H2_LCU

May 19, 2025

[1]:
```
import numpy as np
from math import pi, cos
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, transpile
from qiskit_aer import AerSimulator
from collections import OrderedDict
from qiskit.circuit.library import QFT
from qiskit import QuantumRegister, QuantumCircuit, ClassicalRegister
from qiskit.quantum_info import Operator
import numpy as np



# H2 Hamiltoniaan (STO-3G) Pauli-termen en coëfficiënten
coeffs = [
    -0.8105479805373266,    # I (identiteit)
    0.17218393261915543,    # Z
    0.17218393261915543,    # Z
    -0.22278592816819915,   # Z Z
    0.16833606312315942,    # X X
    0.16833606312315942     # Y Y
]
lambda_sum = sum(abs(c) for c in coeffs)  #   = sum of |coeff| for␣
 ↪block-encoding

lam = lambda_sum
# Maak registers voor QPE, ancilla en systeem
qpe       = QuantumRegister(4, name="qpe")
anc_final = QuantumRegister(3, name="anc")   # block-encoding heeft 3 ancilla␣
 ↪bits
sys_final = QuantumRegister(2, name="sys")   # systeem is 2 qubits
c_reg     = ClassicalRegister(4, name="c")   # uitlezing 4 fase-bits

anc = QuantumRegister(3, name="anc")
sys = QuantumRegister(2, name="sys")

qc = QuantumCircuit(qpe, anc, sys)
```

```
# **Ancilla-PREP**: bereid superpositie |  =  √{|a_i|}/√ |i  met amplitude␣
↪volgens |coeff|
abs_coeffs = [abs(c) for c in coeffs]
norm = np.sqrt(sum(abs_coeffs))
amps = [0] * (2**anc.size)
for i, c in enumerate(coeffs):
    amps[i] = np.sqrt(abs(c)) / norm
# Bouw PREP als instructie

prep = QuantumCircuit(anc, name="PREP")
prep.prepare_state(amps, anc)         # ← here
prep_gate     = prep.to_gate(label="PREP")
prep_gate_inv = prep_gate.inverse()   # now works!

# ---------- helper: fase-flip op ancilla-basisstaat 'idx' --------------
def phase_flip(circ: QuantumCircuit, anc: QuantumRegister, idx: int):
    """Multicontrolled Z die een -1-fase geeft op ancilla |idx>."""
    bits = f"{idx:0{anc.size}b}"         # LSB = anc[0]
    # X op '0'-bits zodat |idx> → |111…>
    for j, b in enumerate(bits[::-1]):  # reverse order
        if b == '0':
            circ.x(anc[j])
    # CC…CZ (keer anc[0])
    circ.h(anc[0])
    circ.mcx(anc[1:], anc[0])
    circ.h(anc[0])
    # reset X
    for j, b in enumerate(bits[::-1]):  # reverse order
        if b == '0':
            circ.x(anc[j])



#␣
↪---------------------------------------------------------------------------------
# 1. Hulpfunctie: bouw een Pauli-circuit op basis van specificatie mét sign
#␣
↪---------------------------------------------------------------------------------
def _make_pauli_circ(n_sys: int,
                     spec: list[tuple[int, str, list[int]]]
                    ) -> QuantumCircuit:
    """
    Bouwt een QuantumCircuit met de opgegeven Pauli-operaties én een ±1 sign.
     - n_sys : aantal systeem-qubits
     - spec  : lijst van triples (sign, opname, [qubit_indices])
               waarbij sign  {+1, -1}
    """
```

```python
    qc = QuantumCircuit(n_sys)
    total_phase = 0.0

    for sign, opname, qs in spec:
        # eerst de Pauli's zelf
        for q in qs:
            getattr(qc, opname)(q)
        # als sign = -1, voeg dan   globaal toe
        if sign < 0:
            total_phase += np.pi

    # zet de globale fase op het circuit
    qc.global_phase = (qc.global_phase + total_phase) % (2*np.pi)
    return qc


# ---------------------------------------------------------------------------
# 2. Dictionary: ancilla-bitstring (MSB-order) → corresponderend Pauli-circuit
# ---------------------------------------------------------------------------
SELECT_MAP: dict[str, QuantumCircuit] = OrderedDict({
    "000": _make_pauli_circ(2, [(-1, "id", [0])]),     # P : I
    "001": _make_pauli_circ(2, [(+1, "z", [0])]),      # P : Z
    "010": _make_pauli_circ(2, [(+1, "z", [1])]),    # P : Z
    "011": _make_pauli_circ(2, [(-1, "z", [0,1])]),     # P : ZZ
    "100": _make_pauli_circ(2, [(+1, "x", [0,1])]),      # P : XX
    "101": _make_pauli_circ(2, [(+1, "y", [0,1])])  # P : YY
})

# ---------------------------------------------------------------------------
# 4. Beter: Functioneel: geef de gewenste anc-pattern als argument
# ---------------------------------------------------------------------------
def apply_single_select(circ: QuantumCircuit,
                        anc: list[int],
                        sys: list[int],
                        pattern_msb: str,
                        pauli_map: dict[str, QuantumCircuit] = SELECT_MAP) -> None:
    """
    c.q. vorige, maar neemt nu expliciet `pattern_msb` (bv. '011') als invoer.
    """
    if pattern_msb not in pauli_map:
        raise KeyError(f"No Pauli term defined for ancilla pattern '{pattern_msb}'")
    # Haal het circuit op en maak er een gate van

    pauli_gate = pauli_map[pattern_msb].to_gate(label=f"P{pattern_msb}")
```

```
    # ctrl_state verwacht LSB-order, dus reverse de bitstring
    ctrl_state = pattern_msb
    # Append de multi-controlled Pauli
    circ.append(pauli_gate.control(len(anc), ctrl_state=ctrl_state),
                anc[:] + sys[:])

    #print(f"Applied controlled {pattern_msb} to ancilla {anc} and system␣
 ↪{sys}")


def add_block_encoding(circ, anc, sys):
    circ.append(prep_gate, anc[:])

    for pattern_msb in SELECT_MAP:
        apply_single_select(circ, anc, sys, pattern_msb)
    # inverse PREP
    circ.append(prep_gate_inv, anc[:])
    #U_block = circ.to_gate(label="U_block")
    # U_block = circ.to_gate(label="U_block")  # corrected line


# **Walk-operator W**: twee keer block-encode, met reflectie R ertussen
def apply_walk(ctrl_qubit=None):

    # Eerste block-encoding
    print(ctrl_qubit)
    add_block_encoding(qc, anc, sys)
    # Reflectie R: reflecteer om |000_anc (met CCZ)
    # Inverteren van ancilla om 000 -> 111
    qc.x(anc)
    # Multi-controlled Z op |111>
    qc.h(anc[0])  # Hadamard op anc[0] om |000> te krijgen
    qc.mcx([anc[2], anc[1]], anc[0])      # 'mcx' is Qiskit's multi-controlled␣
 ↪X,
                                # hier gebruik je anc[0] als doel-qubit.
    qc.h(anc[0])  # Hadamard op anc[0] om |111> te krijgen                    ␣
 ↪
    # Dit voegt een -1 fase op |111> toe, dus op alle orginele states  |000>.
    # Vervolgens ancilla terug inverteren
    qc.x(anc)

# bouw Ede QP-circuit
qpe_circ = QuantumCircuit(qpe, anc_final, sys_final, c_reg, name="QPE")


# 1. Initialiseer ancilla en systeem
```

```
# Prepare the H  ground-state on the 2-qubit system: (|01> + |10>)/√2
eigvec = np.array([0, 1/np.sqrt(2), 1/np.sqrt(2), 0], dtype=complex)
eigvec = eigvec / np.sqrt(np.sum(np.abs(eigvec)**2))
qpe_circ.initialize(eigvec, sys_final)
#v_{\text{norm}} \approx \begin{pmatrix}0.9238795 \\ -0.3826834\end{pmatrix},

# 2. Hadamards op alle QPE qubits
qpe_circ.h(qpe)
# build W on the same registers you've declared above
walk_circ = QuantumCircuit(anc, sys, name="W")
# first block-encoding
add_block_encoding(walk_circ, anc, sys)

# reflection on |000>_anc
walk_circ.x(anc)
walk_circ.h(anc[0])
walk_circ.mcx([anc[2], anc[1]], anc[0])
walk_circ.h(anc[0])
walk_circ.x(anc)
# second block-encoding


# Make W into an explicit unitary instruction so that cW is known to Aer
W_op  = Operator(walk_circ)
W_gate = W_op.to_instruction()
W_gate.name = "W"
W2_op  = W_op.power(2);  W4_op = W_op.power(4); W8_op = W_op.power(8)
W2_gate = W2_op.to_instruction(); W2_gate.name = "W^2"
W4_gate = W4_op.to_instruction(); W4_gate.name = "W^4"
W8_gate = W8_op.to_instruction(); W8_gate.name = "W^8"
# append controlled-W, W^2, W^4 using QPE bits as controls
# each W_gate is 5-qubit wide, +1 control   6 total qubits
qpe_circ.append(W_gate.control(1),   [qpe[3]] + anc_final[:] + sys_final[:])
qpe_circ.append(W2_gate.control(1),  [qpe[2]] + anc_final[:] + sys_final[:])
qpe_circ.append(W4_gate.control(1),  [qpe[1]] + anc_final[:] + sys_final[:])
qpe_circ.append(W4_gate.control(1),  [qpe[0]] + anc_final[:] + sys_final[:])

# 4. Inverse Quantum Fourier Transform op QPE-register (4 qubits!)
qft_inv = QFT(num_qubits=4, inverse=True).to_gate(label="QFT†")
qpe_circ.append(qft_inv, qpe[:])

# 5. Meet het fase-register
qpe_circ.measure(qpe, c_reg)

from qiskit import transpile
from qiskit_aer import AerSimulator
# Set up the quantum simulator (use statevector to support custom unitaries)
```

```python
backend = AerSimulator(method="statevector")
# Transpile so that controlled-unitary gates get broken into supported primitives
qpe_circ = transpile(qpe_circ, backend)
# Execute the circuit on the simulator
#job = backend.run(qpe_circ, shots=1024)
job = backend.run(qpe_circ, shots=1024)
sim_result = job.result()
counts = sim_result.get_counts()
print(counts)

# Bereken fysische energie-eigenwaarden uit de gefilterde QPE-uitkomsten
t = 1
# take the most often obtained result (bitstring includes both qpe and ancilla bits)
raw_bits = max(counts, key=counts.get)
# extract only the QPE register bits (they come before the space)
phase_bits = raw_bits
print(phase_bits)
phase = 0
for index, bit in enumerate(phase_bits):
    phase += int(bit) / 2**(index + 1)
    print(phase)




#   = gemeten fase uit QPE; zorg dat    0.5
if phase > 0.5:
    phase = 1 - phase
theta = 2 * np.pi * phase


E = -lam * np.cos(theta)
print("Estimated eigenvalue of the Hamiltonian: {}".format(E))
```

```
{'0100': 2, '0000': 9, '1111': 170, '0001': 149, '1010': 26, '1110': 243,
'0011': 48, '0111': 13, '1011': 7, '1101': 46, '0010': 267, '0110': 25, '1000':
3, '0101': 1, '1001': 14, '1100': 1}
0010
0.0
0.0
0.125
0.125
Estimated eigenvalue of the Hamiltonian: -1.2122454103136884
```

[ ]:
```python
qpe_circ.draw("mpl", scale=0.01)
```

[ ]:
```
print(lam); print(np.linalg.norm(amps)**2)
```

[ ]:
```
Hsub = Operator(W_op).data[:4,:4] * lam
```

[ ]:
```
print(np.round(Hsub.real, 2))
```

[ ]:
```
from qiskit import QuantumRegister, QuantumCircuit, ClassicalRegister
from qiskit.quantum_info import Operator
import numpy as np
from qiskit.circuit.library import QFT

# Maak registers voor QPE, ancilla en systeem
qpe       = QuantumRegister(4, name="qpe")
anc_final = QuantumRegister(3, name="anc")   # block-encoding heeft 3 ancilla␣
 ↪bits
sys_final = QuantumRegister(2, name="sys")   # systeem is 2 qubits
c_reg     = ClassicalRegister(4, name="c")   # uitlezing 4 fase-bits

# bouw Ede QP-circuit
qpe_circ = QuantumCircuit(qpe, anc_final, sys_final, c_reg, name="QPE")


# 5.2 Stel ancilla in op |100> (MSB=1, rest=0)
#qc.x(anc_qr[0])
#qc.x(anc_qr[1])  # =0
#c.x(anc_qr[2])  # =0
qpe_circ = QuantumCircuit(qpe, anc_final, sys_final, c_reg, name="QPE")
# 1. Initialiseer ancilla en systeem
        # voorbereid ancilla zoals eerder (alternatief: begin in |00> en laat␣
 ↪PREP deel van U doen)
#qpe_circ.initialize([1,0], sys_final)             # bijvoorbeeld |0> als␣
 ↪starttoestand systeem
# Vervang de standaard initialisatie van de systemqubit
#eigvec = np.array([], dtype=complex)  # |0> als starttoestand systeem
# Bereken de ongewenste vector (proportioneel)
#eigvec = np.array([1, 1 - np.sqrt(2)], dtype=complex)
eigvec = np.array([1, 0])
# Normaliseer de vector
#eigvec = eigvec / np.linalg.norm(eigvec)
eigvec = eigvec / np.sqrt(np.sum(np.abs(eigvec)**2))

# Prepare the H  ground-state on the 2-qubit system: (|01> + |10>)/√2
eigvec = np.array([0, 1/np.sqrt(2), 1/np.sqrt(2), 0], dtype=complex)
qpe_circ.initialize(eigvec, sys_final)
#v_{\text{norm}} \approx \begin{pmatrix}0.9238795 \\ -0.3826834\end{pmatrix},

# 2. Hadamards op alle QPE qubits
```

```
qpe_circ.h(qpe)
# build W on the same registers you've declared above
walk_circ = QuantumCircuit(anc_final, sys_final, name="W")
# first block-encoding
add_block_encoding(walk_circ, anc_final, sys_final)
add_block_encoding(walk_circ, anc, sys)
# reflection on |000>_anc
walk_circ.x(anc)
walk_circ.h(anc[0])
walk_circ.mcx([anc[2], anc[1]], anc[0])
walk_circ.h(anc[0])
walk_circ.x(anc)
# second block-encoding
add_block_encoding(walk_circ, anc, sys)

# Make W into an explicit unitary instruction so that cW is known to Aer
W_op  = Operator(walk_circ)
W_gate = W_op.to_instruction()
W_gate.name = "W"
W2_op  = W_op.power(2);  W4_op = W_op.power(4); W8_op = W_op.power(8)
W2_gate = W2_op.to_instruction(); W2_gate.name = "W^2"
W4_gate = W4_op.to_instruction(); W4_gate.name = "W^4"
W8_gate = W8_op.to_instruction(); W8_gate.name = "W^8"
# append controlled-W, W^2, W^4 using QPE bits as controls
# each W_gate is 5-qubit wide, +1 control  6 total qubits
qpe_circ.append(W_gate.control(1),   [qpe[3]] + anc_final[:] + sys_final[:])
qpe_circ.append(W2_gate.control(1),  [qpe[2]] + anc_final[:] + sys_final[:])
qpe_circ.append(W4_gate.control(1),  [qpe[1]] + anc_final[:] + sys_final[:])
qpe_circ.append(W4_gate.control(1),  [qpe[0]] + anc_final[:] + sys_final[:])

# 4. Inverse Quantum Fourier Transform op QPE-register (4 qubits!)
qft_inv = QFT(num_qubits=4, inverse=True).to_gate(label="QFT†")
qpe_circ.append(qft_inv, qpe[:])

# 5. Meet het fase-register
qpe_circ.measure(qpe, c_reg)

from qiskit import transpile
from qiskit_aer import AerSimulator
# Set up the quantum simulator (use statevector to support custom unitaries)
backend = AerSimulator(method="statevector")
# Transpile so that controlled-unitary gates get broken into supported␣
↪primitives
qpe_circ = transpile(qpe_circ, backend)
# Execute the circuit on the simulator
#job = backend.run(qpe_circ, shots=1024)
job = backend.run(qpe_circ, shots=1024)
```

```python
sim_result = job.result()
counts = sim_result.get_counts()
print(counts)

# Bereken fysische energie-eigenwaarden uit de gefilterde QPE-uitkomsten
t = 1
# take the most often obtained result (bitstring includes both qpe and ancilla␣
↪bits)
raw_bits = max(counts, key=counts.get)
# extract only the QPE register bits (they come before the space)
phase_bits = raw_bits
print(phase_bits)
phase = 0
for index, bit in enumerate(phase_bits):
    phase += int(bit) / 2**(index + 1)
    print(phase)

lam = 2.5
# Bereken de eigenwaarde van de Hamiltoniaan
# met behulp van de gemeten fase
theta = 2*np.pi*phase
if theta < np.pi/2:
    E =  lam * np.cos(theta)
else:
    E = -lam * np.cos(theta)  # symmetrisch voor >



print("Estimated eigenvalue of the Hamiltonian: {}".format(E))
```

[ ]:
```python
qpe_circ.draw()
```

[ ]:
```python
from qiskit import QuantumCircuit, QuantumRegister,  transpile
from qiskit_aer import AerSimulator
import numpy as np


# --- H2 Pauli-coëfficiënten ---
coeffs = [
    -0.8105479805373266,
     0.17218393261915543,
     0.17218393261915543,
    -0.22278592816819915,
     0.16833606312315942,
     0.16833606312315942
]
lam = sum(abs(c) for c in coeffs)
```

```
# --- Ancilla register ---
anc = QuantumRegister(3, 'anc')
prep_circ = QuantumCircuit(anc)

# --- Amplitudes berekenen ---
amps = np.zeros(2**len(anc))
for i, c in enumerate(coeffs):
    amps[i] = np.sqrt(abs(c))
amps /= np.linalg.norm(amps)    # normaliseren

# --- PREP bouwen ---
prep_circ.initialize(amps, anc)

# --- simulatie ---
prep_circ.save_statevector()

backend = AerSimulator(method="statevector")
result = backend.run(prep_circ.decompose(reps=6), shots=1024)
statevector = result.result().get_statevector(prep_circ)



print("\nGegenereerde amplituden (PREP):")
for idx, amplitude in enumerate(statevector):
    if abs(amplitude) > 1e-6:    # alleen niet-nul
        print(f"|{idx:03b}>  amplitude: {amplitude:.6f}")
```

[ ]:
```
from qiskit import QuantumCircuit, QuantumRegister, transpile
import numpy as np
from qiskit_aer import AerSimulator

# ---------- registers -----------
anc = QuantumRegister(3, 'anc')
sys = QuantumRegister(2, 'sys')
qc  = QuantumCircuit(sys, anc)

# ---------- 1) prepare ancilla = |011> ----------
qc.x(anc[1]);
qc.x(anc[0])          # |011>  (binaire index 3)

# ---------- 2) prepare system  (|00>+|01>)/√2 --
qc.h(sys[0])                           # |00>+|01>

# ---------- 3) SELECT  (enkel Pauli-mapping) ----
def pauli_gate(idx):
    g = QuantumCircuit(2)
```

```python
    if idx == 1: g.z(0)                         # Z0
    elif idx == 2: g.z(1)                       # Z1
    elif idx == 3: g.z(0); g.z(1)               # Z0 Z1
    elif idx == 4: g.x(0); g.x(1)               # X0 X1
    elif idx == 5: g.y(0); g.y(1)               # Y0 Y1
    return g.to_gate(label=f"P{idx}")

# multi-controlled uitvoering: ancilla-bits (3) bepalen idx
for idx in range(6):
    gate = pauli_gate(idx)
    ctrl_state = format(idx, '03b')   # little endian
    qc.append(gate.control(3, ctrl_state=ctrl_state), anc[:] + sys[:])




# ---------- simulatie ---------------------------
qc.save_statevector()
# opslaan van de toestandvector
qc_t = transpile(qc, optimization_level=3)
backend = AerSimulator(method="statevector")
job = backend.run(qc_t, shots=2048).result()
state_prep = job.get_statevector()
counts  = job.get_counts()
print("raw counts:", counts)

print("\nStatevector na SELECT (niet-nul amplituden):")
for i,a in enumerate(state_prep):
    if abs(a) > 1e-9:
        print(f"|{i:05b}>  {a.real:+.3f}{a.imag:+.3f}j")

# ---------- verwachte vector --------------------
# indeling bits (anc[2] anc[1] anc[0]  sys[1] sys[0])
# anc = 011   indices 24 (=01100) en 25 (=01101)
psi_exact = np.zeros_like(state_prep)

psi_exact[12] =  1/np.sqrt(2)     # |01100>  (= |anc=011>|sys=00>)
psi_exact[13] = -1/np.sqrt(2)     # |01101>  (= |anc=011>|sys=01>) krijgt fase -
1
print(psi_exact)
print(state_prep)
print("\nFidelity met verwachte vector:",
      abs(np.vdot(psi_exact, state_prep))**2 )
```

[ ]:
```python
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, transpile
from qiskit.quantum_info import Operator
import numpy as np
```

```python
from qiskit_aer import AerSimulator
from qiskit.quantum_info import Operator, Statevector, DensityMatrix

# ---------------- registers -----------------
anc = QuantumRegister(3,'anc')
sys = QuantumRegister(2,'sys')
qc  = QuantumCircuit(anc, sys)

# H2 Hamiltoniaan (STO-3G) Pauli-termen en coëfficiënten
coeffs = [
    -0.8105479805373266,    # I (identiteit)
    0.17218393261915543,    # Z
    0.17218393261915543,    # Z
    -0.22278592816819915,   # Z Z
    0.16833606312315942,    # X X
    0.16833606312315942     # Y Y
]
lambda_sum = sum(abs(c) for c in coeffs)  #   = sum of |coeff| for block-encoding

# Registers: 5 QPE-qubits, 3 ancilla-qubits, 2 systeem-qubits
qpe = QuantumRegister(5, 'qpe')
anc = QuantumRegister(3, 'anc')
sys = QuantumRegister(2, 'sys')
c_qpe = ClassicalRegister(5, 'c_qpe')
c_anc = ClassicalRegister(3, 'c_anc')
qc = QuantumCircuit(qpe, anc, sys, c_qpe, c_anc)

# **Systeemvoorbereiding**: superpositie van |10> en |01> (2-elektron-toestanden)
qc.h(sys[0])
qc.x(sys[0])
qc.cx(sys[0], sys[1])
qc.x(sys[0])
# (Bovenstaande poortvolgorde creëert (|10> + |01>)/√2 op sys[0..1])

# **QPE-voorbereiding**: initialiseer alle 5 QPE-qubits in |+>
qc.h(qpe)

# **Ancilla-PREP**: bereid superpositie |  =  √{|a_i|}/√  |i  met amplitude volgens |coeff|
abs_coeffs = [abs(c) for c in coeffs]
norm = np.sqrt(sum(abs_coeffs))
amps = [0] * (2**anc.size)
for i, c in enumerate(coeffs):
    amps[i] = np.sqrt(abs(c)) / norm
# Bouw PREP als instructie
```

```python
prep = QuantumCircuit(anc, name="PREP")
prep.prepare_state(amps, anc)          # ← here
prep_gate     = prep.to_gate(label="PREP")
prep_gate_inv = prep_gate.inverse()   # now works!

lambda_sum = sum(abs(c) for c in coeffs)  #   = sum of |coeff| for block-encoding
# **Block-encode routine**: PREP -> SELECT -> PREP† (optioneel gecontroleerd door ctrl_qubit)

# Hulpfunctie: faseflip -1 toepassen op het ancilla-register voor toestand 'index'
def phase_flip(ctrl_qubit, index):
    bitstring = format(index, f"0{anc.size}b")  # binaire representatie (3 bits als string)
    # X-gates op ancilla-bits met '0' in het bitstring (bereid |111  toestand)
    for bit_pos, bit_char in enumerate(bitstring[::-1]):  # omgekeerd zodat anc[0] correspondeert met LSB
        if bit_char == '0':
            if ctrl_qubit:
                qc.cx(ctrl_qubit, anc[bit_pos])    # controleer X met ctrl_qubit indien aanwezig
            else:
                qc.x(anc[bit_pos])
    # Multi-controlled faseflip: H op anc[0], dan multi-controlled X (CCX of CCCX) en H terug
    qc.h(anc[0])
    controls = ([ctrl_qubit] if ctrl_qubit else []) + [anc[i] for i in range(1, anc.size)]
    qc.mcx(controls, anc[0])  # toepassen CCX/CCCX: indien alle controls = 1, flip anc[0]
    qc.h(anc[0])
    # X-gates terugzetten
    for bit_pos, bit_char in enumerate(bitstring[::-1]):
        if bit_char == '0':
            if ctrl_qubit:
                qc.cx(ctrl_qubit, anc[bit_pos])
            else:
                qc.x(anc[bit_pos])


def apply_block(ctrl_qubit=None):
    # PREP (voorbereiding ancilla)
    if ctrl_qubit:
        qc.append(prep_gate.control(1), [ctrl_qubit] + anc[:])
```

```python
    else:
        qc.append(prep_gate, anc[:])
    # Faseflip voor elke negatieve coëfficiënt (voegt -fase toe aan die amplitude)
    for idx, c in enumerate(coeffs):
        if c < 0:
            phase_flip(ctrl_qubit, idx)
    # SELECT: multi-controlled toepassen van elke Pauli-term op het systeem
    for idx, c in enumerate(coeffs):

        term_circ = QuantumCircuit(sys.size)
        if idx == 0:
            # Voor idx==0: implementeer -I door een globale fase (of een Z-poort op een vaste qubit)
            continue  # Dit introduceert een -1-fase op de toestand van qubit 0

        elif idx == 1:   # Z
            term_circ.z(0)
        elif idx == 2: # Z
            #term_circ.z(0)  # (let op: in 2-qubit mapping is Z1 dezelfde op beide qubits door symmetrie)
            term_circ.z(1)
        elif idx == 3: # Z Z
            term_circ.z(0); term_circ.z(1)
        elif idx == 4: # X X
            term_circ.x(0); term_circ.x(1)
        elif idx == 5: # Y Y
            term_circ.y(0); term_circ.y(1)

        term_gate = term_circ.to_gate(label=f"term_{idx}")
        # Stel gecontroleerde versie in: ancilla (en ctrl_qubit indien aanwezig) als controle
        if ctrl_qubit:
            #ctrl_state = "1" + format(idx, "0{}b".format(anc.size))  # qpe=1 + specifiek ancilla-patroon
            ctrl_state = "1" + format(idx, "0{}b".format(anc.size))[::-1]
            qc.append(term_gate.control(anc.size + 1, ctrl_state=ctrl_state),
                      [ctrl_qubit] + anc[:] + sys[:])
        else:
            ctrl_state = format(idx, "0{}b".format(anc.size))[::-1]
            qc.append(term_gate.control(anc.size, ctrl_state=ctrl_state), anc[:] + sys[:])
    # Faseflips ongedaan maken (inverse van eerdere faseflip voor negatieve termen)
    for idx, c in enumerate(coeffs):
        if c < 0:
```

```python
            phase_flip(ctrl_qubit, idx)
    # Inverse PREP (uncompute ancilla superpositie)
    if ctrl_qubit:
        qc.append(prep_gate_inv.control(1), [ctrl_qubit] + anc[:])
    else:
        qc.append(prep_gate_inv, anc[:])



# --------------- matrix-analyse -------------
# qc is uw samengestelde QPE-circuit
apply_block()

'''
unitary_circuit = qc.copy()  # Create a copy of the circuit for unitary␣
↪simulation
qc.save_unitary(label="unitary")  # Use save_unitary on the copied circuit
backend = AerSimulator(method="unitary")

Umat = backend.run(transpile(qc, backend)).result().get_unitary(qc)
# Unitary heeft dimensie 2**5 × 2**5  (32×32)

# isoleren 4×4-subblok voor anc=000   (dat is rij 0..3 en kolom 0..3)
Hblock = Umat[0:4, 0:4] * lambda_sum    # schaal terug met

print("Subblok linksboven (× ):")
print(np.round(Hblock.real, 6))

# --------------- verwachte matrix ----------
c = np.array([
    -0.8105479805373266,
     0.17218393261915543,
     0.17218393261915543,
    -0.22278592816819915,
     0.16833606312315942,
     0.16833606312315942
])
H_exp = np.array([
    [c[0]+c[1]+c[2]+c[3] , 0                   , 0                   ,␣
↪c[4]+c[5]],
    [0                   , c[0]+c[1]-c[2]-c[3] , c[4]-c[5]           , 0        ␣
↪],
    [0                   , c[4]-c[5]           , c[0]-c[1]+c[2]-c[3] , 0        ␣
↪],
    [c[4]+c[5]           , 0                   , 0                   ,␣
↪c[0]-c[1]-c[2]+c[3]]
])
```

```
print("\nVerwachte H-matrix × :")
print(np.round(H_exp, 6))
'''
```

```
[ ]: qc.draw( 'mpl', style={'dpi': 300})
```

```
[ ]: print(Operator(qc)) # unitary
```

```
[ ]: Umat = Operator(qc).data
```

```
[ ]: Umat
```

```
[ ]: Hblock = Umat[0:4, 0:4] * lambda_sum    # schaal terug met
```

```
[ ]: Hblock
```

```
[ ]: # --------------- check state-evolutie -------
qc2 = QuantumCircuit(anc, sys)
qc2.x(sys[0])                         # |10> op systeem
apply_block()                         # zelfde U

qc2.save_statevector()
# opslaan van de toestandvector
qc_t = transpile(qc2, optimization_level=3)
backend = AerSimulator(method="statevector")
job = backend.run(qc_t, shots=2048).result()
state_prep = job.get_statevector()
# amplitude-indices: anc (3 bits) vooraan, sys (2 bits) achteraan
amp_00010 = state_prep[int('00010',2)] * lambda_sum
amp_00001 = state_prep[int('00001',2)] * lambda_sum
print("\nAmplitude op |00010>: ", amp_00010)
print("Amplitude op |00001>: ", amp_00001)
print("Verhouding (theorie): ", (c[1]-c[2]), "/", (c[4]+c[5]))
```

```
[ ]: import numpy as np
from math import pi, cos, sqrt
from qiskit import QuantumCircuit, QuantumRegister, transpile
from qiskit.quantum_info import Operator
from qiskit_aer import AerSimulator

# ---------- H-coëfficiënten ----------
coeffs = [
    -0.8105479805373266,   # idx 0 :  -I
     0.17218393261915543,  # idx 1 :   Z
     0.17218393261915543,  # idx 2 :   Z
    -0.22278592816819915,  # idx 3 :  -Z Z
     0.16833606312315942,  # idx 4 :   X X
```

```
     0.16833606312315942   # idx 5 :   Y Y
]
lam = sum(abs(c) for c in coeffs)


# ---------- registers ----------
anc = QuantumRegister(3, 'a')
sys = QuantumRegister(2, 's')
qc  = QuantumCircuit(anc, sys)

bitstring = [
    "000",  # -I
    "001",  #  Z0
    "010",  #  Z1
    "011",  # -Z0Z1
    "100",  #  X0X1
    "101",  #  Y0Y1
]

for idx, c in enumerate(coeffs):
    pattern = bitstring[idx]              # MSB-volgorde
# ---------- PREP ----------
amps = np.zeros(8)
for i,c in enumerate(coeffs):
    amps[i] = sqrt(abs(c))
amps /= np.linalg.norm(amps)
prep = QuantumCircuit(anc, name='PREP')

prep.prepare_state(amps, anc)         # ← here
PREP  = prep.to_gate()
PREPi = PREP.inverse()

# ---------- fase-flip helper ----------
def phase_flip(idx):
      if c < 0:
        bits = pattern                    # GEEN[::-1]  (MSB)
        phase_flip_bits(bits)             # helper neemt bits in MS
    bits = f"{idx:03b}"
    # X op ancilla-bits met '0'
    for j,b in enumerate(bits[::-1]):
        if b == '0':
            qc.x(anc[j])
    qc.h(anc[0]); qc.ccx(anc[1], anc[2], anc[0]); qc.h(anc[0])
    for j,b in enumerate(bits[::-1]):
        if b == '0':
            qc.x(anc[j])
    print("flip", idx, bits) # bits zonder omkering
```

```python
# ---------- SELECT ----------
def apply_select():
    for idx,c in enumerate(coeffs):
        if idx == 0:                      # -I zit al in faseflips; geen poort
            continue
        t = QuantumCircuit(2)
        if idx==1: t.z(0)
        elif idx==2: t.z(1)
        elif idx==3: t.z(0); t.z(1)
        elif idx==4: t.x(0); t.x(1)
        elif idx==5: t.y(0); t.y(1)
        gate = t.to_gate(label=f"P{idx}")
        ctrl_state = f"{idx:03b}"[::-1]      #  ← MSB LSB !!
        qc.append(gate.control(3, ctrl_state=ctrl_state), anc[:] + sys[:])
        print(idx, ctrl_state)   # ctrl_state moet omgekeerd zijn

# ---------- BLOCK-encoding U ----------
def apply_block():
    qc.append(PREP, anc)
    for i,c in enumerate(coeffs):
        if c < 0: phase_flip(i)           # --teken
    apply_select()
    #for i,c in enumerate(coeffs):
    #    if c < 0: phase_flip(i)           # undo anc-fase
    qc.append(PREPi, anc)

# ---------- voeg BLOCK toe en bewaar unitary ----------
apply_block()
#qc.save_unitary()
print(Operator(qc)) # unitary

unitary = Operator(qc).data
# ---------- 4 × 4-subblok voor anc=000 ----------
Hblock = unitary[:4,:4] * lam
print("Subblok × :")
print(np.round(Hblock.real, 6))
```