grover-answers

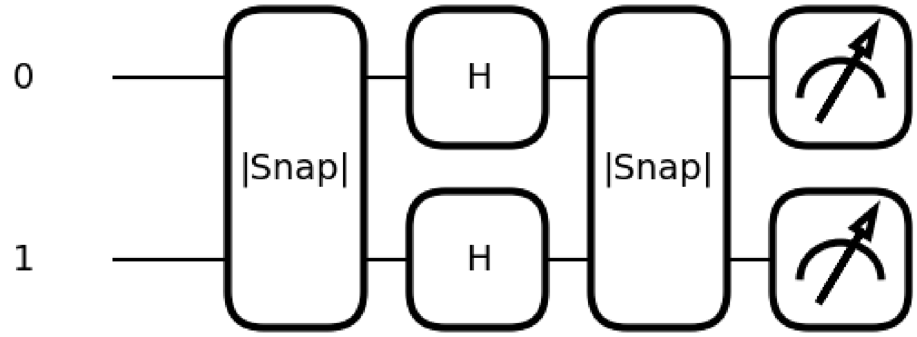
October 20, 2024

[1]:
```python
import pennylane as pl
from pennylane import numpy as np
import matplotlib.pyplot as plt
```

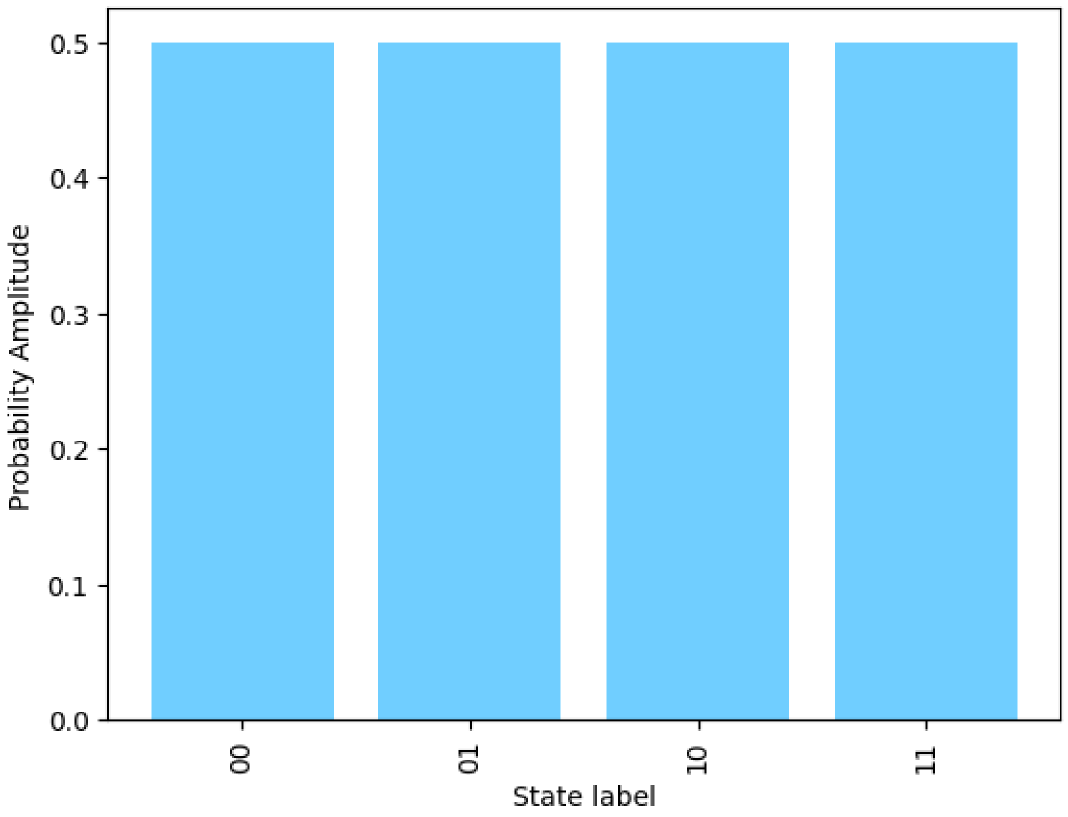
1 Exercise 1: Uniform superposition

[2]:
```python
n_bits = 2
dev = pl.device("default.qubit", wires=n_bits)
wires = list(range(n_bits))

@pl.qnode(dev)
def circuit():
    pl.Snapshot("Initial state")
    pl.broadcast(pl.Hadamard, wires=wires, pattern="single")
    pl.Snapshot("After applying the Hadamard gates")
    return pl.probs(wires=wires)

pl.drawer.use_style("black_white")
pl.draw_mpl(circuit)();
results = pl.snapshots(circuit)()
for k, result in results.items():
    print(f"{k}: {result}")
```

```
Initial state: [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
After applying the Hadamard gates: [0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j]
execution_results: [0.25 0.25 0.25 0.25]
```



[3]:
```
y = np.real(results["After applying the Hadamard gates"])
bit_strings = [f"{x:0{n_bits}b}" for x in range(len(y))]
plt.bar(bit_strings, y, color = "#70CEFF")
plt.xticks(rotation="vertical")
plt.xlabel("State label")
plt.ylabel("Probability Amplitude")
plt.show()
```

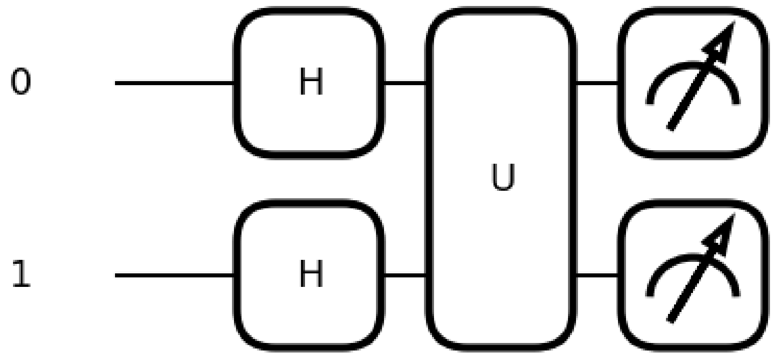


2 Exercise 2: Oracle

[4]:
```python
def oracle(keys):
    matrix = np.identity(2 ** n_bits)
    indices = [np.ravel_multi_index(key, [2]*len(key)) for key in keys]
    for i in range(len(keys)):
        matrix[indices[i], indices[i]] = -1
    return matrix

print(oracle([[0,0],[0,1]]))
```

```
[[-1.  0.  0.  0.]
 [ 0. -1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

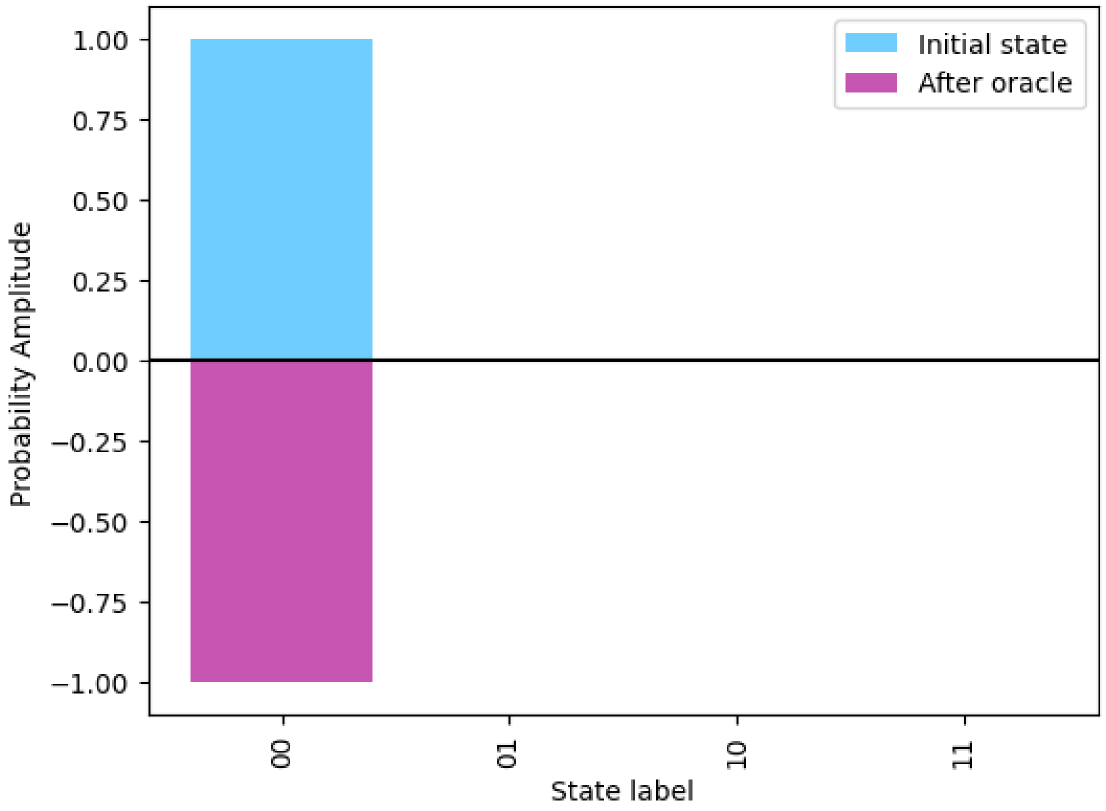
[5]:
```python
@pl.qnode(dev)
def circuit(keys):
    pl.broadcast(pl.Hadamard, wires=wires, pattern="single")
    pl.QubitUnitary(oracle(keys), wires=wires)
    return pl.probs(wires=wires)

pl.drawer.use_style("black_white")
pl.draw_mpl(circuit)([[0,0],[0,1]]);
```



3 Exercise 3: Amplitude amplification

[5]:
```
dev = pl.device("default.qubit", wires=n_bits)

@pl.qnode(dev)
def circuit(keys):
    pl.Snapshot("Initial state")
    pl.QubitUnitary(oracle(keys), wires=wires)
    pl.Snapshot("After oracle")
    return pl.state()

results = pl.snapshots(circuit)([[0,0]])
for k, result in results.items():
    print(f"{k}: {result}")
y1 = np.real(results["Initial state"])
y2 = np.real(results["After oracle"])

bit_strings = [f"{x:0{n_bits}b}" for x in range(len(y))]
plt.bar(bit_strings, y1, color = "#70CEFF")
plt.bar(bit_strings, y2, color = "#C756B2")
plt.xticks(rotation="vertical")
plt.xlabel("State label")
plt.ylabel("Probability Amplitude")
plt.legend(["Initial state", "After oracle"])
plt.axhline(y=0.0, color="k", linestyle="-")
plt.show()
```

```
Initial state: [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
After oracle: [-1.+0.j  0.+0.j  0.+0.j  0.+0.j]
execution_results: [-1.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

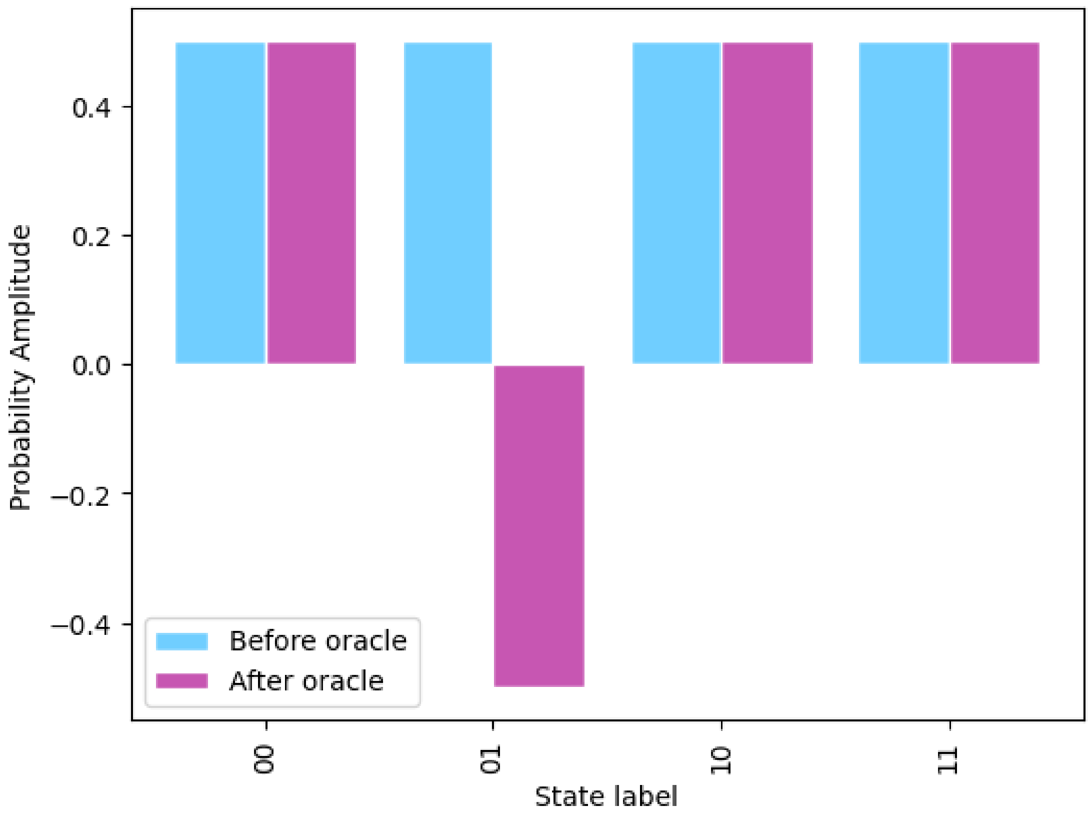


[7]:
```
keys = [[0,1]]
dev = pl.device("default.qubit", wires=n_bits)

@pl.qnode(dev)
def circuit():
    pl.broadcast(pl.Hadamard, wires=wires, pattern="single")
    pl.Snapshot("Before oracle")
    pl.QubitUnitary(oracle(keys), wires=wires)
    pl.Snapshot("After oracle")
    return pl.probs(wires=wires)
```

```
results = pl.snapshots(circuit)()
for k, result in results.items():
    print(f"{k}: {result}")
```

```
Before oracle: [0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j]
After oracle: [ 0.5+0.j -0.5+0.j  0.5+0.j  0.5+0.j]
execution_results: [0.25 0.25 0.25 0.25]
```

[8]:
```
y1 = np.real(results["Before oracle"])
y2 = np.real(results["After oracle"])

bit_strings = [f"{x:0{n_bits}b}" for x in range(len(y1))]

bar_width = 0.4
rect_1 = np.arange(0, len(y1))
rect_2 = [x + bar_width for x in rect_1]
plt.bar(
    rect_1,
    y1,
    width=bar_width,
    edgecolor="white",
    color = "#70CEFF",
    label="Before oracle",
)
plt.bar(
    rect_2,
    y2,
    width=bar_width,
    edgecolor="white",
    color = "#C756B2",
    label="After oracle",
)
plt.xticks(rect_1 + 0.2, bit_strings, rotation="vertical")
plt.xlabel("State label")
plt.ylabel("Probability Amplitude")
plt.legend()
plt.show()
```



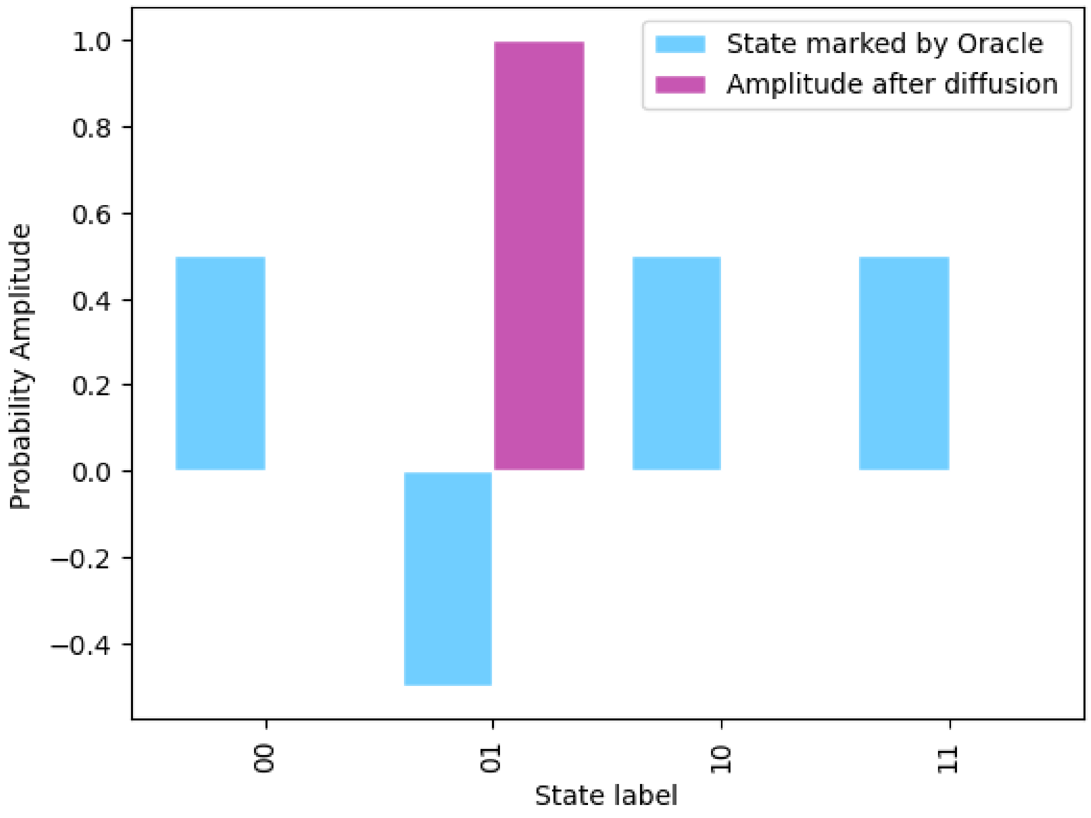
4 Exercise 4: Diffusion operator

[9]:
```python
def diffusion_operator(wires):
    for wire in wires:
        pl.Hadamard(wires=wire)
        pl.PauliZ(wires=wire)
    pl.ctrl(pl.PauliZ, 0)(wires=1)
    for wire in wires:
        pl.Hadamard(wires=wire)

@pl.qnode(dev)
def circuit():
    pl.broadcast(pl.Hadamard, wires=wires, pattern="single")
    pl.Snapshot("Uniform superposition")
    pl.QubitUnitary(oracle(keys), wires=wires)
    pl.Snapshot("State marked by Oracle")
    diffusion_operator(wires)
    pl.Snapshot("Amplitude after diffusion")
    return pl.probs(wires=wires)
```

```python
results = pl.snapshots(circuit)()
for k, result in results.items():
    print(f"{k}: {result}")
```

```
Uniform superposition: [0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j]
State marked by Oracle: [ 0.5+0.j -0.5+0.j  0.5+0.j  0.5+0.j]
Amplitude after diffusion: [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
execution_results: [0. 1. 0. 0.]
```

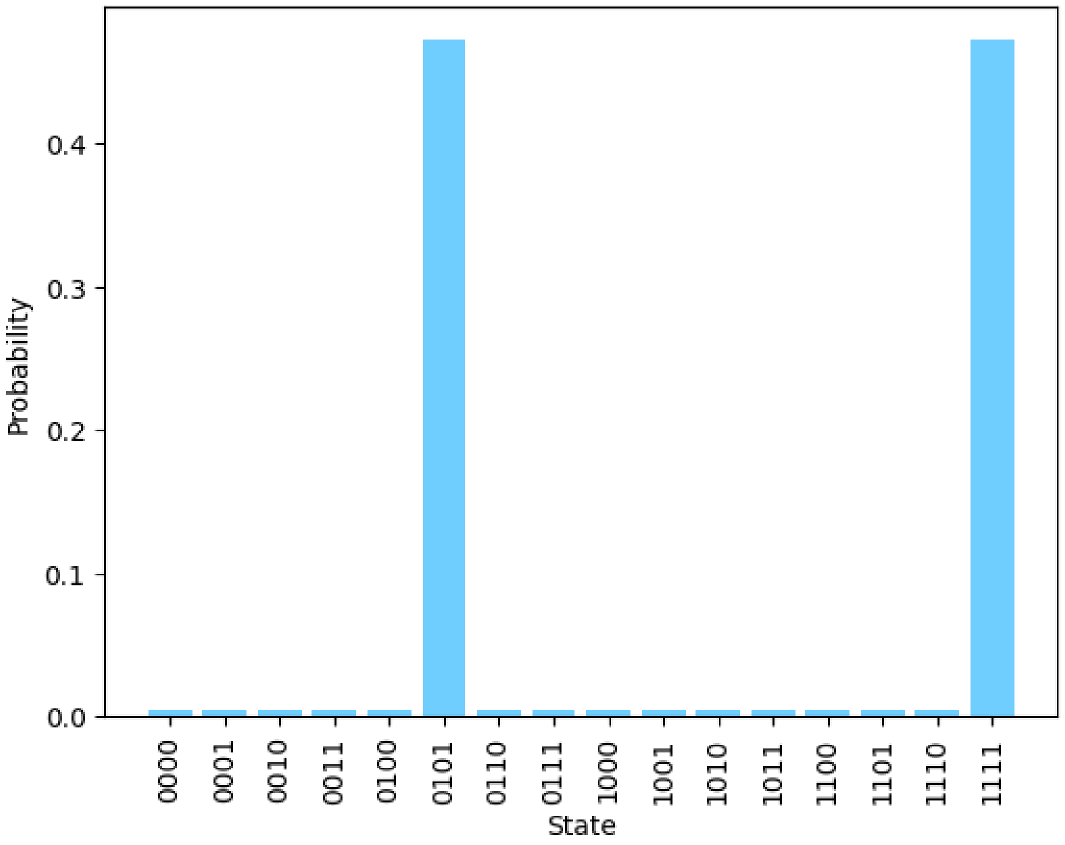
[10]:
```python
y1 = np.real(results["State marked by Oracle"])
y2 = np.real(results["Amplitude after diffusion"])

bit_strings = [f"{x:0{n_bits}b}" for x in range(len(y1))]

bar_width = 0.4
rect_1 = np.arange(0, len(y1))
rect_2 = [x + bar_width for x in rect_1]
plt.bar(
    rect_1,
    y1,
    width=bar_width,
    edgecolor="white",
    color = "#70CEFF",
    label="State marked by Oracle",
)
plt.bar(
    rect_2,
    y2,
    width=bar_width,
    edgecolor="white",
    color = "#C756B2",
    label="Amplitude after diffusion",
)
plt.xticks(rect_1 + 0.2, bit_strings, rotation="vertical")
plt.xlabel("State label")
plt.ylabel("Probability Amplitude")
plt.legend()
plt.show()
```



5 Exercise 5: Grover

```
[11]: n_bits = 4
dev = pl.device("default.qubit", wires=n_bits)
wires = list(range(n_bits))
keys = [[0,1,0,1],[1,1,1,1]]
M = 2
N = 2**n_bits

@pl.qnode(dev)
def circuit():
    iterations = int(np.round(np.sqrt(N / M) * np.pi / 4))
    pl.broadcast(pl.Hadamard, wires, pattern="single")
    for _ in range(iterations):
        pl.QubitUnitary(oracle(keys), wires)
        pl.templates.GroverOperator(wires=wires)
    return pl.probs(wires=wires)

results = pl.snapshots(circuit)()
y = results["execution_results"]
```

```
bit_strings = [f"{x:0{n_bits}b}" for x in range(len(y))]
plt.bar(bit_strings, results["execution_results"], color = "#70CEFF")
plt.xticks(rotation="vertical")
plt.xlabel("State")
plt.ylabel("Probability")
plt.show()
```


Probability
State