CSC299 - Report 4 - Applications

September 1, 2021

[1]:
```
import tequila as tq
from typing import Union
from lcu_v1 import LCU
from numpy import pi
```

1 Introduction

In this report, we focus on several practical applications of the LCU algorithm, which we have discussed in Reports 1 through 3. While we will not be delving into the exact details of the code to the extent that was seen in the previous reports, we hope that the use of the LCU class, as introduced in Report 3, shall prove to be a helpful abstraction into implementing the following applications.

As an additional disclaimer, the applications discussed here claim no result for optimality, and there have been multiple algorithms which prove to be more efficient than applying the LCU method. However, with such a wide range of applications, we wish to show proof-of-concept that the LCU algorithm comes with helpful generalizations which can be used to abstract away some of the technicalities involved in defining circuits for certain operations.

2 Motivation

To motivate finding some applications for the LCU method, we first look at some of the properties of this method.

Firstly, note that the success of our functions is guaranteed independent of the initial state when the operator that we are trying to implement is unitary. Otherwise, the success probability is 1 in the best case, but is still dependent on the initial state taken. However, this is not as bad a limitation as it seems like, because many common operations are unitary and this still provides useful insights into their implementation.

We can essentially view the LCU algorithm as a form of block-encoding an operator within another operator that acts on a greater number of qubits. Thus, by doing so, we can generalize the LCU framework to be generally applicable in any context where post-selection is used to define success of the algorithm. Furthermore, in instances of quantum computing where it is feasible to define an operation of the form $\tau : |j\rangle \rightarrow |j\rangle$ or any other bijective operation on the state space, we can even generalize the post-selection for the LCU where success is defined on measuring some particular state $|\psi\rangle = \tau(|0\rangle)$.

We now look at a few examples of this.

3 Application 1: Implementing powers of operators

3.1 Introduction and problem setting

Often, implementing some Hamiltonian, H, for some type of physical systems is not particularly scaleable due to having a large polynomial complexity, or worse. As such, implementing powers of H, say H^k where $k \in \mathbb{N}$, further worsens this complexity. For instance, suppose that implementing H requires a total of N terms where N is some expression in terms of the input; then, implementing H^k will require N^k terms, which scales exponentially with the power required.

For this problem, it suffices to consider implementing H^2 instead of H^k for some general k, since the extension is fairly straightforward.

3.2 Applying LCU

Note that, using the LCU method, we can simply place two of the LCU blocks for H to approximate the effect of H^2.

The success probability of doing so relies on measuring the ancilla for both LCU blocks in the zero state. For the first LCU block, after going through the amplitude amplification routine, will be approximately $||H|\psi\rangle||^2$, where $|\psi\rangle$ was the initial state. Similarly, the success probability for the second LCU block will be $\left|\left|H^2|\psi\rangle\right|\right|^2$.

Combining the two LCU blocks, we see that the success probability of such an implementation of H^2 is $||H|\psi\rangle||^2 \cdot \left|\left|H^2|\psi\rangle\right|\right|^2$.

More generally, with some initial state $|\psi\rangle$, the success probability of such an implementation of H^k, for any $k \in \mathbb{N}$ will be $\prod_{j=1}^{k} \left|\left|H^k|\psi\rangle\right|\right|^2$.

3.2.1 Note on reusing ancilla qubits

For the general case, we cannot use the ancilla qubits in more than one LCU block in the above implementation. This is because we are unable to deterministically improve the success probability of the LCU method to 1, and so, there remains some non-negligible probability that the ancilla is not in the zero-state as required. Therefore, reusing ancilla qubits in such a generalized implementation introduces certain dependencies of the later LCU blocks on the earlier blocks, which worsens our overall success probability.

Hence, to implement H^k, where H is written as a linear combination of m unitary operations, we shall require $\mathcal{O}(\log m)$ ancilla qubits for each LCU block, for a total of k LCU blocks. Thus, the total number of ancilla qubits required for such an implementation is in $\mathcal{O}(k \log m)$.

However, now suppose that H itself is unitary, or is at least 'close' to being unitary in the sense that there exists some unitary U and some small $\epsilon > 0$ such that $||H - U|| < \epsilon$. Then, we are guaranteed success through the oblivious amplitude amplification procedure at the end of each LCU block. This implies that, after measuring, the probability of the ancilla not being in the zero state is very negligible and thus, we can resuse the ancilla qubits repeatedly. Hence, the number of ancilla qubits in such a case is in $\mathcal{O}(\log m)$.

Note that the same analysis holds true for the process of amplitude amplification that was mentioned in Report 1.

3.3 Implementing in code

We use a qubit mapping for the ancilla used for each LCU block as demonstrated below. The advantage of doing this is so that

[4]:
```
def ham_power(ancilla: list[Union[list[Union[str, int]]]],
              unitaries: list[tuple[float, tq.QCircuit]], power: int) -> tq.
↪QCircuit:
    """Return the circuit corresponding to H^k, where H is the Hamiltonian␣
↪corresponding to
    the linear combination expressed in unitaries, and k = power.

    Preconditions:
        - len(ancilla) == power
        - all(2 ** (len(anc) - 1) < len(unitaries) <= 2 ** len(anc) for anc in␣
↪ancilla)
        - ancilla has no repeated qubits
    """
    circ = tq.QCircuit()
    anc0 = ancilla[0]
    lcu = LCU(anc0, unitaries).full_circuit

    for i in range(power):
        anc = ancilla[i]
        qubit_map = {anc0[k]: anc[k] for k in range(len(anc0))}
        lcu = lcu.map_qubits(qubit_map)
        circ += lcu

    return circ
```

3.4 Uses and extensions

The major advantage of using the LCU for this purpose was that the number of terms in implementing powers of H can now scale linearly with the respective power.

One natural extension of this technique is to consider implementing some arbitrary polynomial P of H with real coefficients $\alpha_j \in \mathbb{R}$ and degree $n \in \mathbb{N}$, $P = \sum_{j=0}^{n} \alpha_j H^j$. We have previously seen how to implement each of the H^j using the block-encoding algorithm as a subroutine.

Since we have shown how to compute each of the powers using LCU blocks, instead of naively considering those blocks as separate circuits, we can even include an additional register which determines when to fire off a LCU block for H or when to fire off an empty Tequila QCircuit to be added to the unitary corresponding to the respective power. Such an implementation would greatly reduce the number of ancillary qubits involved in the process, and further aid in automatizing this technique.

Potential further applications of such techniques also include computing expectation values and excited state optimization.

4 Application 2: Time evolution using truncated Taylor series

4.1 Introduction

We have already shown, in application 1, that it is possible to implement any polynomial with real coeffisients in H of finite degree n. If we can generalize this concept to arbitrary

Now, suppose that we wish to simulate the evolution under some Hamiltonian H for time t within a maximum error of some $\epsilon > 0$ with the operator $U = e^{-iHt}$. For this example, we further make the assumption that H is unitary.

Divide the time t into r equal time-segments of length t/r. Further suppose that we can write H as a linear combination $H = \sum_{j=0}^{m} \alpha_j H_j$, where each H_j is unitary and $\alpha_j > 0$ for all j. For each segment, define the operator for time-evolution within that segment as follows:

$$U_r = e^{-iHt/r} \approx \sum_{j=0}^{K} \frac{1}{j!}(-iHt/r)^j$$

Here, we wish to truncate the Taylor series expansion of the exponential at some order K such that the maximum error for each segment is ϵ/r, which would imply that the maximum result across all the segments is ϵ. Assuming that $r > ||H||\, t$ (where $||\cdot||$ denotes the usual matrix norm), it has been shown by Berry, Childs, Cleve, Kothari, Somma (2015) that we must have:

$$K \in \mathcal{O}\left(\frac{\log(r/\epsilon)}{\log(\log(r/\epsilon))}\right)$$

Therefore, we have that:

$$\begin{aligned}
U &= e^{-iHt} \\
&= \prod_{j=0}^{r-1} U_r && (U_r = e^{-iHt/r}) \\
&\approx \prod_{j=0}^{r-1} \sum_{k=0}^{K} \frac{1}{k!}(-iHt/r)^k \\
&= \prod_{j=0}^{r-1} \sum_{k=0}^{K} \frac{(-it/r)^k}{k!} H^k
\end{aligned}$$

Reference used: Simulating Hamiltonian Dynamics with a truncated Taylor Series (BCCKS)

4.2 Applying LCU

We first note that, in our implementation of the LCU algorithm, the only constraint on the operator H was that we assumed H to be Hermitian. This implies that we can think of H as a Hamiltonian, which we shall now do so.

Now, since we already know how to implement H using the LCU algorithm explained previously, this implies that we can also implement all powers of H, H^k, by simply repeating the circuit for H. Furthermore, since the LCU circuit for H was unitary, this implies that all powers of this circuit will also be unitary. This is indeed nice, as it is now similar to our LCU framework.

4.2.1 Adding phases using generators

Notice that, to construct an implementation of U_r, all we need to do is to apply a $(-i)^k$ phase to the circuit for H^k. This can be done in Tequila by using generators, as follows.

First, consider the example of adding a $(-i)$ phase to some H^k. Note that this is equivalent to simply adding a unitary transformation $U = -i = e^{-i\frac{\pi}{2}}$ to the end of the control block for applying H^k, which is just a generator corresponding to angle $\theta = \pi$. Suppose that Gc represents the control block for H^k, i.e. Gc denotes the particular arrangement of controlled-X gates applied to the ancilla so that it fires off the H^k LCU block when desired. Then, we can implement U in Tequila by setting the generators for U to Gc.

[10]:
```
Gc = tq.paulis.Qp(0)
U = tq.gates.Trotterized(generator=Gc, angle=pi)
```

Similarly, we can add a phase of i using $U = i = e^{-i\frac{3\pi}{2}}$ and also a phase of -1 with $U = -1 = e^{-i\frac{2\pi}{2}}$.

4.3 Further extensions

4.3.1 Applications of Hamiltonian simulation

While there exist other algorithms for simulating Hamiltonians which may be more efficient, such as the qubitization method proposed by Low, Chuang (2019), we are now also able to use the LCU class for the same. Thus, this implies that we can now consider

One such prominent example about a potential application is the renowned HHL algorithm for solving linear systems (2009), which relies on Hamiltonian simulation. By extension, we can even use the LCU method to implement algorithms which either use the HHL algorithm as a black-box, or are inspired by the HHL algorithm, such as the quantum algorithm proposed by Leyton, Osborne (2008) to solve nonlinear differential equations using Euler's method.

References used:

- Hamiltonian simulation by Qubitization (LC)
- Quantum algorithm for solving linear systems of equations (HHL)
- A quantum algorithm to solve nonlinear differential equations (LO)

4.3.2 Implementations of arbitrary polynomials with real or complex coefficients

So far, we have looked at constructing polynomials with real coefficients and polynomials with real and purely imaginary coefficients as well. A potential extension of this technique involves constructing polynomials of the form $P = \sum_{j=0}^{n} \alpha_j H^j$ where $\alpha_j \in \mathbb{C}$. To do so, we split any coefficient α_j with $\mathrm{Re}(\alpha_j) \neq 0$ and $\mathrm{Im}(\alpha_j) \neq 0$ into $\alpha_j = \alpha_{j,0} + i\alpha_{j,1}$ with $\alpha_{j,0}, \alpha_{j,1} \in \mathbb{R}$, and proceed with using the LCU method by absorbing phases or adding phases phases as already described.

5 Application 3: Implementing projectors

Projectors are a key example of helpful non-unitary operators that are difficult to realize in practice because of their non-unitary nature. We have already looked at an example in Report 1 of using

the LCU algorithm to encode a projector onto the $|0\rangle$ ancilla qubit. Thus, using the generalized LCU approach, as long as we are able to write the projector as a linear combination of unitaries, for which we can use the paulistrings attribute in Tequila, we are able to construct a projector onto the zero-state efficiently.

Furthermore, as mentioned earlier, if we are able to construct a bijection from the computational basis onto itself, we shall be able to construct a projector onto any arbitrary computational basis state.

6 Application 4: SWAP Test and Hadamard Test

A nice consequence of implementing the LCU algorithm is that we can use it for other purposes as well. A few such small applications are explained as follows.

6.1 SWAP test

6.1.1 Introduction

The SWAP test denotes an algorithm in quantum computing that checks whether two quantum states are the same, or are at least close to each other. It involves applying a Hadamard gate to a single ancillary qubit, and a controlled-SWAP gate with the ancilla as control, followed by another Hadamard on the ancilla and then measurement of the ancilla register. If the two states are close to each other, then the probability that the outcome of the measurement is $|0\rangle$ is close to 1.

Reference used: arXiv:quant-ph/0102001

6.1.2 Applying LCU

Notice that this is almost identical to the LCU framework. Recall that we have previously seen that, in an equal linear combination of two unitary operations, the Prepare operator is equivalent to a one-qubit Hadamard operation. Hence, we can now implement the SWAP test using our LCU class by using the linear combination $H_{\text{SWAP}} = \frac{1}{2}(I + SWAP)$.

6.2 Hadamard test

6.2.1 Introduction

The Hadamard test describes an algorithm in quantum computing that can be used to obtain either the expected real part or the expected imaginary part of the observed value of a quantum state after applying some unitary operator U. For this report, we only demosntrate obtaining the real part using LCU, but note that obtaining the imaginary part follows a very similar procedure.

The algorithm involves applying a Hadamard gate to a single ancillary qubit, and the desired controlled unitary operation, U, with the ancilla as control, followed by another Hadamard on the ancilla and then measurement of the ancilla register. It outputs 1 if the result of the measurement is $|0\rangle$, and -1 otherwise. Doing so, the expectation value of the output is thus the real part of $\langle\psi| U |\psi\rangle$, where $|\psi\rangle$ denotes the initial state of the state register.

Reference used: arXiv:quant-ph/0511096

6.2.2 Applying LCU

Notice that, once again, this is almost identical to the LCU framework. Recall that we have previously seen that, in an equal linear combination of two unitary operations, the Prepare operator is equivalent to a one-qubit Hadamard operation. Hence, we can now implement the SWAP test using our LCU class by using the linear combination $H_{\text{Hadamard}} = \frac{1}{2}(I + U)$.

[3]:
```
# Add 1-qubit SWAP test implementation
def swap_test(ancillary, qubit0, qubit1) -> tq.QCircuit:
    swap_gate = tq.gates.CNOT(target=qubit1, control=qubit0) + tq.gates.
↪CNOT(target=qubit0, control=qubit1)
    swap_test = LCU(ancillary, [(0.5, tq.QCircuit()), (0.5, swap_gate)])
    return swap_test.lcu_circ
```

[4]:
```
# Add Hadamard test implementation
def hadamard_test(ancillary, unitary) -> tq.QCircuit:
    hadamard_test = LCU(ancillary, [(0.5, tq.QCircuit()), (0.5, unitary)])
    return hadamard_test.lcu_circ
```

7 Application 5: Implementing measurement reduction using LCU

This section is based on the paper by Ralli, Love, Tranter, Coveney (2021) which describes how to use the LCU algorithm

7.1 Introduction

Consider a d-dimensional qubit Hamilonian H over N qubits which is expressed as a sum of k Pauli-strings, as follows:

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

Here, the c_j are some numerical constants and the P_j refer to Pauli-strings, i.e. some products of the Pauli operators, $P_j = \prod_{i=0}^{N-1} \sigma_i^{(j)}$. As shown in the paper, we can rewrite H as:

$$H = \sum_{n=0}^{\ell-1} A_n$$

Here, the A_n denote the "commuting cliques" in the expansion of H as a sum of Pauli-strings. In other words, any Pauli-string in a particular commutes with all other Pauli-string in that same

clique. If C_n denotes the n-th clique, corresponding to A_n, we can express this as follows.

$$\begin{aligned} A_n &= \sum_{j \in C_n} c_j P_j \\ \{P_i, P_j\} &= 2\delta_{i,j} I \qquad (\forall i, j) \\ \sum_{j=0}^{\ell-1} |c_j|^2 &= 1 \\ \mathrm{Im}(c_j^\star c_i) &= 0 \end{aligned}$$

Here, $\delta_{i,j}$ represents the Kronecker delta function.

Note that while this is, in general, a NP-hard problem, we are still able to approximate such cliques efficiently even though we cannot find exact groupings. We renormalize and redefine the A_n such that:

$$H = \sum_{n=1}^{\ell} \gamma_n A_n$$

To remain consistent with the RLTC paper, we introduce the same notation as the one used in the paper. Define S_n to denote the set of Pauli-strings in the n-th clique. Next, define sub-Hamiltonians H_{S_n} where n ranges from 0 through ℓ.

$$H_{S_n} = \gamma_n \sum_{P_j \in S_n} \beta_j^{(l)} P_j^{(l)}$$

Thus, so far, we have defined the following:

$$\begin{aligned} H &= \sum_{j=0}^{k-1} c_j P_j \\ &= \sum_{l=0}^{\ell-1} H_{S_l} \\ &= \sum_{l=0}^{\ell-1} \gamma_n \sum_{P_j \in S_n} \beta_j^{(l)} P_j^{(l)} \\ &= \sum_{l=0}^{\ell-1} \gamma_n A_l \end{aligned}$$

Now, since each of the A_l is Hermitian, there exists some orthonormal eigen-basis for A_l, $(|\psi_a\rangle)_{a=0,\ldots,d-1}$. Furthermore, there exists some unitary operation R_l (since all unitary operations are also isometries) such that R maps this eigen-basis to the standard computational basis $(|e_a\rangle)_{a=0,\ldots,d-1}$, i.e. $R_l(|\psi_a\rangle) = |e_a\rangle$. Thus, define Q_l such that:

$$Q_l = \sum_{a=0}^{d-1} \lambda_a \left|e_a\right\rangle \left\langle e_a\right|$$

We now have the following:

$$\begin{aligned} A_l &= \sum_{a=0}^{d-1} \lambda_a \left|\psi_a\right\rangle \left\langle\psi_a\right| \\ &= \sum_{a=0}^{d-1} \lambda_a R_l^\dagger \left|e_a\right\rangle \left\langle e_a\right| R_l \\ &= R_l^\dagger \left(\sum_{a=0}^{d-1} \lambda_a \left|e_a\right\rangle \left\langle e_a\right| \right) R \\ &= R_l^\dagger Q_l R_l \end{aligned}$$

The expectation value of H can thus be written as

$$\begin{aligned} \left\langle\psi\right| H \left|\psi\right\rangle &= \sum_{l=0}^{\ell-1} \gamma_l \left\langle\psi\right| A_l \left|\psi\right\rangle \\ &= \sum_{l=0}^{\ell-1} \gamma_l \left\langle\psi\right| R_l^\dagger Q_l R_l \left|\psi\right\rangle \end{aligned}$$

Using this expression for the expectation value, we only need to estimate ℓ terms if we are able to efficiently implement each R_l.

7.2 Applying LCU

We now aim to implement each of the R_l's in the above expression using the LCU method. We first need to manipulate each of the ℓ anticommuting sets that H was partitioned into.

Choose some Pauli-string $P_j = P_\omega^{(l)} \in S_l$ to be reduced to. Next, define the operator $H_{S_l \ \{P_\omega^{(l)}\}}$ such that

$$\begin{aligned} H_{S_l \ \{P_\omega^{(l)}\}} &= \sum_{k \in C_n, k \neq \omega} d_{k,l} P_k \\ \sum_{k \in C_n, k \neq \omega} d_{k,l}^2 &= 1 \end{aligned}$$

Therefore, we now have:

$$\frac{1}{\gamma_l} H_{S_l} = \beta_\omega^{(l)} P_\omega^{(l)} + \Omega_l H_{S_l \ \{P_\omega^{(l)}\}}$$

where $\beta_j^{(l)} = \Omega_l d_{j,l}$ and ${\beta_\omega^{(l)}}^2 + \Omega_l^2 = 1$. We can define the angle $\phi_\omega^{(l)}$ such that $\cos\left(\phi_\omega^{(l)}\right) = \beta_\omega^{(l)}$ in order to rewrite the above equation as follows:

$$\begin{aligned} H_\omega^{(l)} &= \frac{1}{\gamma_l} H_{S_l} \\ &= \cos\left(\phi_\omega^{(l)}\right) P_\omega^{(l)} + \sin\left(\phi_\omega^{(l)}\right) H_{S_l \ \{P_\omega^{(l)}\}} \end{aligned}$$

Now, we define the generator $\chi^{(l)}$ as follows, where $[\cdot]$ denotes the anti-commutator function and we define $P_{k\omega}^{(l)} = P_k^{(l)} P_\omega^{(l)}$:

$$\begin{aligned}\chi^{(l)} &= \frac{i}{2}[H_{S_l\ \{P_\omega^{(l)}\}}, P_\omega^{(l)}] \\ &= i \sum_{k \in C_l, k \neq \omega} d_{k,l} P_{k\omega}^{(l)}\end{aligned}$$

Next, define the rotation R_l as:

$$\begin{aligned}R_l &= e^{-i\frac{\alpha^{(l)}}{2}\chi^{(l)}} \\ &= \cos\left(\frac{\alpha^l}{2}\right) I + \sin\left(\frac{\alpha^l}{2}\right) \sum_{k \in C_l, k \neq \omega} d_{k,l} P_{k\omega}^{(l)}\end{aligned}$$

We have now written R_l as a linear combination of unitary operations and for the ease of practicality, we can even simplify this expression further by noting that all of the $P_{k\omega}$'s and I are in the Pauli group. Thus, we can make use of the LCU class and proceed with implementing R_l.

Reference used: Implementation of Measurement Reduction for the Variational Quantum Eigensolver (RLTC)

7.3 Next steps

We can move forward towards an extension of this project which aims to implement the algorithm described in this section in Tequila using the LCU class we defined in Report 3. However, the primary challenge in this case is the fact that a naive implementation (i.e. a direct translation of this algorithm into code) will certainly prove to be difficult to automatize, and so we must look for alternative approach to translate this method into Python.

Further complications arise in defining an optimal heuristic, if such an optimal result can be achieved, for choosing which Pauli-string to reduce to in each of the commuting cliques.

8 References

- Low, G. H., Chuang, I. L., (2019). Hamiltonian Simulation by Qubitization. arXiv:1610.06546
- Leyton, S. K., Osborne, T. J. (2008). A quantum algorithm to solve nonlinear differential equations. arXiv:0812.4423
- Buhrman, H., Cleve, R., Watrous, J., de Wolf, R. (2001). Quantum Fingerprinting. Physical Review Letters. 87 (16). arXiv:quant-ph/0102001
- Aharonov, D., Jones, V., Landau, Z. (2006). A Polynomial Quantum Algorithm for Approximating the Jones Polynomial. arXiv:quant-ph/0511096
- Ralli, A., Love, P., Tranter, A., Coveney, P. (2021). Implementation of Measurement Reduction for the Variational Quantum Eigensolver. arXiv:2012.02765