# CSC299 - Report 2 - Implementation with 2 Ancilla

September 1, 2021

```
[1]: import tequila as tq
     from typing import Iterable
     import numpy as np
     from math import sqrt
     import random
     import copy
```

## 1 Goals of this report

We have previously seen how to create the LCU circuit in the case of having two unitary operations in our linear combination (i.e., where we only need one qubit ancilla). Now, we try to extend this definition of the LCU circuit to the case of having 2 qubits in the ancilla, which will allow for having up to 4 unitary operations in our expansion.

The main challenge in such a step is the fact that having a simplistic implementation of the select operator of the LCU algorithm will result in difficulty in implementing the prepare operator, and vice versa. For this report, we utilize the simple definition for the select operator (as explained in the previous report), with a trade-off of having a more detailed approach for the prepare operator.

## 2 Implementing Prepare operator

Now, for the prepare circuit, we are only concerned about what the $|00\rangle$ state gets mapped to, and we are not concerned about what the other states get mapped to, since we shall only ever apply the operator to the $|00\rangle$ state. The difficulty in such an implementation is that we do not know which circuit would give such a result, and hence, the problem effectively reduces to the general problem of state preparation.

This section utilizes techniques that were presented in Tequila's tutorial for state-preparation by Alba Cervera-Lierta (2020).

Reference used: Tequila state preparation tutorial

With that said, the following cells demonstrate how we defined the prepare circuit for the case of 2 qubit ancilla.

Firstly, we define the fidelity between two wavefunctions as an objective, as follows. Let $F(x, y)$ denote the fidelity

```
[2]: def fidelity(wfn_target: tq.wavefunction.qubit_wavefunction.QubitWaveFunction,
                 qc: tq.QCircuit) -> tq.Objective:
         """Return the fidelity between wfn_target and the expectation value of qc"""
         rho_targ = tq.paulis.Projector(wfn=wfn_target)
         objective = tq.Objective.ExpectationValue(U=qc, H=rho_targ)
         return objective
```

Next, we create a parametrized circuit over two qubits which will allow us to change the circuit as desired by changing some parameters $\theta, \phi, \lambda$.

```
[3]: def param_circ(anc: list) \
             -> tuple[list[tq.Variable], list[tq.Variable], list[tq.Variable], tq.
     ↪QCircuit]:
         """Return a parameterized quantum circuit which acts over two qubits
         with variables theta, phi, lambda for each qubit.

         Preconditions:
             - len(anc) == 2
         """

         # define variables
         th = [tq.Variable(name='theta_{}'.format(i)) for i in range(0, 4)]
         phi = [tq.Variable(name='phi_{}'.format(i)) for i in range(0, 4)]
         lam = [tq.Variable(name='lam_{}'.format(i)) for i in range(0, 4)]

         # PQC
         pqc = unitary_gate(th[0], phi[0], lam[0], anc[0]) + unitary_gate(th[1],␣
     ↪phi[1], lam[1], anc[1])
         pqc += tq.gates.CNOT(control=anc[0], target=anc[1])
         pqc += unitary_gate(th[2], phi[2], lam[2], anc[0]) + unitary_gate(th[3],␣
     ↪phi[3], lam[3], anc[1])

         return (th, phi, lam, pqc)
```

```
[4]: def unitary_gate(th, phi, lam, q0) -> tq.QCircuit:
         """Return a particular quantum gate that is not included in the basic gate␣
     ↪set"""
         ugate = tq.gates.Rz(target=q0, angle=phi) + tq.gates.Ry(target=q0,␣
     ↪angle=th) + tq.gates.Rz(
             target=q0, angle=lam)
         return ugate
```

Now, we simply rephrase the task of constructing the Prepare operator in the form of a minimization problem. We know the target state, as per the algorithm outlined in Report 1. So, all that remains is to construct an objective function which, in this case, the objective is the fidelity between the target state and the trial state obtained by the trial parametrized circuit. We do so as follows:

```
[5]: def prepare_operator_optimize_2anc(ancilla: list, unitaries: list[tuple[float,␣
     ↪tq.QCircuit]]) \
             -> tuple[tq.QCircuit, tq.optimizers, float]:
         """Return the circuit corresponding to the prepare operator.

         Preconditions:
             - all(coeff != 0 for coeff in [pair[0] for pair in unitaries])
             - len(ancilla) == 2
             - 2 < len(unitaries) <= 4
         """
         m = len(ancilla)

         # Define required state
         coefficients = [unit[0] for unit in unitaries]
         normalize = sqrt(sum(coefficients))

         coefficients = [sqrt(coeff) / normalize for coeff in coefficients]

         if len(coefficients) < 2 ** m:
             coefficients.append(0)

         wfn_target = tq.QubitWaveFunction.from_array(np.asarray(coefficients))
         wfn_target = wfn_target.normalize()

         # Create general parametric circuit
         th, phi, lam, pqc = param_circ(ancilla)
         n_th, n_phi, n_lam = len(th), len(phi), len(lam)

         # Initialize random wfn
         th0 = {key: random.uniform(0, np.pi) for key in th}
         phi0 = {key: random.uniform(0, np.pi) for key in phi}
         lam0 = {key: random.uniform(0, np.pi) for key in lam}
         initial_values = {**th0, **phi0, **lam0}

         # Define (in)fidelity
         wfn_pqc = tq.simulate(pqc, variables=initial_values)
         inf = fidelity(wfn_target, pqc)

         # Define bounds (if supported)
         min_angles, max_angles = 0, 4 * np.pi
         bnds_list = [[min_angles, max_angles]]
         for _ in range(len(initial_values)):
             bnds_list.append([min_angles, max_angles])
         th_dict = dict(zip([str(th[i]) for i in range(0, n_th)], bnds_list))
         phi_dict = dict(zip([str(phi[i]) for i in range(0, n_phi)], bnds_list))
         lam_dict = dict(zip([str(lam[i]) for i in range(0, n_lam)], bnds_list))
```

```
    bnds = {**th_dict, **phi_dict, **lam_dict}

    # Minimize objective
    # t0 = time.time()
    infid = tq.minimize(objective=inf, initial_values=initial_values,␣
↪method='TNC',
                        method_bounds=bnds, silent=True)
    # t1 = time.time()

    final_fidelity = 1 - infid.energy
    return pqc, infid, final_fidelity
```

Finally, we wish to implement a fail-safe in the case that the above method fails to give us an acceptable level of fidelity, which we define to be 0.99. To handle such a case, we ask the user to provide their own circuit to prepare the required state, if such a circuit is known already; if no such circuit is known, then we proceed with using the above function instead to come up with a good approximation instead. We take care of such aspects of the code in the final LCU function.

## 3 Implementing Select operator

Much of the implementation of the select operator remains unchanged from the previous report except for the one additional subroutine used to calculate how to use the controls for each controlled unitary in the expansion. The following cells show how this was implemented:

```
[6]: def controlled_unitary(ancilla: list, unitary: tq.QCircuit, n: int) -> tq.
↪QCircuit:
    """Return controlled version of unitary

    SHOULD NOT mutate unitary

    Preconditions:
        - ancilla and unitary cannot have any common qubits
        - 0 <= n < 2 ** len(ancilla)
    """
    m = len(ancilla)

    binary_list = _num_to_binary_list(m, n)
    # print(binary_list)

    # assert all([digit == 0 or digit == 1 for digit in binary_list])

    circuit = tq.QCircuit()
    for i in range(len(binary_list)):
        if binary_list[i] == 0:
            circuit += tq.gates.X(target=ancilla[m - i - 1])
    reverse_gates = circuit.dagger()
```

```python
    circuit += _control_unitary(ancilla, unitary)
    circuit += reverse_gates

    return circuit


def _control_unitary(ancilla, unitary: tq.QCircuit) -> tq.QCircuit:
    """Return controlled version of unitary

    SHOULD NOT mutate unitary

    Preconditions:
        - ancilla and unitary cannot have any common qubits
    """
    gates = unitary.gates
    cgates = []
    for gate in gates:
        cgate = copy.deepcopy(gate)
        if isinstance(ancilla, Iterable):
            control_lst = list(cgate.control) + list(ancilla)
        else:
            control_lst = list(cgate.control) + [ancilla]
        cgate._control = tuple(control_lst)
        cgate.finalize()
        cgates.append(cgate)

    return tq.QCircuit(gates=cgates)


def _num_to_binary_list(m: int, n: int) -> list[int]:
    """Return the binary representation of n in the form of a list of length m.

    Note: bin(int) exists but returns str

    Preconditions:
        - 2 ** m > n
    """
    # binary = bin(n)[2:]
    # binary_list = [int(digit) for digit in binary]

    binary = tq.BitString.from_int(integer=n, nbits=m)
    binary_list = binary.array

    # if len(binary_list) < m:
    #     k = len(binary_list)
    #     extend = [0 for _ in range(m - k + 1)]
    #     binary_list = extend + binary_list
    return binary_list
```

```
[7]: def select_operator(ancilla: list, sum_of_unitaries: list[tuple[float, tq.
     ↪QCircuit]]) \
            -> tq.QCircuit:
         """Return the circuit corresponding to the select operator

         Preconditions:
             - 2 ** (len(ancilla) - 1) < len(sum_of_unitaries) <= 2 ** len(ancilla)
         """
         unitaries = [pair[1] for pair in sum_of_unitaries]
         circuit = tq.QCircuit()

         for i in range(len(unitaries)):
             circuit += controlled_unitary(ancilla, unitaries[i], i)

         return circuit
```

We can now implement the entire LCU algorithm in Python as follows.

```
[8]: def lcu_2ancilla(ancilla, unitaries: list[tuple[float, tq.QCircuit]]) -> tq.
     ↪QCircuit:
         """Return the circuit for the LCU algorithm excluding the amplitude␣
     ↪amplification procedure"""
         prepare = prepare_operator_optimize_2anc(ancilla, unitaries)
         return prepare + select_operator_optimize_2anc(ancilla, sum_of_unitaries) +␣
     ↪prepare.dagger()
```

## 4 Amplitude amplification

Note that we can simply use the same amplitude amplification functions from the case with 1 ancilla qubit since out implementation does not depend on the number of qubits in the ancilla. Hence, we have the following functions, which were explained in Report 1.

```
[9]: def _num_iter(unitaries: list[tuple[float, tq.QCircuit]]) -> int:
         """Return the number of times to apply the amplitude amplificiation to␣
     ↪maximize
         success probability"""
         s = sum(pair[0] for pair in unitaries)
         alpha = arcsin(1 / s)
         frac = (pi / 2) / alpha
         return floor(0.5 * (frac - 1))


     def reflect_operator(state_qubits, ancilla) -> tq.QCircuit:
```

```
    """
    Return the reflection operator R = (I - 2P) \\otimes I_N,
    where:
        - I is the identity operator over the ancilla,
        - P is the projector onto the 0 state for the ancilla,
        - I_N is the identity operator over the state register

    """
    return tq.gates.X(target=ancilla) + tq.gates.X(control=ancilla,␣
 ↪target=state_qubits) \
           + tq.gates.X(target=ancilla)


def amp_amp_op(walk_op: tq.QCircuit, ancilla) -> tq.QCircuit:
    """Return W R W.dagger() R,
     where R is the reflect operator returned by the function␣
 ↪reflect_operator"""
    anc_qubits = ancilla if isinstance(ancilla, list) else [ancilla]
    state_qubits = [qubit for qubit in walk_op.qubits if qubit not in␣
 ↪anc_qubits]

    reflect = reflect_operator(state_qubits=state_qubits, ancilla=ancilla)

    return reflect + walk_op.dagger() + reflect + walk_op


def amp_amp(unitaries: list[tuple[float, tq.QCircuit]], walk_op: tq.QCircuit,␣
 ↪ancilla) \
        -> tq.QCircuit:
    """Amplitude amplification procedure obtained by repeating the amplitude␣
 ↪amplification
    step for a total of s times where s is the result of function _num_iter()
    """
    amplification_operator = amp_amp_op(walk_op, ancilla)
    s = _num_iter(unitaries)

    sum_of_steps = tq.QCircuit()
    for _ in range(s):
        sum_of_steps += amplification_operator

    return walk_op + sum_of_steps
```

## 5 References

- Cervera-Lierta, A., 2020. Tequila State Preparation Tutorial. GitHub. Available at: https://github.com/aspuru-guzik-group/tequilatutorials/blob/main/StatePreparation_tutorial.ipynb {Accessed June 8, 2021}.