QHack

Quantum Coding Challenges

CHALLENGE COMPLETED View successful submissions

Jump to code — Collapse text

Unitary Operators and Beyond

200 points

Backstory

Zenda and Reece try to figure out Sqynet's Hamiltonian, before this eerie conscious quantum computer conquers the entirety of sector III. For this, they need to use their own (non-sentient) quantum computer to simulate the action of a Hamiltonian on a quantum state. How do they do this, if a Hamiltonian is, in general, not a unitary?

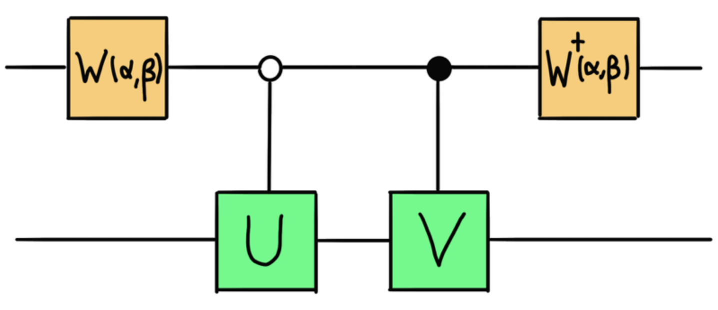
Linear combination of unitaries

Zenda and Reece know that the Hamiltonian that describes Sqynet is a linear combination of unitaries, that is

$$H = \sum_i \alpha_i U_i.$$

We know that quantum circuits can implement unitary operations really easily, but is there a way to implement a sum of unitaries? Note that the sum of unitaries is not always a unitary, so how can we even do this? We can use measurements!

A circuit of the form



will probabilistically implement the combination of unitaries $\alpha U + \beta V$ on the bottom (main) register, where α and β are **positive real numbers**, without loss of generality. Here, the single-qubit unitary $W(\alpha, \beta)$ is represented by the matrix

$$W(\alpha, \beta) = \frac{1}{\sqrt{\alpha + \beta}} \begin{pmatrix} \sqrt{\alpha} & -\sqrt{\beta} \\ \sqrt{\beta} & \sqrt{\alpha} \end{pmatrix}$$

The combination will only be applied on the bottom (main) register when we measure the state of the of the top (auxiliary) register to be $|0\rangle$.

Your task is to calculate the probability that this the linear combination of unitaries is implemented with the circuit above.

This algorithm is often used for Hamiltonian simulation. Check out the Xanadu Quantum Codebook to learn more!

Challenge code

You must complete the `linear_combination` function to build the above circuit that implements the linear combination

$$\alpha U + \beta V$$

of two single-qubit unitaries U and V, and returns the probabilities on the auxiliary register. For simplicity, we take α and β to be positive real numbers.

As a helper function, you are also asked to complete the `W` function, which returns the unitary $W(\alpha, \beta)$.

Input

As input to this problem, you are given:

- `U` (`list(list(float))`): A 2×2 matrix representing the single-qubit unitary operator U.
- `V` (`list(list(float))`): A 2×2 matrix representing the single-qubit unitary operator V
- `alpha` (`float`): The prefactor α of U in the linear combination, as above.
- `beta` (`float`): The prefactor β of V in the linear combination, as above.

Output

The output used to test your solution is a `float` corresponding to the probability of measuring $|0\rangle$ on the main register. This is the first element of your output of `linear_combination`. We will extract this element for you in our testing functions!

If your solution matches the correct one within the given tolerance specified in `check` (in this case it's an absolute tolerance of `0.001`), the output will be `"Correct!"` Otherwise, you will receive a `"Wrong answer"` prompt.

Good luck!

Code

Help

```python
import json
import pennylane as qml
import pennylane.numpy as np
```

```python
def W(alpha, beta):
    """ This function returns the matrix W in terms of
    the coefficients alpha and beta

    Args:
        - alpha (float): The prefactor alpha of U in the linear combination, as in the
        challenge statement.
        - beta (float): The prefactor beta of V in the linear combination, as in the
        challenge statement.
    Returns
        -(numpy.ndarray): A 2x2 matrix representing the operator W,
        as defined in the challenge statement
    """
```

```python
    # Put your code here #
    # Return the real matrix of the unitary W, in terms of the coefficients.
    return np.array([[np.sqrt(alpha), -np.sqrt(beta)],[np.sqrt(beta), np.sqrt(alpha)]])/(np.sqrt(alpha+I
```

```python
dev = qml.device('default.qubit', wires = 2)

@qml.qnode(dev)
def linear_combination(U, V,  alpha, beta):
    """This circuit implements the circuit that probabilistically calculates the linear combination
    of the unitaries.

    Args:
        - U (list(list(float))): A 2x2 matrix representing the single-qubit unitary operator U.
        - V (list(list(float))): A 2x2 matrix representing the single-qubit unitary operator U.
        - alpha (float): The prefactor alpha of U in the linear combination, as above.
        - beta (float): The prefactor beta of V in the linear combination, as above.

    Returns:
        -(numpy.tensor): Probabilities of measuring the computational
        basis states on the auxiliary wire.
    """
```

```python
    # Put your code here #
    # Return the probabilities on the first wire
    qml.QubitUnitary(W(alpha, beta), wires=0)
    #qml.PauliX(0)
    qml.ctrl(qml.QubitUnitary(U, wires=1), control=0, control_values=0)
    #qml.PauliX(0)
    # Put your code here #
    qml.ctrl(qml.QubitUnitary(V, wires=1), control=0)
    qml.adjoint(qml.QubitUnitary(W(alpha, beta), wires=0))
    # Return the probabilities on the first wire
    return qml.probs([0])
```

```python
# These functions are responsible for testing the solution.

def run(test_case_input: str) -> str:
    dev = qml.device('default.qubit', wires = 2)
    ins = json.loads(test_case_input)
    output = linear_combination(*ins)[0].numpy()

    return str(output)

def check(solution_output: str, expected_output: str) -> None:
    solution_output = json.loads(solution_output)
    expected_output = json.loads(expected_output)
    assert np.allclose(
        solution_output, expected_output, rtol=1e-4
    ), "Your circuit doesn't look quite right "
```

```python
test_cases = [['[[[ 0.70710678,  0.70710678], [ 0.70710678, -0.70710678]],[[1, 0], [0, -1]], 1, 3]',
```

```python
for i, (input_, expected_output) in enumerate(test_cases):
    print(f"Running test case {i} with input '{input_}'...")

    try:
        output = run(input_)

    except Exception as exc:
        print(f"Runtime Error. {exc}")

    else:
        if message := check(output, expected_output):
            print(f"Wrong Answer. Have: '{output}'. Want: '{expected_output}'.")

        else:
            print("Correct!")
```

Copy all Reset Open Notebook Submit