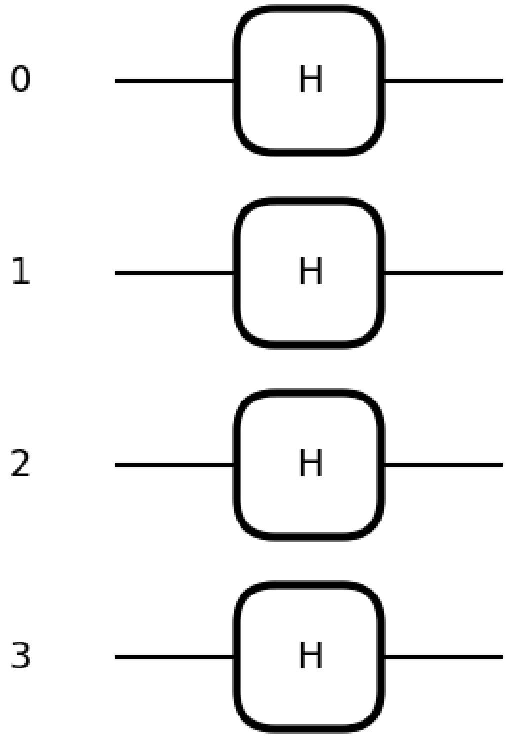
oraclesandpromises-answers

October 11, 2024

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

1 Exercise 1: Uniform superpositions

[2]:
```
n_bits = 4
dev = pl.device("default.qubit", wires=n_bits)

@pl.qnode(dev)
def uniformsuperposition():
    """Build a circuit that creates an n-qubit uniform superposition"""

    for i in range(n_bits):
        pl.Hadamard(wires=[i])
    return pl

pl.drawer.use_style("black_white")
pl.draw_mpl(uniformsuperposition)();
```



2 Exercise 2: Oracles

[4]:
```
def oracle_matrix(key):
    """Create the unitary matrix corresponding to the binary key (list[int])
       e.g. key=[0,1,1] should give the diagonal matrix [1,1,1,-1,1,1,1,1]"""

    matrix = np.identity(2**len(key))
    index = np.ravel_multi_index(key, [2]*len(key))
    matrix[index,index] = -1
    return matrix

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

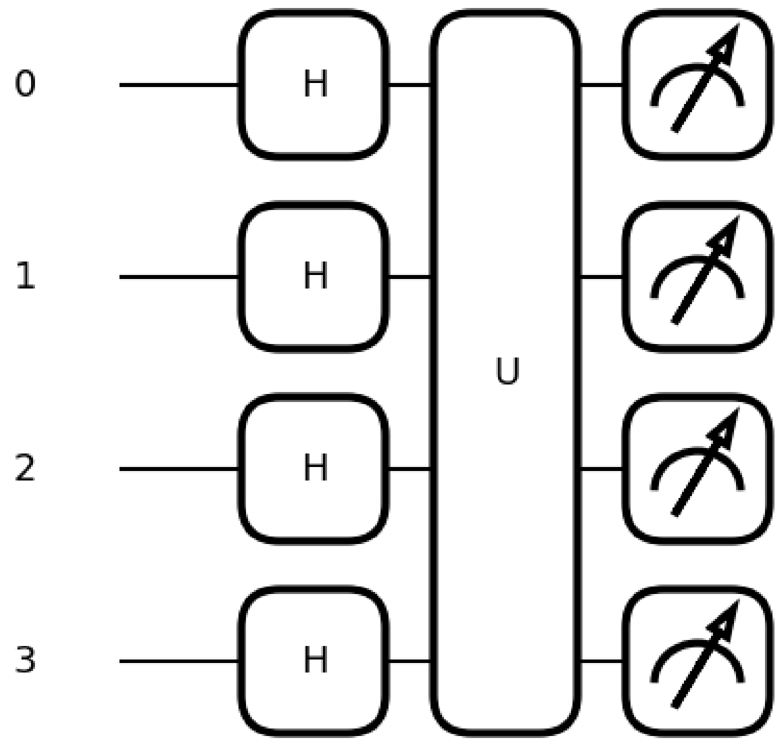
```
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

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

[5]:

```
@pl.qnode(dev)
def oracle_circuit(key):
    for i in range(n_bits):
        pl.Hadamard(wires=[i])
    pl.QubitUnitary(oracle_matrix(key), wires = list(range(n_bits)))
    return pl.probs(wires=range(n_bits))

pl.draw_mpl(oracle_circuit)([0,1,1,0]);
print(oracle_circuit([0,1,1,0]))
```

```
[0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625
 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625]
```

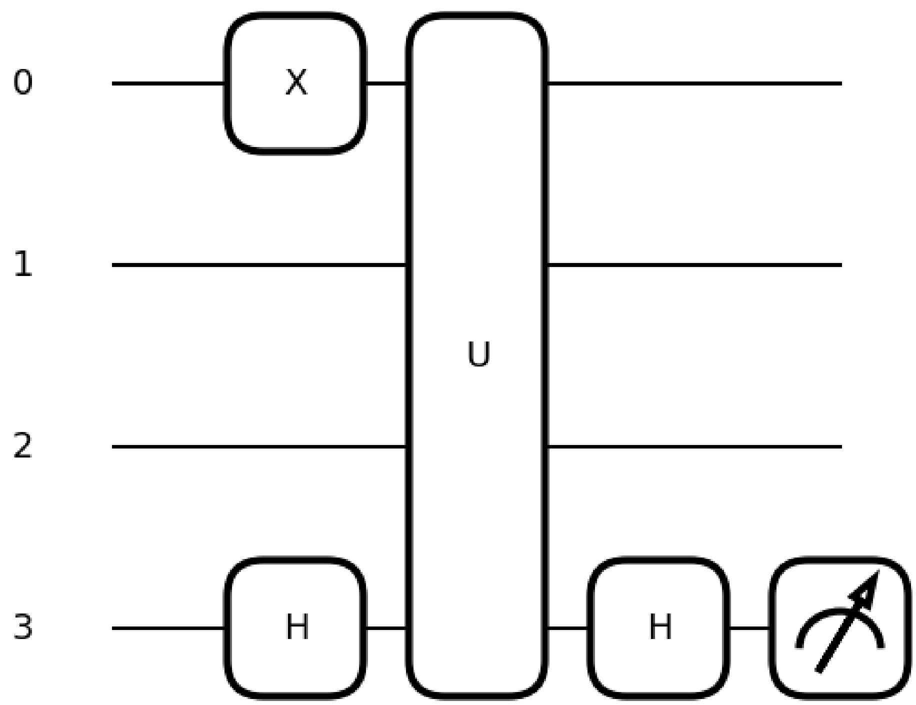


3 Exercise 3: Pair programming

[9]:
```
n_bits = 4
dev = pl.device("default.qubit", wires=n_bits)
@pl.qnode(dev)
def pair_circuit(probe, key):
    """Test whether probe (list[int]) contains a solution to key (list[int])"""
    for i in range(n_bits-1):
        if probe[i] == 1:
            pl.PauliX(wires=i)

    pl.Hadamard(wires = n_bits-1)
    pl.QubitUnitary(oracle_matrix(key), wires = list(range(n_bits)))
    pl.Hadamard(wires = n_bits-1)
```

```python
    return pl.probs(wires=n_bits-1)

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

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



[12]:
```python
secretkey = [0,1,0,1]

def pair_lock_picker(trials):
    keystrings = [np.binary_repr(n, n_bits-1) for n in range(2**(n_bits-1))]
    keys = [[int(s) for s in keystring] for keystring in keystrings]

    testnumbers = []

    for trial in range(trials):
        counter = 0
```

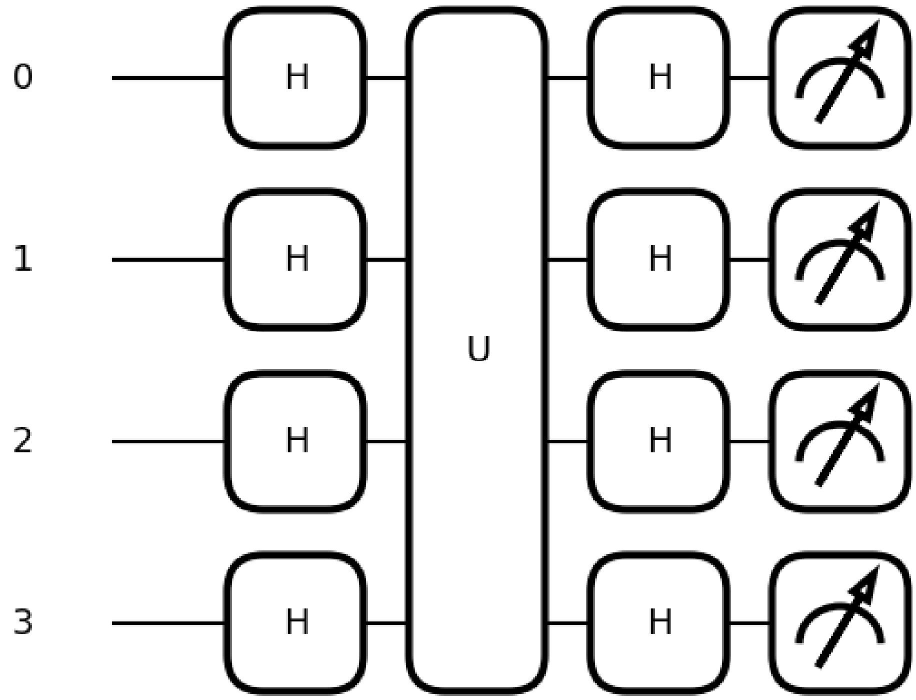
```
        for key in keys:
            counter += 1
            if np.isclose(pair_circuit(key, secretkey)[1], 1):
                break
        testnumbers.append(counter)
    return sum(testnumbers)/trials

trials = 500
output = pair_lock_picker(trials)

print(f"For {n_bits} bits, it takes", output, "pair tests on average.")
```

```
For 4 bits, it takes 3.0 pair tests on average.
```

4 Exercise 4: Deutsch-Jozsa

[13]:
```
n_bits = 4
dev = pl.device("default.qubit", wires=n_bits)

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

@pl.qnode(dev)
def deutschjozsa(keys):
    pl.broadcast(pl.Hadamard, wires = list(range(n_bits)), pattern = "single")
    pl.QubitUnitary(oracle(keys), wires = list(range(n_bits)))
    pl.broadcast(pl.Hadamard, wires = list(range(n_bits)), pattern = "single")
    return pl.probs(wires=range(n_bits))

keys =␣
 ↪([[0,0,0,0],[0,0,0,1],[1,0,0,0],[0,1,0,0],[1,1,0,0],[0,0,1,0],[0,0,1,1],[0,1,1,0]])
pl.draw_mpl(deutschjozsa)(keys)
print(deutschjozsa(keys))
if np.isclose(deutschjozsa(keys)[0],0):
    print("balanced")
else:
    print("constant")
```

```
[0.     0.25   0.0625 0.0625 0.0625 0.0625 0.     0.     0.25   0.
 0.0625 0.0625 0.0625 0.0625 0.     0.    ]
balanced
```

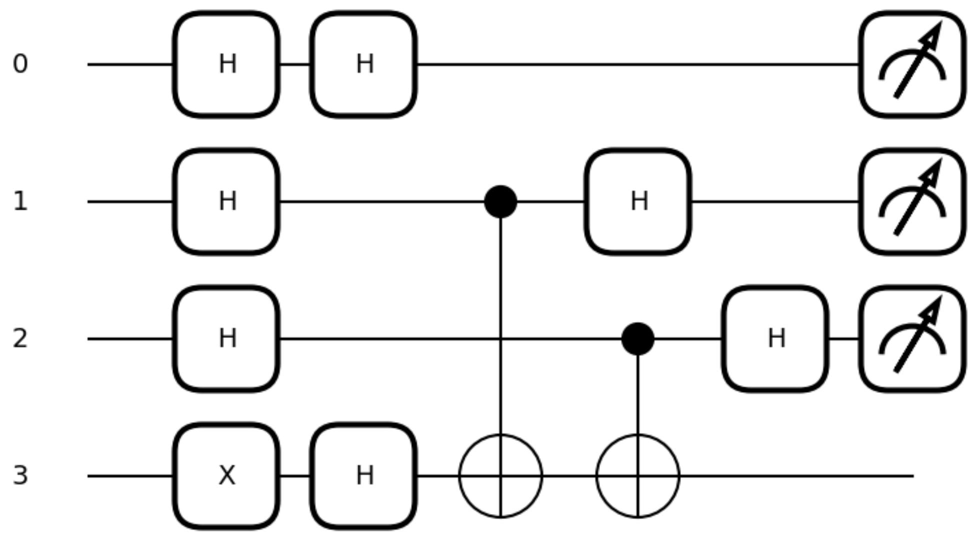


5 Exercise 5: Bernstein-Vazirani

[16]:
```
dev = pl.device("default.qubit", wires = 4, shots = 1)

def oracle():
    """Encode the hidden value in a circuit"""
    pl.CNOT(wires=[1, 3])
    pl.CNOT(wires=[2 ,3])

@pl.qnode(dev)
def bernsteinvazirani():
    """Sample the Bernstein-Vazirani circuit to return the hidden value"""
    pl.PauliX(wires = 3)
    for i in range(4):
        pl.Hadamard(wires = i)
    oracle()
    for i in range(3):
```

```
        pl.Hadamard(wires = i)
    return pl.sample(wires = range(3))

pl.drawer.use_style("black_white")
pl.draw_mpl(bernsteinvazirani)()
a = bernsteinvazirani()
print(f"The value of a is {a}")
```

```
The value of a is [0 1 1]
```




```
[ ]:
```