Chemistry on a quantum computer: Calculating the hydrogen molecule ground state

M. Tiggelman R.J.J. van Gulik

TU Delft
Faculty of Applied Sciences
MSc Applied Physics
Quantum Information Project

February 16, 2018

Abstract

Since the realization of few-qubit quantum computers, there has been an increasing effort in the simulation of quantum systems. Quantum computers are well-suited for this task since they are inherently quantum mechanical, in contrast to classical computers which must scale exponentially in computing power with the problem size. In this report, we investigate the recent results obtained by IBM and others in the electronic structure calculations of simple molecules. In particular, we perform a calculation of the bond energy of a hydrogen molecule using the quantum variational eigensolver method. The calculated bond length of 75 pm and bond energy of 5.4 eV agree with the results found by Kandala et al. [1]. About 30% of our calculated ground energies are within chemical accuracy (0.0016 hartree) of the exact energies obtained from the Hamiltonian.

Contents

1 Introduction

Using a quantum computer for the purpose of simulating quantum systems is a promising endeavour, since these kinds of systems are exponentially hard to solve on classical computers as the problem size increases. One of the most direct applications could result from solving so-called electronic structure problems, which corresponds to finding the wavefunctions and energies of a system of nuclei and electrons [2]. These results could be used to optimize the synthesis of chemical compounds or even to invent new molecules for use in medicine or catalysts. Such computations eliminate the need for possibly difficult or even dangerous experiments.

Solving electronic structure problems is already done in the field of (computational) quantum chemistry [2], but there is a limit to the complexity of molecules that can be simulated. Since a quantum computer is inherently quantum mechanical, the electron states of a molecule can in theory be efficiently represented. The primary research question is in which way this might be done most effectively, and what requirements there are on the amount of qubits, gate fidelity, and qubit coherence times.

In the last two decades, as few-qubit quantum computers have become a reality, a growing effort has been made to develop a theoretical and experimental understanding of the requirements for a quantum simulation of molecules [3–8]. Of the methods investigated, quantum phase estimation (QPE) [3–5, 9] and more recently the quantum variational eigensolver (QVE) [1, 6, 8–10] have emerged as successful approaches. A particular disadvantage of QPE compared to the QVE is the need for high qubit coherence times [1] and as such we consider only the QVE as it looks more promising for short term applications.

In this report, we will follow the approach that the research team at IBM has taken [1] to calculate the bond energy of a hydrogen molecule. We will use a different approach in the derivation of the molecular Hamiltonian, and compare the resulting energies accordingly. Our aim is to have a good understanding of the steps required, advantages, and limitations of performing a quantum simulation of molecular energies using the QVE method.

2 Theory

In this chapter the theory required for calculating the molecular ground state energy starting from first principles (the Schrödinger equation) is described. Finally, properties of the molecule of interest (H_2) that are needed for this calculation are given.

2.1 The Molecular Hamiltonian

In ab-initio (from first principles) methods of quantum chemistry the fundamental objective is to find solutions to the Schrödinger equation, in this case the molecular Hamiltonian [2]

$$\begin{aligned}\hat{H} &= \hat{T}_n + \hat{T}_e + \hat{V}_{ne} + \hat{V}_{nn} + \hat{V}_{ee} \\ &= -\sum_i \frac{\nabla^2_{R_i}}{2M_i} - \sum_i \frac{\nabla^2_{r_i}}{2} - \sum_{ij} \frac{Z_i}{|R_i - r_j|} + \sum_{i<j} \frac{Z_i Z_j}{|R_i - R_j|} + \sum_{i<j} \frac{1}{|r_i - r_j|}\end{aligned} \tag{1}$$

where $\hat{T}_n$ and $\hat{T}_e$ are the kinetic energy operators for the nuclei and electrons respectively. Finally $\hat{V}_{ne}$, $\hat{V}_{nn}$, and $\hat{V}_{ee}$ are the potential energy operators for the Coulomb interaction between the nuclei and electrons, nuclei, and electrons respectively. The units typically used in computational chemistry are atomic units, where the unit of mass is the electron mass m_e, that of charge is the electron charge e, and that of action is $\hbar$. This allows for the compact form of Eq. 1.

Directly solving Eq. 1 is hard because of the amount of degrees of freedom. One therefore typically applies the Born-Oppenheimer approximation, where the nuclei are assumed to be stationary. From this it follows that $\hat{T}_n = 0$ and $\hat{V}_{nn}$ is constant. The resulting Hamiltonian is typically said to be in first-quantized form.

We now move from this form to the second-quantized form, where we remove even more degrees of freedom by assuming the electrons can only be in certain orbits instead of keeping track of their spatial coordinates and momenta. A state of a molecule can now be denoted in the occupational basis $|n_N, ..., n_1\rangle$, where each n_j can be either 0 (unoccupied) or 1 (occupied), because we are dealing with fermions and Pauli's exclusion principle. We use the creation and annihilation operators $a_j^\dagger$ and a_j to respectively add or remove an electron from orbit j. The molecular many-body Hamiltonian can now be written as [8]

$$H = H_1 + H_2 = \sum_{pq} h_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s \tag{2}$$

where H_1 contains one-body operators and H_2 contains two-body operators. In general this many-body Hamiltonian is truncated at two-body interactions, but for us it is exact since we only have two electrons. The summation indices p, q, r, s are integers denoting certain orbits. The coefficients h_{pq} and h_{pqrs} are defined as the following expectation values [2]

$$h_{pq} = \langle \phi_p | \hat{T}_e + \hat{V}_{ne} | \phi_q \rangle = \int \phi_p^*(\sigma) \left(-\frac{\nabla_r^2}{2} - \sum_i \frac{Z_i}{|R_i - r|} \right) \phi_q(\sigma) \, \mathrm{d}\sigma \tag{3}$$

$$h_{pqrs} = \langle \phi_p \phi_q | \hat{V}_{ee} | \phi_r \phi_s \rangle = \int \phi_p^*(\sigma_1) \phi_q^*(\sigma_2) \left(\frac{1}{|r_1 - r_2|} \right) \phi_s(\sigma_1) \phi_r(\sigma_2) \, \mathrm{d}\sigma_1 \, \mathrm{d}\sigma_2 \, . \tag{4}$$

The above integrals can be readily computed using most quantum chemistry software packages, and are usually referred to as one- and two-electron integrals.

2.2 Jordan-Wigner transformation

At this point we want to convert our fermionic Hamiltonian (Eq. 2) into a series of qubit gates. This is a mapping from creation and annihilation operators to spin operators and can be done using the Jordan-Wigner or Bravyi-Kitaev transformations [5]. The Jordan-Wigner transform works in the occupation number basis and as such it is straightforward to define the following creation and annihilation operators in terms of qubit gates

$$\hat{Q}^- = \frac{1}{2}(\hat{X} + i\hat{Y}) = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \qquad \hat{Q}^+ = \frac{1}{2}(\hat{X} - i\hat{Y}) = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \tag{5}$$

where $\hat{X}$ and $\hat{Y}$ denote the Pauli X and Y gates respectively. These operators have the following effect on the computational basis states

$$\hat{Q}^- |0\rangle = 0 \qquad \hat{Q}^- |1\rangle = |0\rangle \qquad \hat{Q}^+ |0\rangle = |1\rangle \qquad \hat{Q}^+ |1\rangle = 0. \tag{6}$$

There is however still a sign problem when we apply the operators in Eq. 5 on an arbitrary occupation state $|n_N, ..., n_1\rangle$ due to the fermionic anticommutation relations

$$\{a_i^\dagger, a_j^\dagger\} = \{a_i, a_j\} = 0 \tag{7}$$

$$\{a_i^\dagger, a_j\} = \delta_{ij}. \tag{8}$$

These relations result in an overall sign change of the wavefunction depending on the parity of the number of occupied orbits below the orbit that is acted upon. This can fortunately be solved by applying a series of $\hat{Z}$ gates in front of our operators from Eq. 5 [5]

$$a_j^\dagger = \left(\prod_{k=1}^{j-1} \hat{Z}_k\right) \frac{1}{2}(\hat{X}_j + i\hat{Y}_j) \qquad a_j = \left(\prod_{k=1}^{j-1} \hat{Z}_k\right) \frac{1}{2}(\hat{X}_j - i\hat{Y}_j). \tag{9}$$

where j is the affected orbit (and the number of the affected qubit). We can use the above definitions directly in Eq. 2.

2.3 Quantum variational eigensolver

The quantum variational eigensolver (QVE) algorithm is based upon the variational principle, which states that the energy expectation value of a wavefunctions is always larger or equal to the ground state energy [2]

$$E_{\text{ground}} \leq \langle \psi | \hat{H} | \psi \rangle \,. \tag{10}$$

This principle is powerful since it allows us to create an approximate wavefunction depending on a set of parameters $\psi(\theta_1, \theta_2, ..., \theta_N)$ of which we can then iteratively calculate the energy expectation value and adjust the parameters to move towards the ground state solution [8].

Ideally we would like to be able to use this set of parameters to explore the entire Hilbert space, but this is impractical due to the amount of parameters needed. In practice one makes an educated guess (ansatz) to find a starting wavefunction that has a reasonable overlap with the ground state.

The choice of ansatz is however not straightforward. Methods used until now are adiabatic evolution [4], unitary coupled-cluster theory [9], and single qubit rotations combined with naturally entangling interactions [1].

2.4 Hydrogen molecule

The hydrogen molecule H_2 is one of the simplest molecules to work with in quantum chemistry. As two hydrogen atoms get closer together the electron clouds are shared and form a covalent bond, reducing the energy of the system. At an interatomic distance of 74.1 pm [11] the molecule is in an energy minimum and thus in its ground state. The lowest energy orbit is symmetric and the higher energy orbit is antisymmetric, as can be seen in Fig. 1. These two orbitals are simply two orthogonal linear combinations formed from two single-electron wavefunctions.

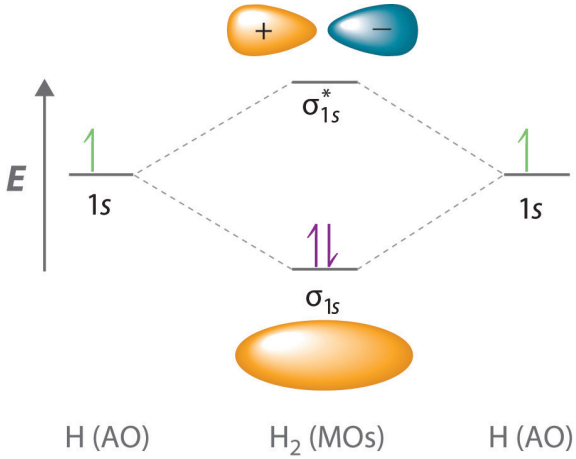


Figure 1: Electron orbitals of H_2 as a function of the interatomic distance [12]. As the atoms move closer together the atomic orbitals (AO) form two σ molecular orbitals (MO). In the ground state the symmetric orbital is fully occupied.

As both orbits can be occupied by a spin-down and a spin-up electron, we have a total of four orbits, and so in the second quantized Hamiltonian (Eq. 2) the summation indices will run from zero to three.

3 Methods

3.1 Generation of the molecular Hamiltonian

The results of the one- and two-electron integrals that are needed in the second quantized Hamiltonian are computed using the open source PSI4 quantum chemistry package [13]. This is done by calculating the orbital wavefunctions using a standard Hartree-Fock method and the minimal STO-3G Gaussian orbital basis set, after which the integrals are evaluated for a series of interatomic distances. The Python code for this calculation is in appendix A.

There are two unique integrals for the one-body operators, denoted h_0 and h_1 and four for the two-body operators, denoted h_2 up to h_5, making up a total of six integrals to evaluate per interatomic distance.

For the one-body operators we only retain the number operators $a_p^\dagger a_q$ where $p = q$, resulting in four terms. As for the two-body operators, the summation goes over $4^4 = 256$ terms. Many of these terms cancel or are zero due to the fact that $a_p^\dagger a_q^\dagger a_r a_s = (-a_q^\dagger a_p^\dagger) a_r a_s = a_p^\dagger a_q^\dagger (-a_s a_r) = a_q^\dagger a_p^\dagger a_s a_r$ as follows from the anticommutation relation in Eq. 7. The final terms in the full Hamiltonian are given below in table 1.

Table 1: Integral values corresponding to specific terms in the second quantized Hamiltonian [7].

One-body terms	Corresponding integral values
$h_{00} = h_{11}$	h_0
$h_{22} = h_{33}$	h_1
Two-body terms	
h_{0110}	h_2
h_{2332}	h_3
$h_{0220} = h_{0330} = h_{1221} = h_{1331}$	h_4
$h_{0202} = h_{0303} = h_{1212} = h_{1313} = h_{2130} = h_{2310} = h_{0213} = h_{0132}$	h_5

The resulting Hamiltonian is generated symbolically using the SymPy Python package. The creation and annihilation operators are substituted by their Jordan-Wigner transformed equivalents and the full equations is expanded and simplified such that every term contains only one Pauli gate per qubit. This yields the Hamiltonian

$$\begin{aligned} H = {} & 1.0h_0 + 0.5h_0Z_0 + 0.5h_0Z_1 + 1.0h_1 + 0.5h_1Z_2 + 0.5h_1Z_3 + 0.25h_2 + 0.25h_2Z_0 + 0.25h_2Z_0Z_1 \\ & + 0.25h_2Z_1 + 0.25h_3 + 0.25h_3Z_2 + 0.25h_3Z_2Z_3 + 0.25h_3Z_3 + 1.0h_4 + 0.5h_4Z_0 + 0.25h_4Z_0Z_2 \\ & + 0.25h_4Z_0Z_3 + 0.5h_4Z_1 + 0.25h_4Z_1Z_2 + 0.25h_4Z_1Z_3 + 0.5h_4Z_2 + 0.5h_4Z_3 + 1.0h_5 \\ & - 0.25h_5X_0X_2X_1X_3 - 0.25h_5Y_0Y_2X_1X_3 - 0.25h_5Y_1Y_3X_0X_2 - 0.25h_5Y_1Y_3Y_0Y_2 + 0.5h_5Z_0 \\ & + 0.25h_5Z_0Z_2 + 0.25h_5Z_0Z_3 + 0.5h_5Z_1 + 0.25h_5Z_1Z_2 + 0.25h_5Z_1Z_3 + 0.5h_5Z_2 + 0.5h_5Z_3 \end{aligned} \tag{11}$$

which is now ready to be used in the QVE algorithm. For every interatomic distance, these integral values need to be calculated and plugged into the Hamiltonian. After plugging these in and collecting the terms we are left with 15 Pauli terms in the Hamiltonian:

$$\begin{aligned} H = {} & g_0\mathbb{I} + g_1Z_0 + g_2Z_1 + g_3Z_2 + g_4Z_3 + g_5Z_0Z_1 + g_6Z_0Z_2 + g_7Z_0Z_3 + g_8Z_1Z_2 + g_9Z_1Z_3 \\ & + g_{10}Z_2Z_3 + g_{11}X_0X_1X_2X_3 + g_{12}X_0Y_1X_2Y_3 + g_{13}Y_0X_1Y_2X_3 + g_{14}Y_0Y_1Y_2Y_3 \end{aligned} \tag{12}$$

where the g terms are the integral values collected by Pauli term.

3.2 Creating the parametrized wavefunction

The quantum variational eigensolver requires a parametrized state. In the case of quantum chemistry, we are interested in non-computational states that are superpositions. This is where we can take advantage of the quantum computer. To be able to do this we need a variational trial state that is entangled. We have used the parametrization used by Kandala et al. [1]. It can be seen in figure 2 and is defined as:

$$|\psi(\boldsymbol{\theta})\rangle = \left[U_{\text{ent}} U_{\text{single}}(\boldsymbol{\theta})\right]^m |+\rangle \tag{13}$$

where U_{ent} is a collection of C-phase gates between the first qubit and any of the other and m is the 'depth' of the circuit (the amount of iterations of the entangling and rotational operations). The single qubit operations are parametrized and defined as:

$$U_{\text{single}}(\boldsymbol{\theta}) = \prod_{i=1}^{n} Z(\theta_i) Y(\theta_{n+i}) \tag{14}$$

where n is the amount of qubits. In our case we have 4 qubits (one for each orbital) and a depth of $m = 4$ was used. This depth allows us to reach chemical accuracy and gives us 32 optimization parameters.

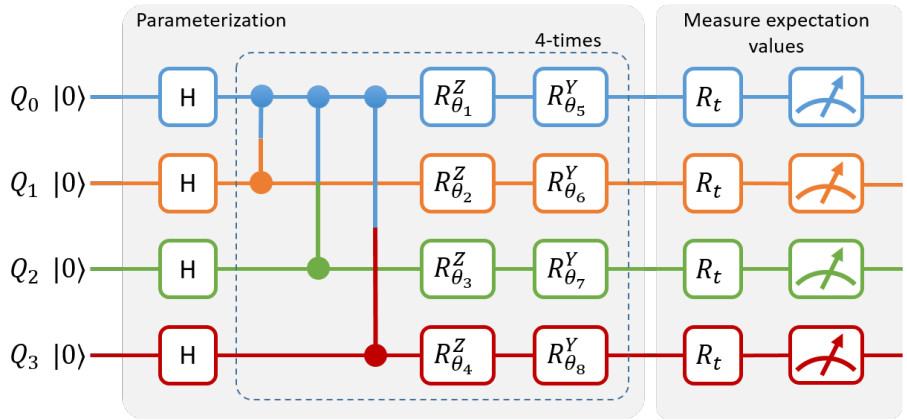


Figure 2: Parametrization as implemented and suggested by [1]. At every run the system gets placed in the $|+\rangle$ state before being run through the entangling and rotation operations. These operations are run 4 times to create an entangled state with 32 optimization parameters. A final set of rotations is applied to measure the expectation value of each Pauli term in the Hamiltonian of equation 12.

3.3 Measurement of the energy expectation values

With the variational quantum eigensolver we solve for the parameters $\boldsymbol{\theta}$ by a classical optimization routine. What this optimizer does is change the parameters to minimize the energy. The way to estimate the energy is by measuring the expectation value of each of the terms in equation 12 and summing them up:

$$E(\boldsymbol{\theta}) = \sum_i g_i \langle\psi(\boldsymbol{\theta})| H_i |\psi(\boldsymbol{\theta})\rangle \tag{15}$$

Where g_i are scalars found by combining the results of the one and two electron integrals, and the H_i are the Pauli terms of the Hamiltonian as in equation 12.

This output is then fed into the classical minimization routine, which in turn suggests a new set of parameters. The energy of this new state can then be measured and fed into the algorithm. This keeps on going for an n amount of iterations until the energy converges to a minimum, at which point the ground state energy is found according to the VQE approximation.

3.4 Optimization of the parameters using the SPSA algorithm

To find the ground state, we need to minimize the energy using the parametrization explained above. This can be done classically using one of the readily available minimization algorithms. The one we used is Simultaneous Perturbation Stochastic Approximation (SPSA). This is an algorithm well suited for a problem with a larger number of parameters [14].

The goal is then to minimize $E(\boldsymbol{\theta})$. It is then assumed that E is differentiable and that the at the minimum point $\boldsymbol{\theta}^*$ the gradient is equal to zero:

$$g(\boldsymbol{\theta}^*) = \frac{\delta E(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^*} = 0 \tag{16}$$

But unlike other gradient based minimization algorithms, no direct measurements of the gradient are assumed to be available. If the function E has multiple minima, it is possible for the algorithm to converge only to a local minimum. This can be avoided to a certain extent by the choice of parameters within the SPSA algorithm.

The algorithm consists of six steps to iteratively produces improving estimates of the parameters.

Step 1: initial guess We start by picking initial values for the parameters $\boldsymbol{\theta_0}$. These can be randomly generated and are independend. Two coefficients also need to be defined: c_k and a_k; which determine the size of perturbations used in the algorithm. There are no optimal values available, although good estimates can be made. In our case the SPSA calibration present in the QISKit [15] is used.

Step 2: generate perturbation Now a random perturbation vector $\boldsymbol{\Delta_k}$ needs to be generated. The dimension p of the vector is equal to the number of parameters. Each element of the perturbation vector is randomly and independently generated.

Step 3: evaluate Based on the perturbation, two measurements of the energy can be done: $E(\boldsymbol{\theta_k} + c_k\boldsymbol{\Delta_k})$ and $E(\boldsymbol{\theta_k} - c_k\boldsymbol{\Delta_k})$. Where $\boldsymbol{\theta_k}$ is the estimate at iteration k.

Step 4: gradient After evaluating, the gradient of the function can be approximated:

$$\hat{g_k}(\boldsymbol{\theta_k}) = \frac{E(\boldsymbol{\theta_k} + c_k\boldsymbol{\Delta_k}) - E(\boldsymbol{\theta_k} - c_k\boldsymbol{\Delta_k})}{2c_k} \begin{bmatrix} \Delta_{k1}^{-1} \\ \Delta_{k2}^{-1} \\ \vdots \\ \Delta_{kp}^{-1} \end{bmatrix} \tag{17}$$

with Δ_{ki} the ith element of the Δ_k perturbation vector.

Step 5: update estimate Now the estimate can be updated for the next iteration using the gradient found in the previous step:

$$\boldsymbol{\theta_{k+1}} = \boldsymbol{\theta_k} - a_k\hat{g_k}(\boldsymbol{\theta_k}) \tag{18}$$

Step 6: iterate If the algorithm has not converged or the maximum amount of steps is not reached, return to step 2 with $k = k + 1$. Otherwise terminate the algorithm and return the optimized parameters.

This optimization routine and the calibration was run with the open source Quantum Information Software Kit (QISKit) by IBM [15]. Figure 3 shows an example of one run with the SPSA algorithm set to 280 iterations at an intermolecular distance of 0.1Å. The error between the exact energy and the energy from the SPSA algorithm is 0.0077 Hartree, not far from the chemical accuracy (0.0016 Hartree). It could be improved by either letting the algorithm run for more iterations or getting a better starting estimate.

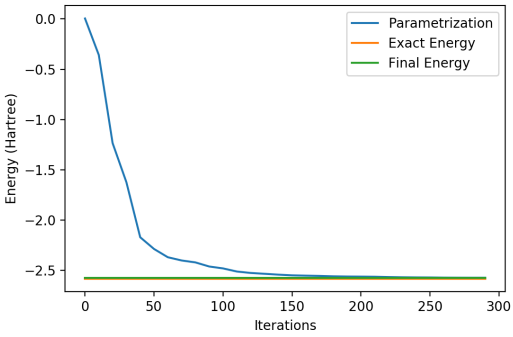


Figure 3: Convergence of the SPSA algorithm on a sample run at an intermolecular distance of 0.1Å, set to a maximum of 280 iterations. The error in the final value is 0.0077 Hartree. This could be improved by a better estimate or more iterations.

4 Results and discussion

The results were achieved by a computation of 20 energy values between 0.1 and 4 Ångstrom, running for 300 iterations per measurement point. Four runs were done on each measurement point and the lowest outcome was selected in post processing. This is done to increase the chance of reaching a global minimum; as the initial parameter guess is random, there exists the chance to converge slowly or get stuck in a local minimum if the initial angles are chosen poorly.

Figure 4a shows the resulting H_2 energy curve and errors from the exact value. In 4a we see the exact and measured energies. For comparison, the energies calculated by IBM are also included. The bond length at minimum energy ($R = 0.75$Å) corresponds to literature [11]. And the bond energy (5.4 eV) corresponds to those found in other literature [1, 9].

Errors of the QVE computation with respect to the exact value are shown in figure 4b. Due to the variational principle the errors are always positive, as the energy is always equal to or larger than the exact energy. In this figure we can see that the chemical accuracy (dotted line) is within reach: seven out of twenty data points lie within the chemical accuracy. The accuracy of the other points could be increased by either running the SPSA algorithm for more iterations, or by a different set of initial guess parameters. In this simulation the SPSA algorithm was set for a fixed amount of iteraions. Another possibility to assure chemical accuracy, is to let the algorithm proceed until it converges within chemical accuracy.

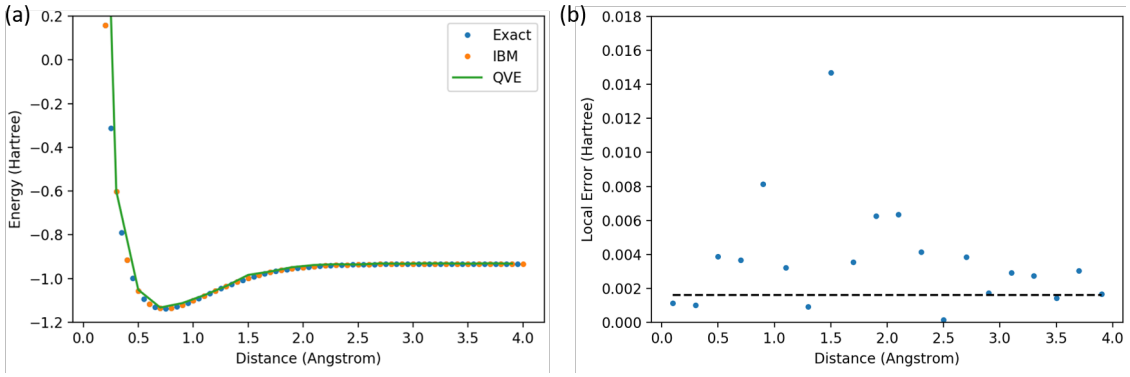


Figure 4: The computed H_2 energy curve and the errors with the exact value. (a) shows the energy curve as determined by QVE, exact calculations and values obtained by IBM. The bond length and bond energy correspond to literature: $R = 0.75$Å and $E = 5.4$ eV respectively. (b) shows the error of the QVE computation with respect to the exact values. The dotted line represents the chemical accuracy. This shows that the chemical accuracy is obtainable with this QVE implementation.

5 Conclusion and outlook

The computations presented in this paper demonstrate the possibility of using QVE to simulate molecular systems using (a simulation of) a quantum computer. Using the IBM QISKit, state parametrization and a classical minimalization algortihm, we were able to compute the potential energy curve of molecular hydrogen. From this potential energy curve we could extract values of the bond length and energy which agree to with literature. In one third of the measurements, the chemical accuracy threshold was reached. This shows that chemical accurate calculation are within reach.

For implementation on an actual quantum computer, the parametrization should be optimized to decrease the amount of gates required. In our calculations we used 20 gates per qubit for parametrization. This would require coherence times of more than one microsecond for one run of a Hamiltonian term evaluation. Unitary coupled cluster theory might be able to provide parametrization using a smaller amount of gates.

Another way to optimize the calculations is to reduce the number of qubits in the system. This could be done by using the Bravyi-Kitaev transformation instead of the Jordan-Wigner transformation. Using this transformation reduces the representation of the creation/annihilation operators from $O(n)$ to $O(\log(n))$, with n the amount of qubits.[16]

References

[1] A. Kandala et al. "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets". In: *Nature* 549.7671 (2017), pp. 242–246. arXiv: 1704.05018. URL: http://dx.doi.org/10.1038/nature23879.

[2] I. N. Levine, D. H. Busch, and H. Shull. *Quantum chemistry*. Vol. 5. Prentice Hall Upper Saddle River, NJ, 2000.

[3] S. B. Bravyi and A. Y. Kitaev. "Fermionic quantum computation". In: *Annals of Physics* 298.1 (2002), pp. 210–226. arXiv: 0003137 [quant-ph].

[4] A. Aspuru-guzik. "Simulated Quantum Computation of Molecular Energies". In: *Science* 1704.2005 (2005), pp. 1704–1708.

[5] A. Tranter et al. "The Bravyi-Kitaev transformation: Properties and applications". In: *International Journal of Quantum Chemistry* 115.19 (2015), pp. 1431–1441.

[6] B. P. Lanyon et al. "Towards quantum chemistry on a quantum computer". In: *Nature Chemistry* 2.2 (2010), pp. 106–111. arXiv: 0905.0887.

[7] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik. "Simulation of electronic structure Hamiltonians using quantum computers". In: *Molecular Physics* 109.5 (2011), pp. 735–750. arXiv: 1001.3855.

[8] J. R. McClean et al. "The theory of variational hybrid quantum-classical algorithms". In: *New Journal of Physics* 18.2 (2016), p. 23023. arXiv: 1509.04279. URL: http://dx.doi.org/10.1088/1367-2630/18/2/023023.

[9] P. J. O'Malley et al. "Scalable quantum simulation of molecular energies". In: *Physical Review X* 6.3 (2016), pp. 1–13. arXiv: 1512.06860.

[10] A. Peruzzo et al. "A variational eigenvalue solver on a photonic quantum processor". In: *Nature Communications* 5.May (2014). arXiv: 1304.3061.

[11] W. M. Haynes. *CRC handbook of chemistry and physics*. CRC press, 2014.

[12] D. A. McQuarrie et al. *Physical Chemistry: A molecular approach*. 539 M34. 1997.

[13] J. M. Turney et al. "Psi4: an open-source ab initio electronic structure program". In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 2.4 (2012), pp. 556–565.

[14] J. C. Spall. "Implementation of the simultaneous perturbation algorithm for stochastic optimization". In: *IEEE Transactions on Aerospace and Electronic Systems* 34.3 (July 1998), pp. 817–823.

[15] A. Cross. "The IBM Q experience and QISKit open-source quantum computing software". In: *Bulletin of the American Physical Society* (2018).

[16] J. T. Seeley, M. J. Richard, and P. J. Love. "The Bravyi-Kitaev transformation for quantum computation of electronic structure". In: *The Journal of chemical physics* 137.22 (2012), p. 224109.

Appendix A: Computation of the one- and two-electron integrals

```
import psi4
import numpy as np

def get_integrals(r):
    """ Interatomic distance r in angstrom """
    h2 = psi4.geometry("""
    H
    H 1 {}
    symmetry c1
    """.format(r))

    psi4.set_options({'basis': 'sto-3g'})

    scf_e, scf_wfn = psi4.energy('scf', molecule=h2, return_wfn=True)

    mints = psi4.core.MintsHelper(scf_wfn.basisset())

    H = np.asarray(mints.ao_kinetic()) + np.asarray(mints.ao_potential())

    # Coefficients of the linear combinations that form the
    # symmetric (occupied) and antisymmetric (virtual) orbit
    coef_occ = scf_wfn.Ca_subset('AO', 'OCC')
    coef_vir = scf_wfn.Ca_subset('AO', 'VIR')

    h0 = (H[0,0] + H[0,1]) * (np.asarray(coef_occ)[0][0]**2)*2
    h1 = (H[0,0] - H[0,1]) * (np.asarray(coef_vir)[0][0]**2)*2

    # Both electrons in the ground state
    h2 = np.asarray(mints.mo_eri(coef_occ, coef_occ, coef_occ, coef_occ))[0,0,0,0]
    # Both electrons in the excited state
    h3 = np.asarray(mints.mo_eri(coef_vir, coef_vir, coef_vir, coef_vir))[0,0,0,0]
    # One in excited and one in ground state
    h4 = np.asarray(mints.mo_eri(coef_occ, coef_occ, coef_vir, coef_vir))[0,0,0,0]
    h5 = np.asarray(mints.mo_eri(coef_occ, coef_vir, coef_occ, coef_vir))[0,0,0,0]

    return h0, h1, h2, h3, h4, h5


dr = 0.05
r_arr = np.arange(0.1, 4, dr)
results = np.zeros((len(r_arr), 6))

for i, r in enumerate(r_arr):
    results[i] = np.array(get_integrals(r))

np.savez('psi4_electron_integrals_dr={}.txt'.format(dr), r=r_arr, h=results)
```

Appendix B: QVE implementation

QVE - Report version final final v12.332

February 16, 2018

In [5]:
```
# General imports
from functools import reduce, partial
import numpy as np
from scipy import linalg as la

# Basic plot tools
import matplotlib.pyplot as plt
%matplotlib notebook

# Use sympy for symbolic mathematics
from sympy import *
init_printing()
from sympy import symbols
from sympy.physics.quantum import AntiCommutator, Commutator, Operator
from sympy.physics.quantum import Dagger, TensorProduct
from sympy.physics.quantum.gate import XGate, YGate, ZGate, gate_simp
from sympy.physics.quantum.gate import gate_sort, IdentityGate

# QISkit imports
from qiskit import QuantumProgram
from qiskit.tools.qi.pauli import Pauli, label_to_pauli
from qiskit.tools.apps.optimization import Hamiltonian_from_file, make_Hamiltonian
from qiskit.tools.apps.optimization import eval_hamiltonian, group_paulis
from qiskit.tools.apps.optimization import trial_circuit_ryrz, SPSA_optimization
from qiskit.tools.apps.optimization import SPSA_calibration, trial_circuit_ry
```

0.0.1 Utility functions

In [6]:
```
def add_z_gates(ex, qubit):
    # Add Z gates to account for parity,
    # in order to satisfy anticommutator
    if qubit > 0:
        z_gates = [ZGate(i) for i in range(qubit)]
        return ex * reduce(lambda x, y: y*x, z_gates)
    else:
        return ex
```

```
def qubit_creation(qubit):
    """ Jordan-Wigner transform of creation operator """
    return add_z_gates(0.5 * (XGate(qubit) - I*YGate(qubit)), qubit)

def qubit_annihilation(qubit):
    """ Jordan-Wigner transform of annihilation operator """
    return add_z_gates(0.5 * (XGate(qubit) + I*YGate(qubit)), qubit)

gate_simplify = {XGate: {XGate: (IdentityGate, 1),
                         YGate: (ZGate, I),
                         ZGate: (YGate, -I)},
                 YGate: {XGate: (ZGate, -I),
                         YGate: (IdentityGate, 1),
                         ZGate: (XGate, I)},
                 ZGate: {XGate: (YGate, I) ,
                         YGate: (XGate, -I),
                         ZGate: (IdentityGate, 1)}}

def simplify_circuit(circuit):
    """
    Simplify all products of two Pauli gates on the same qubit
    """
    # Perform the simplification for each term
    if isinstance(circuit, Add):
        return sum(simplify_circuit(t) for t in circuit.args)

    new_gates = 1
    not_optimized = 1

    # For every term in the product
    for i, term in enumerate(circuit.args):
        # If we are dealing with a Pauli gate
        if type(term) in [XGate, YGate, ZGate]:
            qubit = term.targets[0]
            changed = False
            # Loop over all terms again
            for j, term2 in enumerate(circuit.args[i+1:]):
                # If both Pauli gates operate on the same qubit,
                # simplify them
                if term2.targets[0] == qubit and changed == False:
                    replacement = gate_simplify[type(term)][type(term2)]
                    new_gates *= replacement[0](qubit) * replacement[1]
                    changed = True
                else:
                    not_optimized *= term2
            if changed == False:
                new_gates *= term
            break
```

```
            else:
                new_gates *= term

        if type(not_optimized) != int:
            if len(not_optimized.args) > 1 :
                new_gates *= simplify_circuit(not_optimized)
            else:
                new_gates *= not_optimized

        return new_gates

    def to_paulis(ex, N=4):
        """
        Convert a product of sympy Gate's to text form, where the sequence
        must be first simplified by simplify_gates in order to have
        one gate per qubit.

        The gates will be in order of the qubit numbers.

        Example
        X2*Z3*Y0 -> 'YIXZ'
        """

        cutoff = 1
        if len(ex.args) > 1 and ex.args[1] == I:
            cutoff = 2

        qubits = [arg.args[0] for arg in ex.args[cutoff:]]

        qubit_to_gate = {arg.args[0]: arg for arg in ex.args[cutoff:]}
        gate_to_str = {XGate: 'X', YGate: 'Y', ZGate: 'Z'}

        paulis = ''
        for i in range(N):
            if i in qubits:
                paulis += gate_to_str[type(qubit_to_gate[i])]
            else:
                paulis += 'I'

        return paulis

    def save_hamiltonian(H, filename):
        """
        Save the generated Hamiltonian in a text format readable
        by the QISKit functions
        """
        with open(filename, 'w') as f:
            for term in H.args:
```

```
            f.write(to_paulis(term, N=4) + '\n')
            if len(term.args) == 0:
                f.write(str(term) + '\n')
            else:
                extra = 'j' if term.args[1] == I else ''
                f.write(str(term.args[0]) + extra + '\n')

def get_h_pq(p, q):
    """
    Return the proper integral symbol for a certain p and q
    """
    h = IndexedBase('h')

    if p == q:
        if p < 2:
            return h[0]
        elif p >= 2:
            return h[1]
    else:
        return 0

def get_h_pqrs(p, q, r, s):
    h = IndexedBase('h')

    values = {'0110': 2, '1001': 2,

              '2332': 3, '3223': 3,

              '0220': 4, '0330': 4, '1221': 4, '1331': 4,
              '2002': 4, '3003': 4, '2112': 4, '3113': 4,

              '0202': 5, '1313': 5,
              '2130': 5, '2310': 5, '0312': 5, '0132': 5,

              '2020': 5, '3131': 5,
              '1203': 5, '3201': 5, '3021': 5, '1023': 5,

              '0303': 5, '3030': 5, '1212': 5, '2121': 5,
             }

    key = str(p) + str(q) + str(r) + str(s)
    if key in values:
        return h[values[key]]
    else:
        return 0

class h_pq(Function):
    """
```

```
        Wrap the get_h_pq function in a sympy Function object
        to be able to use it in a sympy expression.
        """
        @classmethod
        def eval(cls, p, q):
            if p.is_Integer and q.is_Integer:
                return get_h_pq(p, q)

    class h_pqrs(Function):
        """
        Wrap the get_h_pqrs function in a sympy Function object
        to be able to use it in a sympy expression.
        """
        @classmethod
        def eval(cls, p, q, r, s):
            if p.is_Integer and q.is_Integer and r.is_Integer and s.is_Integer:
                return get_h_pqrs(p, q, r, s)
```

0.0.2 Generating the Hamiltonian for the hydrogen molecule

In [7]:
```
# Create sympy symbols
h = IndexedBase('h')
cr = IndexedBase('a^+')
an = IndexedBase('a^-')
p, q, r, s = symbols('p q r s', integer=True)

# Generate the sum terms of the Hamiltonian
# We sum p,q,r,s over the 4 orbitals in the molecule:
# 2 in the sigma and 2 in the sigma* energy levels
h1 = Sum(h_pq(p,q)*cr[p]*an[q], (p,0,3), (q,0,3)).doit()
h2 = Sum(h_pqrs(p,q,r,s)*cr[p]*cr[q]*an[r]*an[s],
         (p,0,3), (q,0,3), (r,0,3), (s,0,3)).doit()

ex = h1 + 0.5 * h2

# Convert creation/annihilation operators to qubit gates
for i in range(4):
    ex = ex.subs(cr[i], qubit_creation(i))
    ex = ex.subs(an[i], qubit_annihilation(i))

# Simplify it
ex = gate_simp(expand(ex))

# Collapse to one gate per qubit
ex = simplify_circuit(ex)
ex
```

Out[7]:

$1.0h_0 + 0.5h_0Z_0 + 0.5h_0Z_1 + 1.0h_1 + 0.5h_1Z_2 + 0.5h_1Z_3 + 0.25h_2 + 0.25h_2Z_0 + 0.25h_2Z_0Z_1 + 0.25h_2Z_1 + 0.25h_3 + 0.25$

0.0.3 Exact ground state energies of our Hamiltonian

```
In [9]: # h_pq and h_pqrs values for different interatomic distances
        # generated using the PSI4 python quantum chemistry package
        data = np.load('psi4_electron_integrals_dr=0.05.npz')
        r_own, integrals = data['r'], data['h']

        E_own = []

        # For all interatomic distances
        for i in range(len(r_own)):
            # We have the Hamiltonian, all that is left is to substitute
            # our calculated integral values
            H = ex.subs(h, Array(integrals[i]))

            filename = 'H2/H2_r={}.txt'.format(i)
            save_hamiltonian(H, filename)

            # Load the Hamiltonian into QISKit
            pauli_list=Hamiltonian_from_file(filename)
            H = make_Hamiltonian(pauli_list)

            # Find the ground state energy eigenvalue exactly
            exact = np.amin(la.eig(H)[0])
            dist = r_own[i] / 0.529177
            coulomb_repulsion = 1 / dist

            E_own.append(exact.real + coulomb_repulsion)

        plt.figure()
        plt.plot(r_own, E_own, '.')
        plt.ylim(-1.25, 0.5)
        plt.xlabel('r (angstrom)')
        plt.ylabel('E (hartree)')
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>


Out[9]: <matplotlib.text.Text at 0x7fba3d988a90>
```

0.0.4 Compare our own results with those from the IBM example

IBM data taken from: https://github.com/QISKit/qiskit-tutorial/tree/master/4_applications/H2

```
In [10]: i = np.arange(0, 39)
         r_ibm = np.arange(0.2, 4.1, 0.1)

         E_ibm = []

         for j in range(len(i)):
             file = 'H2_ibm/PESMap{}atdistance{:.1f}.txt'.format(i[j], r_ibm[j])

             pauli_list = Hamiltonian_from_file(file)
             H = make_Hamiltonian(pauli_list)
             exact = np.amin(la.eig(H)[0])

             dist = r_ibm[j] / 0.529177
             coulomb_repulsion = 1 / dist

             E_ibm.append(exact.real + coulomb_repulsion)

         plt.figure()
         plt.plot(r_own, E_own, '.', label='own results')
         plt.plot(r_ibm, E_ibm, '.', label='IBM')

         plt.ylim(-1.25, 0.5)
         plt.xlabel('r (angstrom)')
         plt.ylabel('E (hartree)')

         plt.legend()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>


Out[10]: <matplotlib.legend.Legend at 0x7fba3b8517b8>
```

0.0.5 QVE

Now approximate/calculate the ground state using the QVE algorithm using the QISkit simulator

```
In [8]: # Set QVE parameters

        n = 4 #number of qubits
        m = 4 #depth of parametrization circuit
        shots = 1 #if shots=1, evaluate hamiltonian until converged
        max_trials = 300 #amount of SPSA iterations
        device = 'local_qasm_simulator'
        initial_theta = np.random.randn(2*n*m) #generate initial angles
        entangler_map = {0: [1,2,3]} #entagle qubit 0 with all others
```

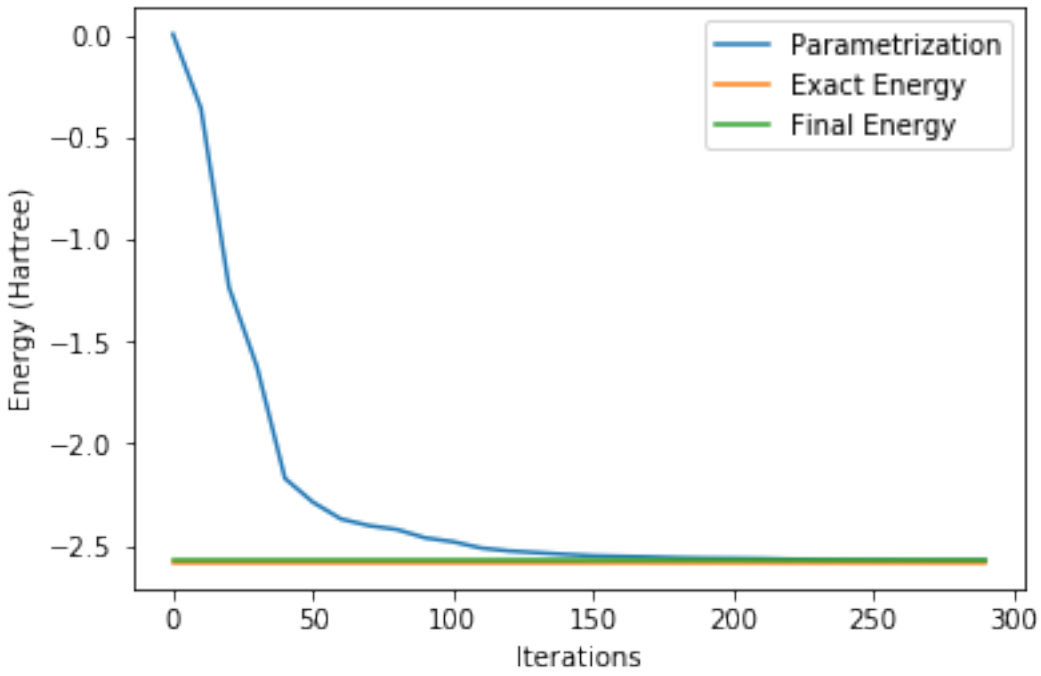
```
initial_c=0.01
target_update=2*np.pi*0.1
save_step = 400 #if save_step>max_trials, dont save inbetween data
```

In [12]:
```
# The function that SPSA has to optimize
def cost_function(Q_program,H,n,m,entangler_map,shots,device,theta):
    # Generate parametrized circuit with Ry and Rz rotation
    trial_circuit = trial_circuit_ryrz(n,m,theta,entangler_map,None,False)
    # Calculate the energy by evaluating the Hamiltonian
    energy = eval_hamiltonian(Q_program, H, trial_circuit, shots, device).real

    return np.real(energy)
```

In [11]:
```
plt.figure()
plt.plot(np.arange(0, max_trials,save_step),SPSA_iter, label = 'Parametrization')
plt.plot(np.arange(0, max_trials,save_step),
         np.ones(max_trials//save_step)*SPSA_compare,
         label='Exact Energy')
plt.plot(np.arange(0, max_trials,save_step),
         np.ones(max_trials//save_step)*SPSA_outcome,
         label='Final Energy')
plt.legend()

plt.xlabel('Iterations')
plt.ylabel('Energy (Hartree)')
plt.savefig('Plots/SPSA.png', dpi=200)
```



In [ ]:
```
E_qve = []
coulomb_rep = []

# Loop over the interatomic distances in steps of 4
for i in tqdm(range(0,len(r_own),4)):
    print("At r=" + str(r_own[i]))
    filename = 'H2/H2_r={}.txt'.format(i)

    # Load the Hamiltonian into QISKit
    pauli_list=Hamiltonian_from_file(filename)
    H = make_Hamiltonian(pauli_list)

    Q_program = QuantumProgram()

    # Calculate every point 5 times, with newly generated initial angles
    E_r_vals = []
    for k in range(5):
        initial_theta = np.random.randn(2*n*m)

        target_function = partial(cost_function, Q_program, H, n, m,
                                  entangler_map, shots, device)

        SPSA_params = SPSA_calibration(target_function, initial_theta,
                                       initial_c, target_update, 25)
        output = SPSA_optimization(target_function, initial_theta,
                                   SPSA_params, max_trials, save_step, 1);
        E_r_vals.append(output[0])

    E_qve.append(E_r_vals)

    # Calculate coulomb repulsion
    dist = r_own[i] / 0.529177
    coulomb_repulsion = 1 / dist
    coulomb_rep.append(coulomb_repulsion)
```

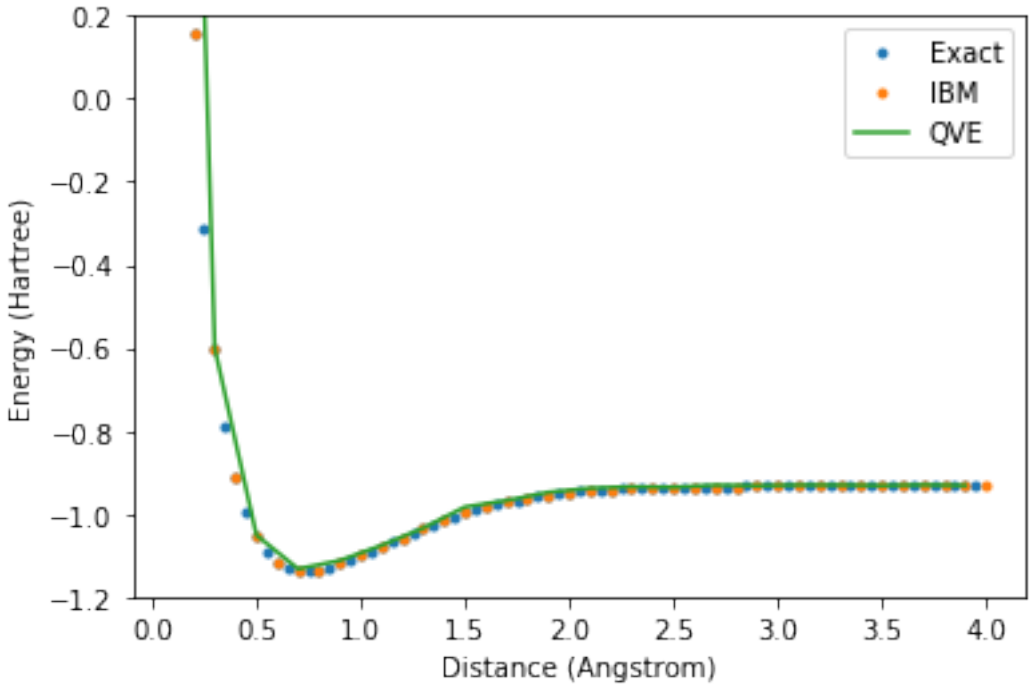
In [46]:
```
# Find the lowest energy values and add to the coulomb repulsion
E_min = np.zeros(len(E_qve))
for k, val in enumerate(E_qve):
    E_min[k] = np.min(val)
E_qve2 = np.asarray(E_min) + np.asarray(coulomb_rep)
```

In [78]:
```
plt.figure()
plt.plot(r_own, E_own, '.', label='Exact')
plt.plot(r_ibm, E_ibm, '.', label='IBM')
plt.plot(r_own[0:-1:4], E_qve2, '-', label = 'QVE')
plt.xlabel('Distance (Angstrom)')
plt.ylabel('Energy (Hartree)')
plt.legend()
```

```
plt.ylim(-1.2, 0.2)
```

Out[78]:

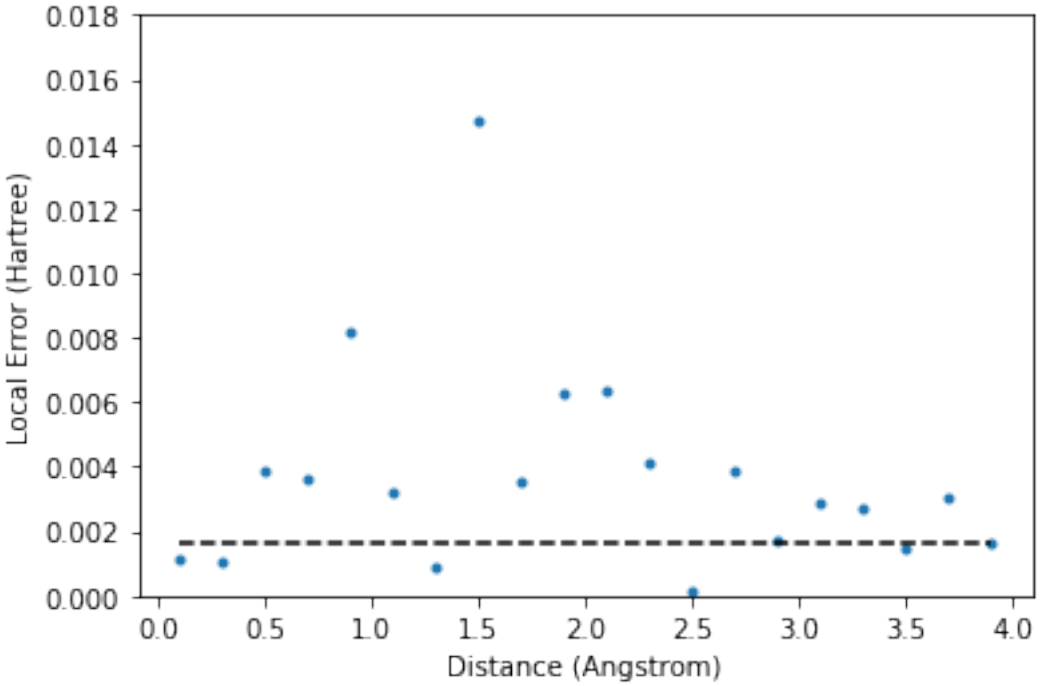
$$(-1.2,\quad 0.2)$$




```
In [ ]: # Calculate the error from the exact value
        E_error = []
        k = 0
        for i in tqdm(range(0,len(r_own),4)):
            error = E_qve2[k] - E_own[i]
            E_error.append(error)
            k += 1
```

```
In [75]: length = len(r_own[0:-1:4])

         plt.figure()
         plt.plot(r_own[0:-1:4], E_error,  '.', label = 'Error')
         plt.plot(r_own[0:-1:4], np.ones(length)*0.0016, 'k--')
         plt.xlabel('Distance (Angstrom)')
         plt.ylabel('Local Error (Hartree)')
         plt.ylim(0, 0.018)
```


0.018
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
Local Error (Hartree)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Distance (Angstrom)