ.gamma: gamma}), choice_reg)
            super().append(mix_circ.to_instruction({mix_circ.beta: beta}), [*choice_reg, *weight_reg, *flag_regs])
        super().save_statevector()
        super().measure_all()
```

Circuits.py

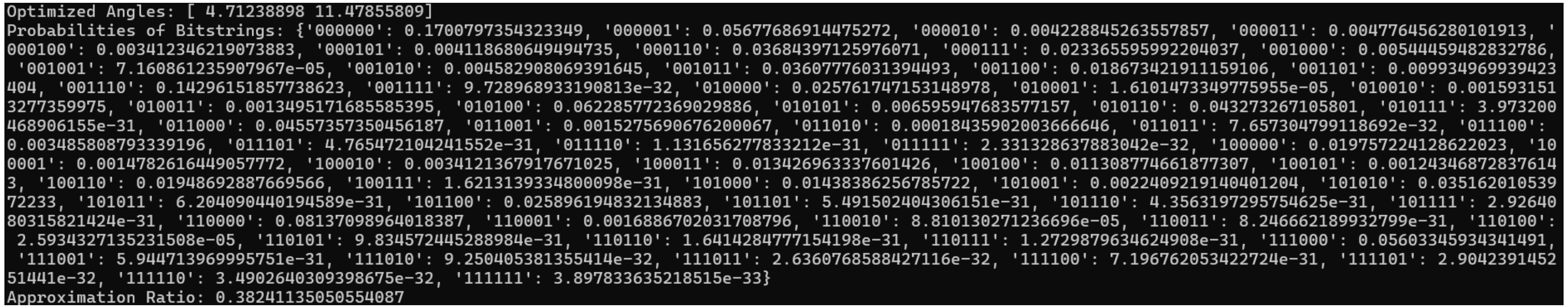


```python
class Dephase(QuantumCircuit):
    def __init__(self, choice_reg, problem):
        self.gamma = Parameter("gamma")
        super().__init__(choice_reg, name="Dephase Value")
        for qubit, value in zip(choice_reg, problem.values
            super().p(- self.gamma * value, qubit)
```

The Results

We use the Qiskit simulator to get our results in terms of probability distribution in binary values

The binary string with the highest amplitude is chosen for the best quantum result



```
Optimized Angles: [ 4.71238898 11.47855809]
Probabilities of Bitstrings: {'000000': 0.1700797354323349, '000001': 0.05677686914475272, '000010': 0.004228845263557857, '000011': 0.004776456280101913, '
000100': 0.003412346219073883, '000101': 0.004118680649494735, '000110': 0.03684397125976071, '000111': 0.023365595992204037, '001000': 0.005444594828327 86,
 '001001': 7.160861235907967e-05, '001010': 0.004582908069391645, '001011': 0.03607776031394493, '001100': 0.018673421911159106, '001101': 0.009934969939423
404, '001110': 0.14296151857738623, '001111': 9.728968933190813e-32, '010000': 0.025761747153148978, '010001': 1.6101473349775955e-05, '010010': 0.001593151
3277359975, '010011': 0.0013495171685585395, '010100': 0.062285772369029886, '010101': 0.006595947683577157, '010110': 0.043273267105801, '010111': 3.973200
468906155e-31, '011000': 0.04557357350456187, '011001': 0.0015275690676200067, '011010': 0.00018435902003666646, '011011': 7.657304799118692e-32, '011100':
0.003485808793339196, '011101': 4.765472104241552e-31, '011110': 1.131656277833212e-31, '011111': 2.331328637883042e-32, '100000': 0.019757224128622023, '10
0001': 0.0014782616449057772, '100010': 0.0034121367917671025, '100011': 0.013426963337601426, '100100': 0.011308774661877307, '100101': 0.0012434687283761 4
3, '100110': 0.01948692887669566, '100111': 1.6213139334800098e-31, '101000': 0.014383862567857 22, '101001': 0.0022409219140401204, '101010': 0.035162010539
72233, '101011': 6.204090440194589e-31, '101100': 0.025896194832134883, '101101': 5.491502404306151e-31, '101110': 4.3563197295754625e-31, '101111': 2.92640
80315821424e-31, '110000': 0.08137098964018387, '110001': 0.0016886702031708796, '110010': 8.810130271236696e-05, '110011': 8.246662189932799e-31, '110100':
 2.5934327135231508e-05, '110101': 9.834572445288984e-31, '110110': 1.6414284777154198e-31, '110111': 1.2729879634624908e-31, '111000': 0.05603345934341491,
 '111001': 5.944713969995751e-31, '111010': 9.250405381355414e-32, '111011': 2.636076858842711 6e-32, '111100': 7.196762053422724e-31, '111101': 2.9042391452
51441e-32, '111110': 3.4902640309398675e-32, '111111': 3.897833635218515e-33}
Approximation Ratio: 0.38241135050554087
```

The approximation ratio is based on the comparison of the results of the brute force method (which checks all the feasible solutions w.r.t the constraints mentioned)

https://github.com/jyotiraj-code/SQCC-Hacktathon-2.0-Quantum-Core