CSC299 - Report 3 - General Implementation

September 1, 2021

[10]:
```
# Imports (must add to requirements)
# Requirements: Python 3.9+

import tequila as tq
from warnings import warn
from numpy import pi, sqrt, arcsin, asarray, floor
import time
from typing import Optional, Iterable, Union, Any
import copy
from random import uniform

# Global variables
TARGET_FIDELITY = 0.99
```

1 Introduction

In this report, we shall focus on generalizing the implementation of the LCU algorithm to beyond the one- and two-qubit ancilla cases. Note that we can simply reuse the amplitude amplification routine from Report 1 and the implementation of the Select operator from Report 2, since both of these functions have no inherent restrictions on the size of the ancillae used.

Hence, the primary emphasis of this report shall be to construct a functional implementation of the Prepare operator in Python. We also note the fact that we are not currently concerned about the optimality of such an implementation and, as such, the main intention is to simply have a functional program which passes all our tests. The subject of reaching optimality for a program in a generalized setting is often a difficult endeavour, but we shall, nonetheless, introduce certain altenative implementations and potential improvements in certain scenarios in Report 5.

2 Implementing Prepare operator

2.1 Setting up the stage

Firstly, recall the strategy that we used to construct a circuit for the Prepare operator for the case where we have only two ancilla qubits. In that case, we constructed a parameterized gate and adjusted the parameters using a variational minimization algorithm. We shall obtain inspiration from that approach and proceed in a similar fashion for the generalized case as well.

We define a helper function to outsource preparing the target wavefunction, $|\psi\rangle$, since this process

remains the same as compared to the case for the two-qubit ancilla. This function, _target_wfn, is shown below.

Next, we define the $|0\rangle$ state for the ancilla. This is computed in Python by calculating how many terms are provided in the linear combination and, if necessary, extending the list of coefficients by appending zeroes to the end of the list based on the number of qubits in the ancilla.

2.2 Generator-based variational optimization

We then define the required generator which would give us the required circuit if we used an infinite Trotter series expansion. We define the following terms:

$$G_1 = |\psi\rangle\langle 0|,$$
$$G_2 = |0\rangle\langle\psi|$$

Note that, because we want to prepare the state $|\psi\rangle$ from the $|0\rangle$ state, applying G_1 to $|0\rangle$ is what we essentially seek for.

To obtain Hermiticity, we thus define the generator, G, as $G = i(G_1 - G_2) = i(|\psi\rangle\langle 0| - |0\rangle\langle\psi|)$. The reason for subtracting G_2 from G_1 instead of adding is to avoid encountering a complex phase in the result, as we shall see later. Note that G has three distinct eigenvalues: 0, +1, and -1. As can be verified easily, the state corresponding to the eigenvalue +1 is $|\text{Plus}\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |\psi\rangle)$. Similarly, the state corresponding to the eigenvalue -1 is $|\text{Minus}\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |\psi\rangle)$.

Finally, we can see that all states which are orthogonal to $|\text{Plus}\rangle$ and $|\text{Minus}\rangle$ are eigenstates corresponding to the eigenvalue 0. Thus, we define the null space projector of G as $Q = I - |\text{Plus}\rangle\langle\text{Plus}| - |\text{Minus}\rangle\langle\text{Minus}|$.

We now make the assumption that $\langle 0|\psi\rangle = 0$. Thus, this implies that:

$$\begin{aligned}
G^2 &= i^2(|\psi\rangle\langle 0| - |0\rangle\langle\psi|)(|\psi\rangle\langle 0| - |0\rangle\langle\psi|) \\
&= |\psi\rangle\langle\psi| + |0\rangle\langle 0| \\
G^3 &= i(|\psi\rangle\langle\psi| + |0\rangle\langle 0|)(|\psi\rangle\langle 0| - |0\rangle\langle\psi|) \\
&= i(|\psi\rangle\langle 0| - |0\rangle\langle\psi|) \\
&= G
\end{aligned}$$

Therefore, we notice that, for any arbitrary $k \in \mathbb{Z}^+$, we have $G^{2k} = G^2$ and $G^{2k+1} = G$.

Now, consider the operation $U = e^{-i\frac{\theta}{2}G}$. Using the Taylor series expansion, we can thus express U as follows:

$$\begin{aligned}
U &= e^{-i\frac{\theta}{2}G} \\
&= \sum_{j=0}^{\infty} \frac{\left(-i\frac{\theta}{2}\right)^j}{j!} G^j \\
&= \cos\left(\frac{\theta}{2}\right) I - i\sin\left(\frac{\theta}{2}\right) G + \left(1 - \cos\left(\frac{\theta}{2}\right)\right) Q
\end{aligned}$$

Let S denote the 2D-subspace with a basis $b = (|\text{Plus}\rangle, |\text{Minus}\rangle)$. Thus, since $|0\rangle = \frac{1}{\sqrt{2}}(|\text{Plus}\rangle + |\text{Plus}\rangle)$, this implies that $|0\rangle \in S$ and hence, $Q|0\rangle = 0$.

Therefore, we have that:

$$\begin{aligned} U|0\rangle &= e^{-i\frac{\theta}{2}G}|0\rangle \\ &= \cos\left(\frac{\theta}{2}\right) I|0\rangle - i\sin\left(\frac{\theta}{2}\right) G|0\rangle + \left(1-\cos\left(\frac{\theta}{2}\right)\right) Q|0\rangle \\ &= \cos\left(\frac{\theta}{2}\right)|0\rangle - i\sin\left(\frac{\theta}{2}\right)|\psi\rangle \end{aligned}$$

If we want $U|0\rangle = |\psi\rangle$, this means that we must set $\theta = \pi$. However, this only gives an exact result in the case where $\langle 0|\psi\rangle = 0$ and we take an infinite Taylor series expansion. Since these assumptions are not particularly realistic, we must tweak the values of the exponents in our finite Taylor series expansion.

Let G be written as a sum of k Pauli-strings, as follows.

$$G = \sum_{j=0}^{k-1} c_j P_j$$

In this notation, the c_j's are some constants and P_j's are some products of Pauli operations on the ancilla qubits. We now consider the Trotter expansion of U with s steps, which we denote by $\tilde{U}$, as follows:

$$\begin{aligned} \tilde{U} &= \prod_{r=0}^{s-1}\prod_{j=0}^{k-1} e^{-i\frac{\pi}{2s}c_j P_j} \\ &\approx e^{-i\frac{\pi}{2}\sum_{j=0}^{k-1} c_j P_j} \\ &= e^{-i\frac{\pi}{2}G} \end{aligned}$$

However, since we are now considering a finite expansion, this results in some difference between the actual desired circuit and the circuit obtained with $\tilde{U}$. To counter-act this change, we introduce variables $t_{r,j}$ into the exponents, all initialzed to 1 in the first ansatz, as follows:

$$\tilde{U} = \prod_{r=0}^{s-1}\prod_{j=0}^{k-1} e^{-i\frac{\pi}{2s}c_j t_{r,j} P_j}$$

Next, we aim to change the values of the parameters in this circuit so that the expectation value of the result of applying this circuit to the $|0\rangle$ state of the ancilla is close to the target state. We use the fidelity of the expectation value of the circuit with the target state as the objective function in this optimization and aim to maximize it to as close to 1 as is feasible.

Since Tequila objectives are fully differentiable (as explained by Kottmann et al. (2021)), all objectives in Tequila support analytic gradients. Further, since our initial guess is a good guess in itself, as it is based on the values expected with an infinite Trotter series expansion, this implies that we can simply use gradient-based optimization.

2.3 Custom errors and warnings

Of course, we cannot guarantee, with a variational algorithm used as a black-box for the optimization process, that we will be able to construct an optimal circuit for every list of coefficients in the linear combination. To address such an issue, we allow the users to optionally specify some threshold fidelity of the expectation value of the resultant wavefunction with the target wavefunction. The threshold is initially set to 0.99, or 99%, if the user does not pass in any particular value. If the resulting circuit at the end of the function fails to achieve this fidelity, then we raise a custom warning to the user. We leave it up to the user whether they wish to try and catch the warning in hopes of pursuing alternative implementations, or proceed anyways.

We also define a custom error for when the size of the ancilla is too small for the number of unitaries provided in the linear combination. When raised, this prompts the user to input some additional qubits for the ancilla register.

[2]:
```
class LCUFidelityWarning(UserWarning):
    """Warning raised when resulting state has low fidelity with target state.
 ↪"""

    def __str__(self) -> str:
        """Return a string representation of this warning."""
        return 'Resulting state may have a lower fidelity with target state␣
 ↪than ideally expected'


class LCUMismatchWarning(UserWarning):
    """Warning raised when length of ancilla does not match with the number of␣
 ↪unitaries."""

    def __str__(self) -> str:
        """Return a string representation of this warning."""
        return 'Length of given ancilla does not match with the number of␣
 ↪unitaries'
```

[3]:
```
def _target_wfn(ancilla: list[Union[str, int]],
                unitaries: list[tuple[float, tq.QCircuit]]) -> tq.
 ↪QubitWaveFunction:
    """Return the ideal target wavefunction expected after applying the prepare␣
 ↪operator to the
    zero state of the ancilla register"""
    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:
        extension = [0 for _ in range(2 ** m - len(coefficients) + 1)]
        coefficients.extend(extension)

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

    return wfn_target


def prepare_operator(ancilla: list, unitaries: list[tuple[float, tq.QCircuit]],
                     steps: Optional[int] = 1, tolerance: Optional[float] =␣
↪TARGET_FIDELITY,
                     debug: Optional[bool] = False) -> tq.QCircuit:
    """Return the circuit corresponding to the prepare operator
    Preconditions:
        - 2 ** (len(ancilla) - 1) < len(sum_of_unitaries) <= 2 ** len(ancilla)
        - int > 0
    """
    wfn_target = _target_wfn(ancilla=ancilla, unitaries=unitaries)

    # Define zero state
    m = len(ancilla)

    coefficients = [unit[0] for unit in unitaries]
    if len(coefficients) < 2 ** m:
        extension = [0 for _ in range(2 ** m - len(coefficients) + 1)]
        coefficients.extend(extension)

    zero_state_coeff = [1.0] + [0 for _ in range(len(coefficients) - 1)]
    zero_state = tq.QubitWaveFunction.from_array(asarray(zero_state_coeff))

    # Define generators
    generator_1 = tq.paulis.KetBra(bra=wfn_target, ket=zero_state.normalize())
    generator_2 = tq.paulis.KetBra(ket=wfn_target, bra=zero_state.normalize())

    g = 1.0j * (generator_1 - generator_2)

    # Don't remove this line! It's required!! This line was added because of␣
↪the way
    # Tequila defined=s functions. Removing it would lead to errors.
    # However, this line is not necessary for our implementation and logically,␣
↪removing
    # it would make no difference.
    tq.simulate(tq.gates.Trotterized(generator=g, angle=pi / 2, steps=1))

    circ = tq.QCircuit()
```

```
    projector = tq.paulis.Projector(wfn=wfn_target)

    for step in range(steps):
        for ps in g.paulistrings:
            t = tq.Variable((str(ps), step))
            circ += tq.gates.ExpPauli(paulistring=ps, angle=pi / steps * t)

    # Define objective function to be fidelity, based on the expectation value
    expect = tq.ExpectationValue(H=projector, U=circ)

    result = tq.minimize(1 - expect, initial_values=1.0, silent=True)

    # If satisfactory fidelity not achieved, raise a warning
    fid = 1 - result.energy
    if fid < tolerance:
        warn(f'Target fidelity of {tolerance} not achieved. Fidelity was {fid}',
             LCUFidelityWarning)

    return circ.map_variables(variables=result.variables)
```

3 The LCU class

We wrap up all of our functions in a LCU class which can be initialized by simply passing in the list of ancilla and the unitary decomposition of the operator. Users can then access the various immutable properties of the LCU object, which are listed below:

- ancilla: list of qubits in the ancilla register
- prep_circ: circuit corresponding to prepare operator for LCU circuit
- select_circ: circuit corresponding to select operator in LCU circuit
- lcu_circ: LCU circuit, excluding amplitude amplification procedure
- amp_amp: amplitude amplification operator, signifying one step of amplitude amplification
- full_circ: full LCU circuit, including amplitude amplification procedure

4 Testing functions

We first test the prepare function for the case with only one ancillary qubit works as intended. We do so by first choosing a random positive number $0 < \alpha_0 < 1$. Then, we define $\alpha_1 = 1 - \alpha_0$ to construct our desired linear combination of unitary operations. We designate 0 as the ancillary qubit, and hence the desired wavefunction after the application of the Prepare circuit to $|0\rangle_0$ shall be $\sqrt{\alpha_0}\,|0\rangle_0 + \sqrt{1-\alpha_0}\,|1\rangle_0$.

This is translated as checking whether the fidelity between the resulting state and the target state is close to 1, which can be tested as follows.

[4]:
```
def test_prepare_1anc() -> None:
    """Test whether the prepare operator for 1 qubit ancilla works as intended
    """
    num0 = random()
    num1 = 1 - num0
    unitaries = [(num0, tq.gates.X(1)), (num1, tq.gates.Z(1))]
    prepare = _prepare_1ancilla(0, unitaries)

    wfn_target = tq.QubitWaveFunction.from_array(asarray([sqrt(num0),␣
↪sqrt(num1)]))

    projector = tq.paulis.Projector(wfn=wfn_target)

    expval = tq.ExpectationValue(H=projector, U=prepare)

    p = tq.simulate(expval)

    assert isclose(p, 1, abs_tol=0.001)
```

Next, we test the prepare function for the general case. We use a similar approach to create a random normalized list of coefficients as explained above, and once again, check the fidelity between the target state and the state resulting from the application of the Prepare circuit. This is written in Python as follows.

[5]:
```
def test_prepare_general() -> None:
    """Test whether the general case for the prepare operator works as expected.
↪"""
    # Define random positive length between 1 and 8
    n = 0
    while n == 0:
        n = int(4 * random())

    length = 2 ** n

    # Define ancilla as first 2^n natural numbers
    ancilla = list(range(length))

    # Define coefficients
    coeffs = [0.5 + 0.49 * random() for _ in range(length)]

    unitaries = [(coeff, tq.gates.X(16)) for coeff in coeffs]

    norm = sqrt(sum(coeffs))
    coefficients = [sqrt(coeff) / norm for coeff in coeffs]

    # Define target wavefunction
    wfn_target = tq.QubitWaveFunction.from_array(asarray(coefficients))
```

```
    # Compute prepare circuit
    prepare = prepare_operator(ancilla=ancilla, unitaries=unitaries)

    # Calculate fidelity
    fid = abs(wfn_target.inner(tq.simulate(prepare))) ** 2

    # Fidelity must be close to 1
    assert isclose(fid, 1, abs_tol=0.001)
```

We move on to testing the ampitude amplification functions. We first test the reflection function.

[6]:
```
def test_reflection() -> None:
    """Test whether the reflection operator works as intended."""
    circ = tq.gates.H(0) + reflect_operator(1, 0)
    wfn_target = tq.QubitWaveFunction.from_array(asarray([-1 / sqrt(2), 0, 1 /␣
↪sqrt(2), 0]))

    projector = tq.paulis.Projector(wfn=wfn_target)

    expval = tq.ExpectationValue(H=projector, U=circ)

    p = tq.simulate(expval)

    assert isclose(p, 1, rel_tol=0.001)
```

We next check whether the amplitude amplification works as expected for the stationary angles, i.e. the cases where applying one step of the routine does not change the probability for measuring the ancilla in the $|0\rangle$ state.

[7]:
```
def test_amp_amp_stationary() -> None:
    """Test whether the non-trivial stationary angles for amp_amp_op are as␣
↪expected"""
    # Create uniform superposition of ancilla: qubit 0
    state_prep = tq.gates.H(0)

    # Define target state
    wfn_target = tq.QubitWaveFunction.from_array(asarray([- 1 / sqrt(2), 0, 1 /␣
↪sqrt(2), 0]))

    # Define amplitude amplification circuit
    amp_amp = amp_amp_op(state_prep, 0)
    circ = amp_amp + state_prep

    fid = abs(wfn_target.inner(tq.simulate(circ))) ** 2

    assert isclose(fid, 1.0, abs_tol=0.001)
```

5 Next steps and possible additions

In this section, we look at some limitations and minor changes to be added to the current implementation of the LCU algorithm that we have presented in these three reports.

5.1 Changes to mplitude amplification

The amplitude amplification currently does not respond well to the cases for stationary angles, as explained in Report 1, and will also function well for cases where the operator to be embedded is itself unitary. To resolve the first issue, we simply request the user to input an extra qubit for the case where the function encounters the non-trivial stationary angle. For the second problem, we propose changing the structure of our functions to pass in a list of ancillary lists as an optional parameter to the amplitude amplification functions and use a qubit mapping to change which ancilla is used in each of the steps of the amplitude amplification process.

5.2 Other errors

Currently, our functions assume that the user passes in the correct length of ancilla to handle the number of terms given in the linear combination. However, if we wish to also count for possible mismatches and automatized adjustments to be made, we must define a LCUMismatch custom error to prompt the user to pass in additional requirements, as necessitated.

[11]:
```
def amp_amp_new(unitaries: list[tuple[float, tq.QCircuit]], walk_op: tq.
↪QCircuit, ancilla) \
        -> tq.QCircuit:
    """Return the 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_new(walk_op, ancilla, unitaries)
    s = _num_iter(unitaries)

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

    return sum_of_steps


def amp_amp_op_new(walk_op: tq.QCircuit, ancilla: Union[list[Union[str, int]],␣
↪str, int],
                unitaries: list[tuple[float, tq.QCircuit]], qubit: Optional[Any]␣
↪= None) \
        -> tq.QCircuit:
    """Return WRW.dagger()R,
     where R is the reflect operator returned by the func reflect_operator"""
    s = sum([pair[0] for pair in unitaries])
```

```
    if s == sqrt(2):
        while qubit is not None and qubit not in walk_op.qubits:
            print('Stationary angle encountered for amplitude amplification.')
            qubit = input('Input extra ancilla qubit')

    anc_qubits = ancilla if isinstance(ancilla, list) else [ancilla]
    if qubit is not None:
        anc_qubits.append(qubit)

    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 reflect_operator(state_qubits: Union[list[Union[str, int]], str, int],
                     ancilla: Union[list[Union[str, int]], str, int]) \
        -> 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

    Preconditions:
        - state_qubits and ancilla have no qubits in common
    """
    if isinstance(state_qubits, list) and isinstance(ancilla, list):
        qubits = list(set(state_qubits + ancilla))
    elif isinstance(state_qubits, list) and not isinstance(ancilla, list):
        qubits = list(set(state_qubits + [ancilla]))
    elif not isinstance(state_qubits, list) and isinstance(ancilla, list):
        qubits = list(set([state_qubits] + ancilla))
    else:
        qubits = list(set([state_qubits] + [ancilla]))

    z_gate, cz_gate = tq.gates.Z(target=qubits), tq.gates.Z(control=ancilla,␣
↪target=state_qubits)

    return z_gate + cz_gate
```

6 References

- Kottmann, J., et al. (2021). Tequila: A platform for rapid development of quantum algorithms. arXiv:2011.03057