Quantum computing TP
Phase estimation for quantum chemistry

Supervisor: Bertrand Marchand

December 5, 2022

Abstract

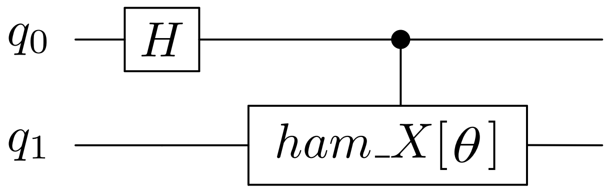
You will have to write code by following the questions in this document. The code will be both written in the notebook (for Question 1) and in the file `tp_library_pea.py` (for the other questions). For each question, the notebook **launches some tests** using your implementation. It is highly recommended to do the questions in order, and to only move forward if all tests so far are correct.

The purpose of this TP is to plot the "dissociation curve" of H_2, for various values of the Trotterization parameter p, which is done at the end of the notebook.

Both `tp_library_pea.py` and the completed notebook **need to be sent by mail before next Thursday, December** 15^{th} **2022, 23h59**, at bertrand.marchand@lix.polytechnique.fr. They will be corrected and graded.

1 Quantum Programming Basics

Question 1 Implement the following circuit, simulate it for $\theta = 0.3$, and print all states along with their amplitudes and probabilities.



Where $ham\_X(\theta) = e^{-i\theta X}$ with $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Hint Define *ham_X* as an "AbstractGate", then apply it using ".ctrl()" to get a controlled version, as suggested in the "minimal notebook". Don't hesitate to also refer to the slides that were sent to you, for instance for the definition of what a controlled gate is (slide 2).

2 Iterative quantum phase estimation, reproducing results from [1]

2.1 Hamiltonian simulation

The Hamiltonians whose ground state energy we are trying to compute are of the form:

$$H(R) = g_0 I + g_1 ZI + g_2 IZ + g_3 ZZ + g_4 YY + g_5 XX$$

Because we use **Trotterization**, we will only have to implement the unitary evolutions generated by each of the individual terms: e^{-iXXdt}, e^{-iZIdt}, e^{-iIZdt}, e^{-iZZdt}, e^{-iYYdt} and e^{-iXXdt}.

We could implement them using Abstract gates and a numerical computation of the matrix exponential. But the computation of the matrix exponential does not scale well with the number of qubits, and thus this method cannot be used in the "general case", when working with a large molecule.

What we will do therefore instead is find small parametrized circuits implementing each of the unitary evolutions. The small circuits may only use "usual gates", such as CNOTs and single-qubit Pauli rotations, that are already pre-implemented in PyAQASM/myqlm. We will then use **QRoutines** to paste these circuits into the final algorithm. Such a method does yield circuits of polynomial size for arbitrary Pauli products over n qubits.

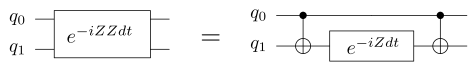
Example: You can check that: $Z \otimes Z = CNOT_{0\to 1} \cdot I \otimes Z \cdot CNOT_{0\to 1}$

with $CNOT_{0\to 1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$, $\otimes$ the Kronecker product and $\cdot$ the standard matrix product.

It allows us to write:

$$\begin{aligned} e^{-iZZdt} &= e^{-i(CNOT_{0\to 1})\cdot(IZ)\cdot(CNOT_{0\to 1})dt} \\ &= CNOT_{0\to 1} \cdot e^{-iIZdt} CNOT_{0\to 1} \\ &= CNOT_{0\to 1} \cdot \left(I \otimes e^{-iZdt}\right) \cdot CNOT_{0\to 1} \end{aligned}$$

i.e:



where $e^{-iZdt} = \begin{pmatrix} e^{-idt} & 0 \\ 0 & e^{idt} \end{pmatrix} = RZ(2 * dt)$.

RZ is a "standard gate", pre-implemented in PyAQASM, defined as $RZ(\theta) = e^{-i\frac{\theta Z}{2}}$. Likewise, $RX(\theta) = e^{-i\frac{\theta X}{2}}$ and $RY(\theta) = e^{-i\frac{\theta Y}{2}}$ are pre-implemented. You can look at their precise definitions here.

Question 2 Following the example above, which is already implemented in the `tp_library_pea.py`, and knowing in addition that:

$$X \otimes X = CNOT_{0\to 1} \cdot X \otimes I \cdot CNOT_{0\to 1}$$

$$SXS^\dagger = Y$$

with $S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
and

$$Y \otimes X = CNOT_{0\to 1} \cdot Y \otimes I \cdot CNOT_{0\to 1}$$

write down the QRoutines carrying out each of the required Hamiltonian simulations, by filling the templates already written in the file.

Test your code by executing the notebook cell containing `''tp_question2_tests.py''`

2.2 Iterative Phase estimation

2.2.1 Trotterization

Question 3 Write a function taking as input hamiltonian coefficients, an interval of time dt, and a Trotter number p, and returning a Qroutine which implements the corresponding Trotterized implementation of the Hamiltonian.

We recall that Trotterization consists in approximating $e^{-idt\left(\sum_l c_l H_l\right)}$ as $\left(\prod_l e^{-i\frac{dt c_l H_l}{p}}\right)^p$. Therefore, your Qroutine should make use of the other Qroutines you implemented in Question 2. As for the coefficients weighting the influence of each individual terms, they are available in the list of dictionaries loaded at the beginning of the notebook, from hamiltonian_data.json. As specified in the signature of the function, your function should also take an "energy shift" as input parameter. It consists in adding a $+shift \cdot I$ term to your Hamiltonian. We will use it to ensure the energies we compute are > 0. Take a look at the slides (slide 10) for more explanations.

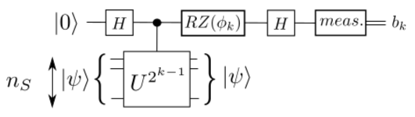
To be more explicit, the QRoutine should implement:

$$dt, p \to \left[e^{-i\frac{g_0 dt}{p}I} e^{-i\frac{g_1 dt}{p}ZI} e^{-i\frac{g_2 dt}{p}IZ} e^{-i\frac{g_3 dt}{p}ZZ} e^{-i\frac{g_4 dt}{p}YY} e^{-i\frac{g_5 dt}{p}XX}\right]^p$$

We recall that: $H = g_0 I + g_1 ZI + g_2 IZ + g_3 ZZ + g_4 YY + g_5 XX$

Test your code by executing the notebook cell containing `''tp_question3_tests.py''`

```
for k in range($n_B$, 0, -1)[3]:
    1. compute $\phi_k = 2\pi \cdot \sum_{l=k+1}^{n_B} \frac{b_l}{2^{l-k+1}}$
    2. produce and execute the following circuit:
```



```
    which yields $b_k$.
```

Figure 1: The pseudocode and circuit to implement at **Question 4**

2.2.2 Iterative Phase estimation

Question 4.1 As one can see on Figure 1, a crucial part of iterative phase estimation is the "adaptive" computation of ϕ_k, whose value depends on measurement results obtained in previous rounds.

Given the criticality of the correct computation of ϕ_k, its implementation has been separated into its own function, `compute_phi_k` that takes as input `bits`, `nBits` and `k`. `bits` is a dictionary, initialized and filled in the `phase` function later down the notebook (in the `phase` function). It associates integers ℓ to the result b_ℓ of round ℓ of the iterative phase estimation procedure.

Test your code by executing the notebook cell containing `‘‘tp_question4_tests.py’’`

Question 4.2 Complete the implementation of iterative phase estimation provided in the notebook, the definitions given in the slides (slide 6) and Figure 1.

In a few words, the function loops over $k = n_B, ..., 1$ in that order, and for each of these values:

- Produce the circuit displayed on the slides (slide 6) and Figure 1. Depending on the value of the "trotterization" Boolean flag, the Hamiltonian simulation can either be the ideal "cheat mode" version that exponentiates the Hamiltonian, or the Trotterized version you implemented. In place of "$|\psi\rangle$", the ground state of the Hamiltonian, we will use a simple ansatz, the Hartree-Fock approximation to our problem: $|10\rangle = (X \otimes I)|00\rangle$, as suggested in [1], and explained in the slides (slide 10).
- simulate the circuit, specifying at job creation that you are interested in the first qubit only, and select the most probable result[1].

[1]Of course in a real setting we would have to run the circuit many times to estimate the probabilities, here, we compute them numerically.

2.3 final plots

The cells actually computing the curves are already written. The "ideal" result should be a smooth H_2 dissociation curve, while the Trotterized version should be more "wiggly", the curve with $p = 10$ being a closer approximation to the ideal curve, as expected.

A tensor products, matrix exponential and Pauli matrices

A.1 Pauli matrices

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

A.2 tensor products

$$(A \otimes B) \cdot (C \otimes D) = AC \otimes BD$$

$$I \otimes A^\dagger = I^\dagger \otimes A^\dagger = (I \otimes A)^\dagger$$

$$\cancel{(A \otimes I)^\dagger}$$

with I the plain identity matrix.

A.3 matrix exponential

$$\begin{aligned} e^{U^\dagger A U} &= \sum_{k=0}^{+\infty} \frac{(U^\dagger A U)^k}{k!} \\ &= \sum_{k=0}^{+\infty} \frac{U^\dagger A^k U}{k!} \\ &= U^\dagger \sum_{k=0}^{+\infty} \frac{A^k}{k!} U \\ &= U^\dagger e^A U \end{aligned}$$

This is what happens with $CNOT_{0 \to 1}$ in the example that is given.

$$\begin{aligned}
e^{-idt(I\otimes Z)} &= \sum_k \frac{(-idt)^k (I\otimes Z)^k}{k!} \\
&= \sum_k \frac{(-idt)^k (I\otimes Z^k)}{k!} \\
&= I \otimes \left(\sum_k \frac{(-idt)^k Z^k}{k!} \right) \\
&= I \otimes RZ(2\cdot dt)
\end{aligned}$$

References

[1] Peter JJ O'Malley, Ryan Babbush, Ian D Kivlichan, Jonathan Romero, Jarrod R McClean, Rami Barends, Julian Kelly, Pedram Roushan, Andrew Tranter, Nan Ding, et al. Scalable quantum simulation of molecular energies. *Physical Review X*, 6(3):031007, 2016.