CSC299 - Report 5 - Alternate Implementations

September 1, 2021

```
[2]: import tequila as tq
```

1 Introduction

In Reports 1 through 3, we used certain techniques to implement the block-encoding of the LCU algorithm in Python. For example, in reports 2 and 3, we used a variational approach to construct the appropriate circuit for the Prepare operator, and we used a binary encoding to construct the Select operator, and so on.

This report focuses on alternate implementatiosn for this algorithm. Much of the content in this report is based on analyzing the drawbacks of the way in which the current implementation works, and comparing it with potential improvements resulting from other options of implementation. For instance, we look at exchanging the binary encoding of the ancilla registers in the original implementation with a unary embedding, and we also look at how we could have used non-variational algorithms for the necessary state preparation and the resulting tradeoffs.

2 Potential speed-up in current approach

In this section, we look at certain ways in which we could have improved upon the current implementation of the Prepare operator, in the function prepare_operator from Report 3. All the suggestions in this section still revolve around using variational algorithms, and offer a speed-up as compared to the current implementation in the average case; this implies that, on average, the functions would be faster, but there is no guarantee of a speed-up in any particular example.

2.1 Commuting cliques in Trotterization

The reason we consider this feature is because when Pauli-strings commute, we can reduce the number of steps involved in the optimization. The process of doing so is explained in detail in the paper by Verteletskyi, Yen, Izmaylov (2020) as follows. Consider a qubit Hamilonian H which is expressed as follows:

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

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

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

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.

$$A_n = \sum_{j \in C_n} c_j P_j$$
$$[P_i, P_j] = 0 \qquad (\forall i, j \in C_n)$$

In the original paper, expressing H in such a form was used to be able to measure all Pauli-string in A_n using only one set of single-qubit measurements, and thus removing the need for multi-qubit measurements. However, we are not concerned with optimizing the measurement of these groupings, but rather, we use it to reduce the number of groupings we have to consider in the Trotterization step.

Reference used: Measuring all compatible operators in one series of single-qubit measurements using unitary transformations (VYI)

2.1.1 Implementing in code

This method of optimizing measurements has already been implemented in Tequila, and can be used by setting the parameter of "optimize_measurements" in the ExpectationValue method to True. This is carried forth after we have defined our generator in the current implementation of the function "prepare_operator", as shown in the code below.

The variable expval will then be a collection of expectation values for the different cliques found, instead of being a single expectation value as seen earlier. We can then count the number of cliques found in the generator by simply counting the number of expectation values stored in expval.

2.1.2 Uses

By considering each clique as a single block, we can reduce the number of blocks in the Trotterization, by reducing the number of possibilities of arranging the blocks in the Trotter expansion. However, we also need to compute small basis transformations which ensure that each of the blocks are diagonalized.

[3]:
```python
# The following code is taken from the body of the function prepare_operator.
def prepare_operator(ancilla, unitaries, steps):
    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()

    # Define zero state

    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)

    # Use measurement optimization to find commuting cliques
    expval = tq.ExpectationValue(H=g, U=tq.QCircuit(),␣
↪optimize_measurements=True)
    commuting_groups = expval.count_expectationvalues()

    # The rest of the code in the function is unchanged.

    # Don't remove this line! It's required!!
    # assert g.is_hermitian()

    tq.simulate(tq.gates.Trotterized(generator=g, angle=pi / 2, steps=1))

    # Initialize empty circuit
    circ = tq.QCircuit()

    # Define projector to measure fidelity
    projector = tq.paulis.Projector(wfn=wfn_target)

    for step in range(steps):
        for i, x in enumerate(expval.get_expectationvalues()):
            g = x.H[0]
            circ += x.U
            circ += tq.gates.Trotterized(generator=g,
                                         angle=pi / steps * tq.
↪Variable(name=("t", i, step)),
                                         steps=1)
            circ += x.U.dagger()
    # construct an operator that represents the fidelity object
    expect = tq.ExpectationValue(H=projector, U=circ)

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

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

2.1.3 Testing for speed-up

As is the case with such changes, we must test whether implementing this feature indeed results in any change in the runtime of our function. Now, since it is often complicated to construct a running-time analysis for variational quantum algorithms, such as the minimization function we used as a submodule, we shall make do with simply observing the time taken to return a circuit instead. We can use the same approach for this as we did for testing the prepare operator in Report 3.

In other words, construct a list of ancilla of random length and then construct a random linear combination of unitaries with an appropriate number of terms. Then, run the two different functions for creating the Prepare operator, with the debug parameter set to True, and compare the running times output by Python. Repeat for a large sample of such randomized cases, and analyze the difference in the time taken to complete running the functions, if any.

2.2 Approximations to gradient computation and other parameters in minimization

We currently use the default settings for the additional arguments to the minimization subroutine used in constructing the Prepare circuit, which involves analytic gradients for the gradient descent algorithm. However, we can improve upon the running time by instead using finite-difference gradients, to approximate the gradient to a satisfactory precision. While this is not necessitated for our purposes, this tool is indeed rather valuable for cases where exact computation of the gradient proves tedious.

We can also change our preference of the optimizer used, and we leave it up to the user to decide their preference, with the default being set to scipy. For instance, the bfgs gradient-based optimizer is often chosen when one expects fast convergence to a minimum (or maximum) for a smooth function. However, this optimizer relies upon information obtained from all previous computed gradients to approximate the Hessian of the function at the current trial point, so it is rather infeasible for cases where one expects the convergence towards the minimum to be a fairly long process. While that is not the case for our purposes, when faced with those cases, the user can instead opt to use the lbfgs optimizer instead, which only stores local information.

We reiterate that, since every objective in Tequila is automatically differentiable, the user can simply choose to use the default options and treat the optimization subroutine as a black-box instead. The primary advantage of such an implementation is that it leaves the freedom of choice to the user instead of predetermining what combination of settings would be best for a given scenario. Another key advantage of this design is that since optimization and measurement optimization are being continually improved upon, if new optimizers are supported by Tequila, then the users are free to use those options as well.

While we have not thoroughly tested which combination of settings provide the quickest results, the process of such testing will be the same as outlined above.

3 Non-variational approach to Prepare operator

In this section, we look at alternative implementations of the Prepare operator which do not use a variational approach, The advantage of such implementations lies in the fact that variational algorithms are difficult to analyze.

With a non-variational approach, we hope that we can arrive at a program that still satisfies all our postconditions with an acceptable level of error, while also allowing for relatively easier running time analysis. This will allow us to identify parts of the code which can then be further optimized, to improve efficiency.

3.1 Black-box quantum state preparation

We first look at some recent work by Sanders, Low, Scherer and Berry (2020) that focuses on the problem of state preparation. The main goal of the algorithm presented in their paper is to almost identical to the goal of implementing the Prepare operator for our case.

The paper we refer to considers how to construct a state with a linear relation of the coefficients of the target state with the target input vector, as well as a square-root relation. Furthermore, they extend their algorithm to also account for the possibility of complex numbers as elements of the target input vector.

However, for our problem, we are only concerned with the case of linear relation with the square-root of the coefficients with only real (and positive) values, so that simplifies the work we have to do. Note that this is not the same as the case of a square-root relation with the coefficients because we have chosen a different normalization as compared to the one in the paper.

We start with a list of coefficients $\alpha = (\alpha_0, \ldots, \alpha_{m-1})$ obtained from the linear combination of m unitaries that was given to us. We append zeroes to the end of α, if necessary, until the length of α is a perfect power of 2. Suppose at this point that we have $\alpha = (\alpha_0, \ldots, \alpha_{d-1})$ and $d = \dim(\alpha)$, i.e. we think of α as a d-dimensional vector. Now, define $\beta = (\beta_0, \ldots, \beta_{d-1}) = (\sqrt{\alpha_0}, \ldots, \sqrt{\alpha_{d-1}})$.

Further suppose that we are able to access a quantum oracle $\mathfrak{amp}$ that is defined as follows. Let z be an n-bit integer encoded into an n-bit register; let $\oplus$ represent a bitwise-XOR operation; and let $\beta_j^{(n)} = \lfloor 2^n \beta_j \rfloor$. We wish to prepare the state:

$$|\psi_{\text{target}}\rangle = \frac{1}{||\beta||_2} \sum_{j=0}^{d-1} \beta_j |j\rangle$$

Next, suppose that we have a function $\mathfrak{comp}$ which takes as input two n-bit registers storing n-bit integers a and b and one ancillary qubit, defined as follows:

$$\text{comp}\,|a\rangle\,|b\rangle\,|0\rangle = \begin{cases} |a\rangle\,|b\rangle\,|0\rangle\,, & a < b \\ |a\rangle\,|b\rangle\,|1\rangle\,, & a \geq b \end{cases}$$

To begin the algorithm, start with the $\lceil \log_2(m) \rceil$ main ancilla register, which we shall call $\wr\sqcap\sqcup\sqrt{\sqcap\sqcup}$ in the $|0\rangle$ state, and then prepare the register in a uniform superposition of all computational basis states. Next, use the $\mathfrak{amp}$ oracle to write the target amplitudes into a new n-qubit register, which we call ⌈⊣⊔⊣.

Now, using $\mathfrak{comp}$, compare the value of the output after applying $\mathfrak{amp}$ with a superposition of all possible outputs prepared in a new n-qubit register, which we shall call ∇⌉{, with the result written to a single-qubit register which is denoted by {↕⊣}. The resulting state after applying $\mathfrak{comp}$ is denoted $|\psi_{\text{comp}}\rangle$, where $|\psi_{\text{comp}}\rangle$ is defined as:

$$|\psi_{\text{comp}}\rangle = \frac{1}{\sqrt{2^n d}} \sum_{l=0}^{d-1} |l\rangle_{\texttt{out}} \left|\alpha_l^{(n)}\right\rangle_{\texttt{data}} \otimes \left(\sum_{x=0}^{\alpha_l^{(n)}-1} |x\rangle_{\texttt{out}} |0\rangle_{\texttt{flag}} + \sum_{x=\alpha_l^{(n)}}^{2^n-1} |x\rangle_{\texttt{out}} |1\rangle_{\texttt{flag}} \right)$$

Next, apply a Hadamard transform to the register ∇⌉{ to unprepare the uniform superposition to obtain the following state $\left|\psi_{\text{ready}}\right\rangle$, where $|\omega\rangle$ denotes some unnormalized state supported in the orthogonal subspace to the 1D subspace with basis $|0\rangle_{\texttt{ref}}^{\otimes n} |0\rangle_{\texttt{flag}}$.

$$\left|\psi_{\text{ready}}\right\rangle = \frac{1}{2^n\sqrt{d}} \sum_{l=0}^{d-1} \alpha_l^{(n)} |l\rangle_{\texttt{out}} \left|\alpha_l^{(n)}\right\rangle_{\texttt{data}} |0\rangle_{\texttt{ref}}^{\otimes n} |0\rangle_{\texttt{flag}} + |\omega\rangle_{\texttt{out}\otimes\texttt{data}\otimes\texttt{ref}\otimes\texttt{flag}}$$

The state preparation has succeeded when the ∇⌉{ and {↕⊣} registers are measured in the zero state. We can deterministically increase the success probability to close to 1 by applying amplitude amplification. However, to minimize the additional cost of the amplification procedure, we can include it at the end of the LCU algorithm, along with the standard amplitude amplification that we currently have implemented.

We can reset and discard the ⌈⊣⊔⊣ register at the end of the amplitude amplification procedure by simply applying the $\mathfrak{amp}$ oracle once again. This allows for reusing the qubits in the ⌈⊣⊔⊣ register in subsequent LCU blocks.

Note that there are continually new improvements being made in the field of state preparation and so, there have been multiple advancements that have been made in this field since this particular algorithm was first discovered, such as the paper by Yutaro Iiyama (2020). In essence, we are free to choose whichever state preparation algorithm we want, of which there are plenty. Hence, another potential extension of this project would be to allow the user to choose which implementation of the prepare function they prefer, variational or otherwise.

Reference used:

- Black-box quantum state preparation without arithmetic (SLSB)
- Quantum state preparation with multiplicative amplitude transduction (Iiyama)

3.2 Two-level universal quantum gates

Next, we take a look at some methods that were presented in the textbook "Quantum Computation and Quantum Information" by Michael Nielsen and Isaac Chuang (commonly called 'Mike and Ike'). These methods rely on preparing a set of universal quantum gates that are used to construct any arbitrary quantum circuit.

Recall that a (finite) set, S, of quantum gates is said to be universal when one can use only gates from S to compute any arbitrary quantum function. To achieve universality, we must hence be able to construct any arbitrary unitary quantum function $f : \mathcal{H} \to \mathcal{H}$, where $\mathcal{H}$ is the respective Hilbert space, to arbitrary accuracy using only a finite number of gates from the set S. Nielsen and Chuang present three constructions to prove universality in quantum computation, but we shall only focus on the first such construction in this report.

For the purposes of this section, we define the matrix of a quantum circuit, or a quantum operator, as the matrix representation of the operator in the standard computational basis. In other words, when we refer to the operator U as a matrix, we are implicitly referring to its matrix representation in the computational basis.

First, consider some arbitrary 3×3 unitary matrix J with complex coefficients, which is expressed as follows:

$$J = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & j \end{bmatrix}$$

We wish to find two-level unitary matrices U_0, U_1, U_2 such that $U_2 U_1 U_0 J = I$, which would imply that $J = U_2^\dagger U_1^\dagger U_0^\dagger$.

Suppose that $b = 0$. In this case, we define U_0 as $U_0 = I$. Else, if $b \neq 0$, then we define U_0 as

$$U_0 = \begin{bmatrix} \frac{a^\star}{\sqrt{|a|^2+|b|^2}} & \frac{b^\star}{\sqrt{|a|^2+|b|^2}} & 0 \\ \frac{b}{\sqrt{|a|^2+|b|^2}} & \frac{-a}{\sqrt{|a|^2+|b|^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Left-multiplying with J, we have that:

$$U_0 J = \begin{bmatrix} a' & d' & g' \\ 0 & e' & h' \\ c' & f' & j' \end{bmatrix}$$

Similarly, we define U_1 in such a way that left-multiplying with $U_0 J$, we will have that the $(3, 1)$ element of the resulting matrix is also zero. Much like what we did with U_0, first suppose that $c' = 0$; in this case, we define:

$$U_1 = \begin{bmatrix} a'^\star & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Else, in case $c' \neq 0$, we define:

$$U_1 = \begin{bmatrix} \frac{a'^\star}{\sqrt{|a'|^2+|c'|^2}} & 0 & \frac{c'^\star}{\sqrt{|a'|^2+|c'|^2}} \\ 0 & 1 & 0 \\ \frac{c'}{\sqrt{|a'|^2+|c'|^2}} & 0 & \frac{-a'}{\sqrt{|a'|^2+|c'|^2}} \end{bmatrix}$$

Indeed, left-multiplying with $U_0 J$, we have:

$$U_1 U_0 J = \begin{bmatrix} a'' & d'' & g'' \\ 0 & e'' & h'' \\ 0 & f'' & j'' \end{bmatrix}$$

Since U_1, U_0, and J are unitary, we know that U_1U_0J is also unitary and hence $|U_1U_0J| = 1$, which implies that $d'' = g'' = 0$. Next, we define U_2 as:

$$U_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & e''^{\star} & h''^{\star} \\ 0 & f''^{\star} & j''^{\star} \end{bmatrix}$$

Left-multiplying with U_1U_0J, we see that $U_2U_1U_0J = I$, and hence, $J = U_0U_1U_2$.

It is then straightforward to extend this argument to any matrix U larger dimensions using the method of block-matrices, i.e. by partitioning U into smaller 3×3 and 2×2 sub-matrices inside U. Now, suppose that U denotes the matrix of the Prepare operator and let d be the dimension of the space U acts on. Thus, this implies that only the first row of U is relevant for our purposes. By the argument above, we can find two-level unitary matrices $U_{d-1}, U_{d-2}, \ldots, U_0$ such that

$$\left(\prod_{j=0}^{d-1} U_{d-1-j} \right) U = I$$

Hence, the problem of constructing the Prepare operator reduces to a problem of finding these d two-level unitary operations instead.

Reference: Quantum Computation and Quantum Information, by Nielsen and Chuang

4 Different encoding of qubits

In this section, we look at how using different types of encoding for the qubits could affect the efficiency of our program. In particular, we focus at improving the Select operator and seek to find a way which could deterministically reduce the number of gates and/or the number of qubits required for the ancillary register.

4.1 Analysis of current implementation

Firstly, we shall take a look at how our current implementation works. Recall how we used a conventional binary encoding while implementing the Select operator and how we used X gates before and after each application of a controlled unitary application (with the ancilla as controls) as needed. In the best case, we did not need to apply any X gate to the particular qubit since the unitary was supposed to be active if that qubit was 1. Similarly, in the worse case, we needed to apply two X gates if the unitary was meant to be active with the qubit being in 0 state.

It is easy to see that such an approach requires us to have $m = \log_2(k) \in \mathcal{O}(\log(k))$ qubits in the ancilla register if we are given a total of k unitary operations in the linear combination.

For the simplicity of analysis, suppose that we had $k_m = 2^m$ unitaries to consider in implementing this operation, where m denotes the number of ancilla qubits. It is fairly straightforward to extend this analysis to an arbitrary number of unitaries. Denote by $T(k_m)$ the number of X gates used in implementing the controlled version of all of the unitaries. For $m = 1$, we have that $T(k_1) = 2$. We also notice that, for $m > 1$, we have that:

$$\begin{aligned} T(k_m) &= T(k_{m-1}) + 2(m-1) + T(k_{m-1}) \\ &= 2(T(k_{m-1}) + m - 1) \end{aligned}$$

By using the Master Theorem of solving recurrences, we conclude that $T(k_{m-1}) \in \mathcal{O}(m \log(m))$. We note that this does not seem an ideal result, which raises the question: can we do better? We attempt to find out in this section.

4.2 Unary encoding

A natural response might be to simply switch over to a unary encoding of the ancillary qubits instead. This would entail using one qubit to correspond to one unitary operator in the given linear combination of k unitaries. As such, we would only need $\mathcal{O}(1)$ X gates in this step which is a large improvement. However, this comes at an increased cost of needing $m = k \in \mathcal{O}(k)$ ancilla qubits, which is not particularly desirable, especially with the limitations of current devices.

4.3 Gray-code encoding

We then look at using a gray-code encoding for the operation we are trying to implement. Recall that the gray-code system, which is sometimes referred to as the reflected binary code, is a well-defined ordering of the binary system in a way such that every two successive numbers differ in exactly one location.

Note that, since this is still a type of binary encoding, the number of qubits needed in the ancilla register remains unchanged from the conventional binary encoding we have currently used.

Furthermore, supposing that each of the unitaries is mapped to some binary value in gray-code, this implies that we need a maximum of 2 X gates between each controlled unitary operation. Hence, the total number of X gates needed with this encoding is in $\mathcal{O}(m)$, which is an appreciable speed-up without any apparent disadvantages.

Thus, this demonstrates a feasible optimization of our code assuming that we can create a sub-module to automatize the construction of the gray-code encoding.

5 References

- Verteletskyi, V., Yen, T., Izmaylov, A. F. (2020). Measurement Optimization in the Variational Quantum Eigensolver Using a Minimum Clique Cover. arXiv:1907.03358
- Sanders, Y. R., Low, G. H., Scherer, A., Berry, D. W. (2019). Black-box quantum state preparation without arithmetic. arXiv:1807.03206
- Iiyama, Y. (2020). Quantum state preparation with multiplicative amplitude transduction. arXiv:2006.00975
- Nielsen, M. A., Chuang, I. L. (2010). Universal quantum gates. In Quantum computation and quantum information (pp. 188–194)., Cambridge University Press.