ex14_solution

May 27, 2025

Computational Quantum Physics - PHYS 463

Lecturer: Prof. G. Carleo

Assistants: alessandro.sinibaldi@epfl.ch, linda.mauron@epfl.ch, lorenzo.fioroni@epfl.ch

0.1 Exercise 14.1 : The Variational Quantum Eigensolver (VQE)

In this exercise we are going to see how variational methods can be used in a hybrid quantum-classical scheme in order to approximate quantum states.

In particular the aim of the Variational Quantum Eigensolver (VQE) is to optimize a set of parameterized quantum gates to approximate the ground state of a quantum system, given that the expectation value of the energy and its derivatives with respect to the parameters are evaluated on a quantum hardware in a scalable way.

Preparing ground state is an exponentially complex task even on a quantum computer (QMA), but the VQE is expected to give some advantage for specific tasks in physics, chemistry and/or material sciences.

0.1.1 Classical simulation of H_2

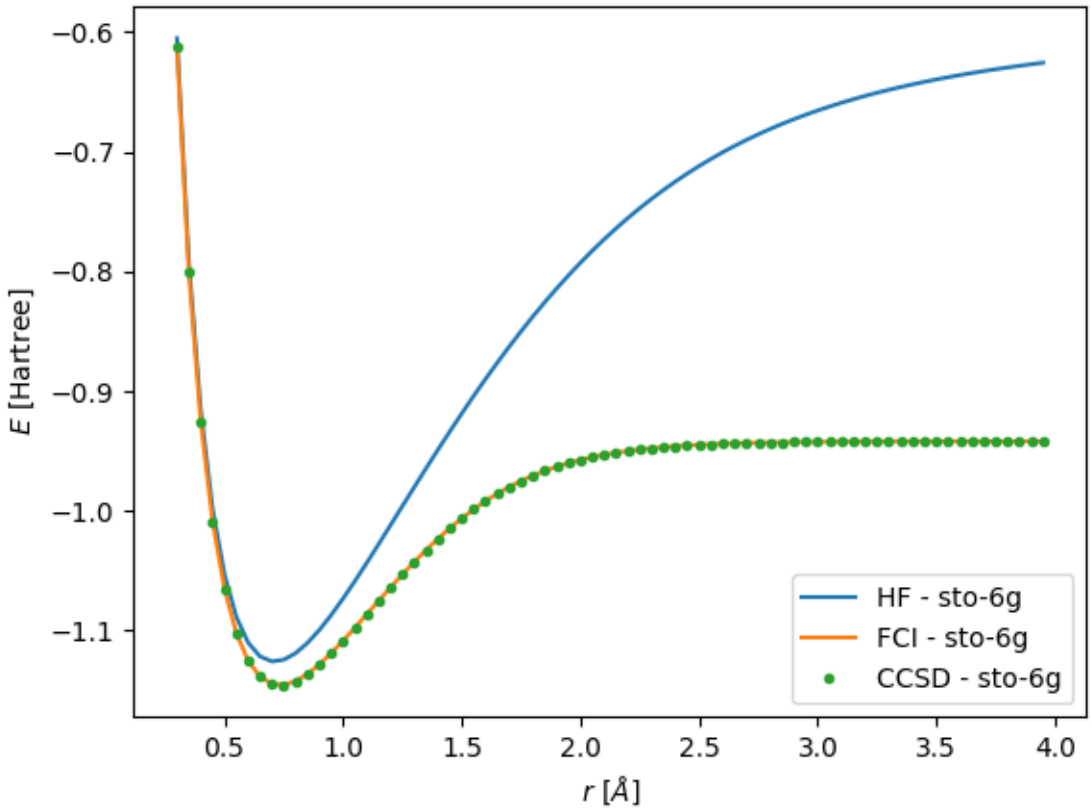
Here we will recall the code for the classical simulation of the dissociation process of H_2 diatomic molecule.

[50]:
```
import numpy as np
from pyscf import gto,scf,ao2mo,mp,cc,fci,tools
import matplotlib.pyplot as plt

np.set_printoptions(precision=2,suppress=1e-8,linewidth=120)
```

[51]:
```
## Define a set of distances between the two atoms
distances = np.arange(0.3, 4, .05)

## and a basis in which the calculations will be made
basis = 'sto-6g' #'6-31g' 'cc-pvdz' 'aug-cc-pvdz' ,...

## Define a dictionary containing the energies derived with different methods
energies = {}
```

[52]:
```
energies["HF_"+basis]   = []
energies["FCI_"+basis]  = []
energies["CCSD_"+basis] = []

for (i,r) in enumerate(distances):
    # Build the 3D geometry of the molecule
    geometry = "H .0 .0 .0; H .0 .0 "+str(r)

    # And pass it to the gaussian orbitals generator (this is an alternative␣
 ↪way to create the molecule object)
    mol = gto.
 ↪M(atom=geometry,charge=0,spin=0,basis=basis,symmetry=True,verbose=0)

    mf  = scf.RHF(mol) # Initialise a Restricted HF calculation using the␣
 ↪molecule constructed
    Ehf = mf.kernel()  # <- calling the kernel we compute the energy using the␣
 ↪orbitals obtained

    fci_h2 = fci.FCI(mf) # FCI calculation
    Efci = fci_h2.kernel()[0]

    ccsd_h2 = cc.CCSD(mf) # CCSD calculation
    e_ccsd  = ccsd_h2.kernel()[0]
    e_ccsd += Ehf # <- this lines is mandatory because CCSD computes the energy␣
 ↪difference with HF


    # Save energies
    energies["HF_"+basis].append(Ehf)
    energies["FCI_"+basis].append(Efci)
    energies["CCSD_"+basis].append(e_ccsd)
```

[53]:
```
## Now plot the result
plt.plot(distances,energies["HF_"+basis],label="HF - "+basis)
plt.plot(distances,energies["FCI_"+basis],label="FCI - "+basis)
plt.plot(distances,energies["CCSD_"+basis],label="CCSD -␣
 ↪"+basis,linestyle="",marker=".")

plt.xlabel(r"$r$ [$\AA$]")
plt.ylabel(r"$E$ [Hartree]")
plt.legend()
plt.show()
```



As can be seen, the CCSD calculation is as accurate as FCI in this small system.

0.1.2 Quantum simulation of H_2

Now we will proceed to the simulation on the quantum computer.

As showed here the Variational quantum eigensolver can be used to calculate dissociation processes of small molecules on quantum processor.

[54]:
```
from qiskit import QuantumCircuit
from qiskit.circuit import ParameterVector
from qiskit.quantum_info import SparsePauliOp # To create operators
from qiskit.primitives import Estimator # to estimate expectation values
```

Preparing the Hamiltonian First, since we want to study a fermionic system on a set of qubits, we must find a mapping between the two.

The oldest mapping proposed is the Jordan-Wigner, which was illustrated also in the lectures, but there are many others.

Two noticeable alternatives are the parity mapping and the Bravyi-Kitaev mapping.

In particular, we are going to focus on the parity mapping for the H_2 system.

Instead of storing the occupation of a fermionic state f_j, as Jordan-Wigner does, this isomorphism uses the qubits to store the quantity

$$p_j = \sum_{i=0}^{j-1} f_i \mod 2 \quad , \tag{1}$$

called parity of the set of occupation numbers $f_{j-1} \dots f_0$.

This is convenient for system like H_2 which conserve the total number of electrons with fixed spin orientation, meaning that we can get rid of two qubits out of the box.

Manually performing the mapping can be a very tedious and prone-to-error task, for this reason we will use the very practical Qiskit PySCF driver, which recalls the creation of a PySCF molecule object that we have seen in previous lectures but gives a qubit operator ready to be measured on a quantum computer.

This driver can be found in the Qiskit optional package `nature`.

[55]:
```
# !pip install 'qiskit-nature'
```

[56]:
```
from qiskit_nature.second_q.drivers import PySCFDriver
from qiskit_nature.second_q.formats.molecule_info import MoleculeInfo
from qiskit_nature.second_q.transformers import FreezeCoreTransformer
```

[57]:
```
# Initialize the molecule using the PySCF driver, this has the same structure␣
 ↪as the molecule object we created
# in the previous lectures
molecule = MoleculeInfo(["H", "H"], [(0.0, 0.0, 0.0), (0.0, 0.0, 0.735)],␣
 ↪charge=0, multiplicity=1)

driver = PySCFDriver.from_molecule(molecule, basis="sto6g")

# this is now done explicitly
problem = driver.run()
```

[58]:
```
# Now the problem has also the nuclear repulsion shift of the Born-Oppenheimer␣
 ↪approximation
problem.nuclear_repulsion_energy
```

[58]:
```
np.float64(0.7199689944489797)
```

[59]:
```
## Calling .second_q_ops will generate a bunch of operator such as dipole,␣
 ↪magnetization, ...
## We are interested in the Hamiltonian
hamiltonian = problem.second_q_ops()[0]
```

[60]:
```
print(hamiltonian)
```

```
Fermionic Operator
number spin orbitals=4, number terms=36
```

```
  -1.260626877670799 * ( +_0 -_0 )
+ -0.4761511477140961 * ( +_1 -_1 )
+ -1.260626877670799 * ( +_2 -_2 )
+ -0.4761511477140961 * ( +_3 -_3 )
+ 0.33782804623258983 * ( +_0 +_0 -_0 -_0 )
+ 0.3326288267523017 * ( +_0 +_1 -_1 -_0 )
+ 0.33782804623258983 * ( +_0 +_2 -_2 -_0 )
+ 0.3326288267523017 * ( +_0 +_3 -_3 -_0 )
+ 0.09060902125379115 * ( +_0 +_0 -_1 -_1 )
+ 0.09060902125379115 * ( +_0 +_1 -_0 -_1 )
+ 0.09060902125379115 * ( +_0 +_2 -_3 -_1 )
+ 0.09060902125379115 * ( +_0 +_3 -_2 -_1 )
+ 0.09060902125379115 * ( +_1 +_0 -_1 -_0 )
+ 0.09060902125379115 * ( +_1 +_1 -_0 -_0 )
+ 0.09060902125379115 * ( +_1 +_2 -_3 -_0 )
+ 0.09060902125379115 * ( +_1 +_3 -_2 -_0 )
+ 0.3326288267523017 * ( +_1 +_0 -_0 -_1 )
+ 0.35008911193642955 * ( +_1 +_1 -_1 -_1 )
+ 0.3326288267523017 * ( +_1 +_2 -_2 -_1 )
+ 0.35008911193642955 * ( +_1 +_3 -_3 -_1 )
+ 0.33782804623258983 * ( +_2 +_0 -_0 -_2 )
+ 0.3326288267523017 * ( +_2 +_1 -_1 -_2 )
+ 0.33782804623258983 * ( +_2 +_2 -_2 -_2 )
+ 0.3326288267523017 * ( +_2 +_3 -_3 -_2 )
+ 0.09060902125379115 * ( +_2 +_0 -_1 -_3 )
+ 0.09060902125379115 * ( +_2 +_1 -_0 -_3 )
+ 0.09060902125379115 * ( +_2 +_2 -_3 -_3 )
+ 0.09060902125379115 * ( +_2 +_3 -_2 -_3 )
+ 0.09060902125379115 * ( +_3 +_0 -_1 -_2 )
+ 0.09060902125379115 * ( +_3 +_1 -_0 -_2 )
+ 0.09060902125379115 * ( +_3 +_2 -_3 -_2 )
+ 0.09060902125379115 * ( +_3 +_3 -_2 -_2 )
+ 0.3326288267523017 * ( +_3 +_0 -_0 -_3 )
+ 0.35008911193642955 * ( +_3 +_1 -_1 -_3 )
+ 0.3326288267523017 * ( +_3 +_2 -_2 -_3 )
+ 0.35008911193642955 * ( +_3 +_3 -_3 -_3 )
```

[61]:
```
from qiskit_nature.second_q.mappers import JordanWignerMapper, ParityMapper
```

[62]:
```
## Create the qubit operator (With 2 qubit reduction)
mapper = ParityMapper()
qubit_op = mapper.map(hamiltonian)
print(qubit_op)
```

```
SparsePauliOp(['IIII', 'IIIZ', 'IIZZ', 'IZZI', 'ZZII', 'IIZI', 'IZZZ', 'ZZIZ',
'ZXIX', 'IXZX', 'ZXZX', 'IXIX', 'IZIZ', 'ZZZZ', 'ZIZI'],
              coeffs=[-0.82+0.j,  0.17+0.j, -0.22+0.j,  0.17+0.j, -0.22+0.j,
0.12+0.j,  0.17+0.j,  0.17+0.j,  0.05+0.j, -0.05+0.j,
```

```
 -0.05+0.j,  0.05+0.j,  0.17+0.j,  0.18+0.j,  0.12+0.j])
```

[63]:
```
## Create the qubit operator (With 2 qubit reduction)
mapper  = ParityMapper(num_particles=[1,1]) # Parity mapper requires the spin␣
 ↪up and down electrons
qubit_op = mapper.map(hamiltonian)
print(qubit_op)
```

```
SparsePauliOp(['II', 'IZ', 'ZI', 'ZZ', 'XX'],
              coeffs=[-1.06+0.j,  0.4 +0.j, -0.4 +0.j, -0.01+0.j,  0.18+0.j])
```

Now we have seen how to create a qubit operator for a single configuration, we will need to repeat this procedure for every r in order to study the dissociation process of H_2.

Initial state We need a starting point for our calculations, that in this case will be the Hartree-Fock (HF) determinant. This can be easily prepared on the quantum circuit, since we are already in the basis of the HF orbitals. This means that we will need to put an X gate if the corresponding spin-orbital is occupied (Jordan-Wigner) or it has a odd occupation number (Parity).

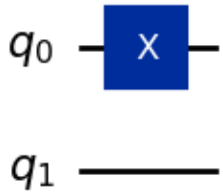
Qiskit provides a useful `HartreeFock` function to create this initial state, that then we will pass to the VQE function.

[64]:
```
from qiskit_nature.second_q.circuit.library.initial_states import HartreeFock
```

[65]:
```
init_state = HartreeFock(num_spatial_orbitals=2, num_particles=[1,1],␣
 ↪qubit_mapper=mapper)
```

[66]:
```
init_state.draw("mpl")
```

[66]:



Variational wave function Now that we have the Hamiltonian operator, we will need an ansatz for our wavefunction. Since we need a variational ansatz, some gates will contains parameters that are going to be iteratively optimized (for example, rotations about the x, y, z axes).

There is no analytical way to choose an ansatz for the system: there are empirical rules based on similarity with what we are studying. Some ansätze come from classical computational chemistry, such as the highly accurate q-UCCSD, but mostly we have to consider some circuits that can be

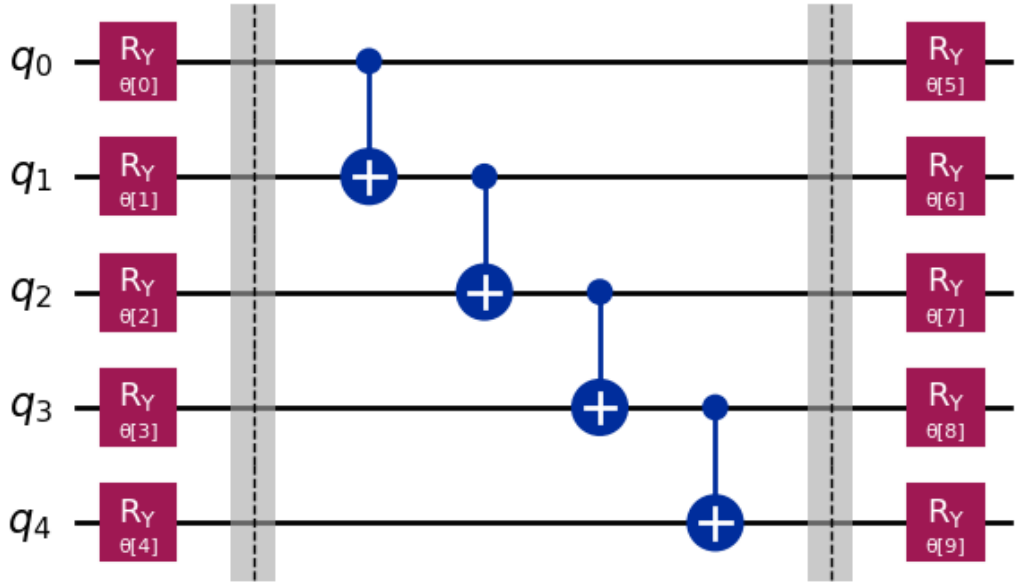
run on current devices, so they have to contain few two qubits gates and be relatively shallow: these ansätze are called hardware-efficient.

What we are going to consider is one of the so-called "hardware-efficient ansätze". The system does not contain many qubits, so our trial ansatz will be very simple : a layer of rotations around the y-axis followed by CNOTs and again a layer of rotations. This simple structure can be easily extended both in depth (adding more CNOTs and rotation) and in width, to study bigger system, therefore is widely used.

[67]:
```
def variational_ansatz(n_qubits,params):

    qc = QuantumCircuit(n_qubits)


    for i in range(n_qubits):
        qc.ry(params[i],i)
    qc.barrier()
    for i in range(n_qubits-1):
        qc.cx(i,i+1)
    qc.barrier()
    for i in range(n_qubits):
        qc.ry(params[n_qubits+i],i)
    return qc
```

[68]:
```
## Let's plot an example

params = ParameterVector(' ',10)
variational_ansatz(5,params).draw("mpl")
```

[68]:



Evaluating energy and the gradient Now we focus on the core part of the VQE algorithm: the evaluation of the energy and of its gradient on the quantum hardware.

Qiskit has built-in methods for this, but we are going to implement ours from scratch.

First, we want a circuit to measure the expecation value of the Hamiltonian on our trial state

[69]:
```
def energy(quantum_circuit,hami,parameters,estimator):
    ## This function evaluates the energy during a VQE calculation
    return estimator.run(quantum_circuit,hami,parameters).result().values[0]
```

[70]:
```
## Example energy estimation
param_vec = ParameterVector(" ",4)
q_circuit = variational_ansatz(2,param_vec)
estimator = Estimator()
```

```
/tmp/ipykernel_1114631/3923375788.py:4: DeprecationWarning: The class
``qiskit.primitives.estimator.Estimator`` is deprecated as of qiskit 1.2. It
will be removed no earlier than 3 months after the release date. All
implementations of the `BaseEstimatorV1` interface have been deprecated in favor
of their V2 counterparts. The V2 alternative for the `Estimator` class is
`StatevectorEstimator`.
  estimator = Estimator()
```

[71]:
```
energy(q_circuit,qubit_op,np.random.rand(4),estimator)
```

[71]:
```
np.float64(-0.604828087212575)
```

Now things get interesting! In order to optimize our parameters θ, a quantum computer has to give to classical optimizer important information such as first- or second-order derivatives! But how can we measure the derivative wrt to a specific parameter on a quantum circuit?

$$\frac{\partial}{\partial\theta_i}L(\theta) = \frac{\partial}{\partial\theta_i}\langle\psi(\theta)|H|\psi(\theta)\rangle$$

many different methods have been proposed recently, in this case we are going to see a method that is called *parameter shift*:

- assume that every parametrized gate is of the form

  $$U_j(\theta_j) = e^{-i\theta_j G_j} = \cos(\theta_j)\mathbf{I} - i\sin(\theta_j)G_j$$

  where G_j is an operator such that $G_j^2 = \mathbf{I}$
- then the derivative can be expressed as

  $$\frac{\partial}{\partial\theta_i}L(\theta) = \frac{L(\theta + e_i s) - L(\theta - e_i s)}{2\sin(s)}$$

  where $s \in \mathbf{R}$ and e_i indicates the versor in the i-th direction. In our case we are going to consider $s = \frac{\pi}{2}$.

This means that to calculate the gradient with N_p parameters, we have to measure H on $2N_p$ different circuits, but it's possible!

Note that the assumption we made is quite general: G_j could be every tensor product of Pauli operators

[72]:
```
#useful function to shift the parameters
def ei(i,n):
    vi = np.zeros(n)
    vi[i] = 1.0
    return vi[:]


def gradient(quantum_circuit,hami,parameters,estimator):


    ## This function evaluates the gradient during a VQE calculation

    nparameters = len(parameters)
    g = np.zeros(nparameters)

    # First create the values dictionary
    values_dict = []


    # Then the values for the gradient
    for i in range(nparameters):
        values_dict.append((parameters + ei(i,nparameters)*np.pi/2.0).tolist())
        values_dict.append((parameters - ei(i,nparameters)*np.pi/2.0).tolist())


    results = []
    for values in values_dict:
        res = energy(quantum_circuit,hami,values,estimator)
        results.append(res)

    for i in range(nparameters):
        rplus  = results[2*i]
        rminus = results[2*i+1]
        # G       = (Ep - Em)/2
        # var(G) = var(Ep) * (dG/dEp)**2 + var(Em) * (dG/dEm)**2
        g[i] = (rplus-rminus)/2

    return g
```

The VQE algorithm Now we declare a function to repeat iteratively the procedure of measuring energy, its gradient and then optimizing the parameters using a standard gradient descent technique,

namely

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta L(\theta)$$

with $\eta \in \mathbf{R}$ as the learning rate.

[73]:
```
def VQE(operator=None, ansatz=None, init_params=None, init_state=None,␣
 ↪estimator=None, max_iter=100, learning_rate=0.1):

    # initialize the useful quantities
    curr_params = init_params

    # Create the variational circuit to measure
    nparameters = len(init_params)
    params_vec      = ParameterVector(' ',nparameters)
    circ_wfn        = init_state.compose(ansatz(n_qubits,params_vec))

    log = {'energies': [], 'gradients': []}

    for i in range(max_iter):

        ## Measure energy and gradient
        E = energy(circ_wfn,operator,curr_params,estimator)
        g = gradient(circ_wfn,operator,curr_params,estimator)

        log['energies'].append(E)
        log['gradients'].append(g)


        # Update the parameters
        curr_params = curr_params - learning_rate*g
    return log
```

[ ]:
```
```

Perform the optimization on a single configuration

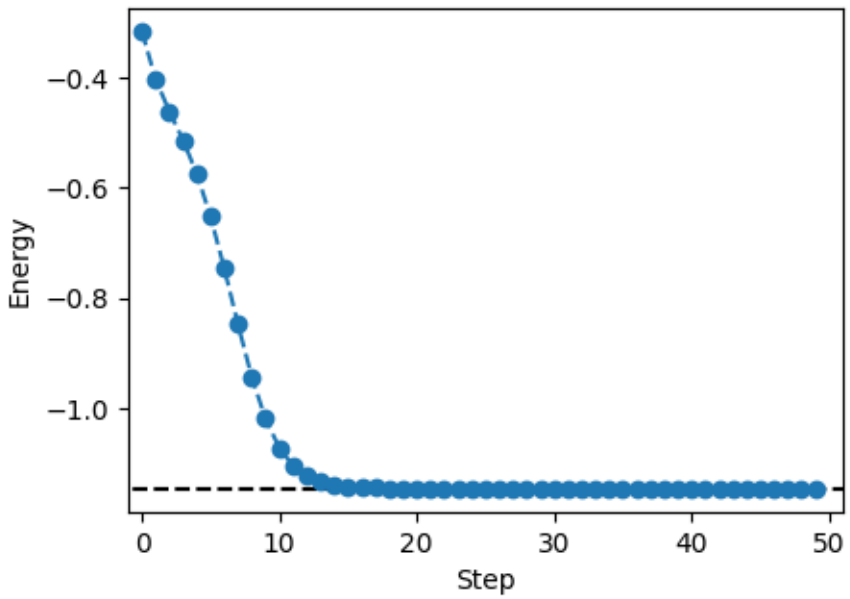
[74]:
```
molecule = MoleculeInfo(["H", "H"], [(0.0, 0.0, 0.0), (0.0, 0.0, 0.735)],␣
 ↪charge=0, multiplicity=1)
driver = PySCFDriver.from_molecule(molecule, basis="sto6g")
problem = driver.run()
hamiltonian = problem.second_q_ops()[0]
nucl_shift = problem.nuclear_repulsion_energy
mapper = ParityMapper(num_particles=[1,1]) # Parity mapper requires the spin up␣
 ↪and down electrons
qubit_op = mapper.map(hamiltonian)
```

[75]:
```
# Now create the circuit
n_qubits = qubit_op.num_qubits
init_params = np.random.rand(2*n_qubits)
```

```
init_state = HartreeFock(num_spatial_orbitals=2, num_particles=[1,1],␣
 ↪qubit_mapper=mapper)


n_reps = 50
learning_rate  = 0.5
estimator = Estimator()

res = VQE(qubit_op,variational_ansatz,init_params,init_state,estimator,n_reps,␣
 ↪learning_rate)
```

```
/tmp/ipykernel_1114631/1692613447.py:9: DeprecationWarning: The class
``qiskit.primitives.estimator.Estimator`` is deprecated as of qiskit 1.2. It
will be removed no earlier than 3 months after the release date. All
implementations of the `BaseEstimatorV1` interface have been deprecated in favor
of their V2 counterparts. The V2 alternative for the `Estimator` class is
`StatevectorEstimator`.
  estimator = Estimator()
```

[76]:
```
steps = list(range(n_reps))

plt.figure(figsize=(4.8,3.4),dpi=100)
plt.errorbar(steps,np.
 ↪array(res['energies'])+nucl_shift,marker='o',linestyle='dashed',label="VQE")
plt.hlines(-1.145, xmin= -10, xmax= 1000,label='Theoretical',linestyle␣
 ↪='dashed',color='black')
plt.xlabel('Step')
plt.xlim(xmin=-1,xmax=51)
plt.ylabel('Energy')

plt.show()
```



Quantum simulation Now we are going to perform the VQE algorithm for every configuration of the H_2 molecule.

[77]:
```
# Prepare the backend
estimator = Estimator()
vqe_distances = list(np.arange(0.3, 1.5, .1)) + list(np.arange(1.5, 4.0, .2))

energies["VQE_"+basis] = []
energies["VQE_error"+basis] = []

# Repeat for every r
for (i,r) in enumerate(vqe_distances):
    print("\n=======================================")
    print(f" DISTANCE: {r}")
    print("=======================================\n")

    molecule    = MoleculeInfo(["H", "H"], [(0.0, 0.0, 0.0), (0.0, 0.0, r)],␣
 ↪charge=0, multiplicity=1)
    driver      = PySCFDriver.from_molecule(molecule, basis="sto6g")
    problem     = driver.run()
    nucl_shift  = problem.nuclear_repulsion_energy


    # Build the (reduced) qubit operator
    hamiltonian = problem.second_q_ops()[0]
    mapper     = ParityMapper(num_particles=[1,1])
```

```
    qubit_op    = mapper.map(hamiltonian)

    # Now create the circuit
    n_qubits    = qubit_op.num_qubits
    params      = np.random.rand(2*n_qubits)
    init_state = HartreeFock(num_spatial_orbitals=2, num_particles=[1,1],␣
 ↪qubit_mapper=mapper)



    #Run the algorithm
    n_reps  = 150
    lr      = 0.5
    res     =␣
 ↪VQE(qubit_op,variational_ansatz,params,init_state,estimator,n_reps,lr)

    print("Final energy: ",res['energies'][-1]+nucl_shift)

    energies["VQE_"+basis].append(res['energies'][-1]+nucl_shift)
```

```
/tmp/ipykernel_1114631/1487965590.py:2: DeprecationWarning: The class
``qiskit.primitives.estimator.Estimator`` is deprecated as of qiskit 1.2. It
will be removed no earlier than 3 months after the release date. All
implementations of the `BaseEstimatorV1` interface have been deprecated in favor
of their V2 counterparts. The V2 alternative for the `Estimator` class is
`StatevectorEstimator`.
  estimator = Estimator()


========================================
 DISTANCE: 0.3
========================================

Final energy:  -0.613030967938903

========================================
 DISTANCE: 0.4
========================================

Final energy:  -0.9251782194739735

========================================
 DISTANCE: 0.5
========================================

Final energy:  -1.0653851727714747
```

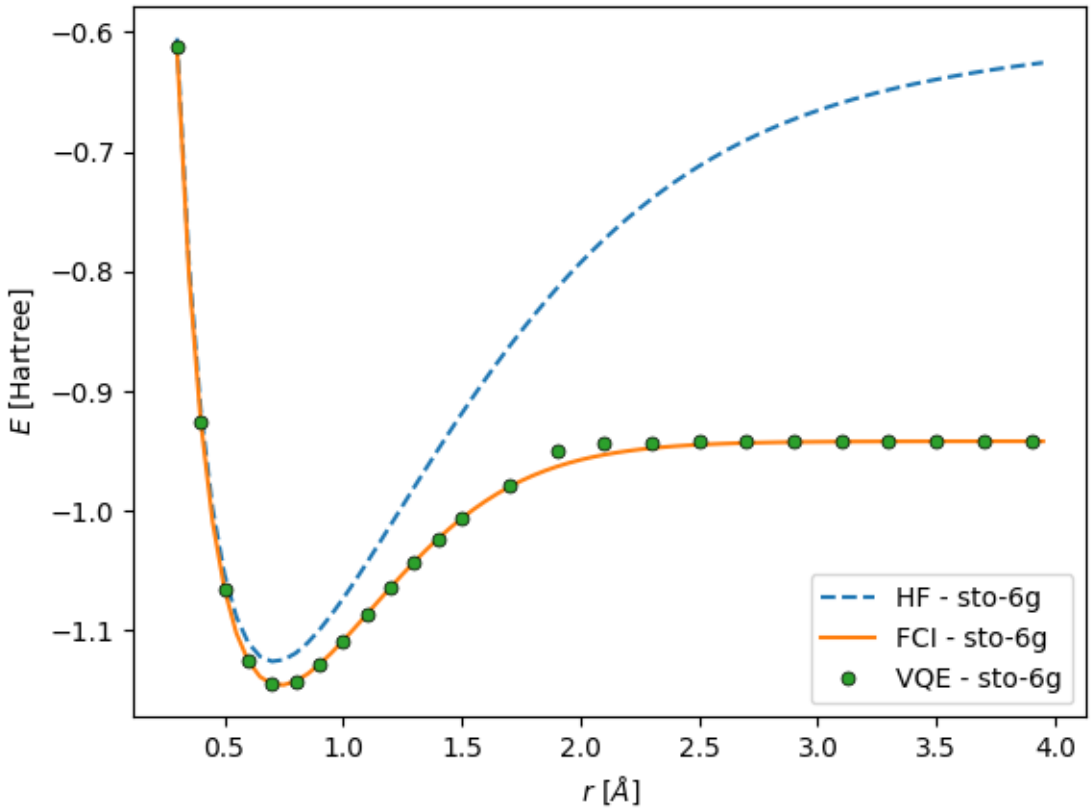
```
=====================================
 DISTANCE: 0.6000000000000001
=====================================

Final energy:  -1.1255968661617182

=====================================
 DISTANCE: 0.7000000000000002
=====================================

Final energy:  -1.1449790794971126

=====================================
 DISTANCE: 0.8000000000000003
=====================================

Final energy:  -1.142613162302784

=====================================
 DISTANCE: 0.9000000000000001
=====================================

Final energy:  -1.1286542684480338

=====================================
 DISTANCE: 1.0000000000000002
=====================================

Final energy:  -1.1088730601683296

=====================================
 DISTANCE: 1.1000000000000003
=====================================

Final energy:  -1.0867110009039278

=====================================
 DISTANCE: 1.2000000000000004
=====================================

Final energy:  -1.0642957382564404

=====================================
 DISTANCE: 1.3000000000000005
=====================================

Final energy:  -1.0429690681419195
```

```
=====================================
 DISTANCE: 1.4000000000000004
=====================================

Final energy:  -1.0235629424238053

=====================================
 DISTANCE: 1.5
=====================================

Final energy:  -1.0062562562577946

=====================================
 DISTANCE: 1.7
=====================================

Final energy:  -0.9783081492049671

=====================================
 DISTANCE: 1.9
=====================================

Final energy:  -0.9500298750837197

=====================================
 DISTANCE: 2.0999999999999996
=====================================

Final energy:  -0.9432833196061

=====================================
 DISTANCE: 2.3
=====================================

Final energy:  -0.9433191083798482

=====================================
 DISTANCE: 2.5
=====================================

Final energy:  -0.9419097906069627

=====================================
 DISTANCE: 2.6999999999999997
=====================================

Final energy:  -0.942189995389784
```

```
=====================================
 DISTANCE: 2.8999999999999995
=====================================

Final energy:  -0.9420684807899089

=====================================
 DISTANCE: 3.0999999999999996
=====================================

Final energy:  -0.9419893547684488

=====================================
 DISTANCE: 3.3
=====================================

Final energy:  -0.9420080058569545

=====================================
 DISTANCE: 3.4999999999999996
=====================================

Final energy:  -0.9420661786635257

=====================================
 DISTANCE: 3.6999999999999993
=====================================

Final energy:  -0.9420784558024419

=====================================
 DISTANCE: 3.8999999999999995
=====================================

Final energy:  -0.9420703057745775
```

Plot the final results

[78]:
```
plt.plot(distances,energies["HF_"+basis],linestyle="dashed",label="HF - "+basis)
plt.plot(distances,energies["FCI_"+basis],label="FCI - "+basis)
plt.errorbar(vqe_distances,energies["VQE_"+basis],label="VQE -␣
  ↪"+basis,linestyle="",marker="o",markersize=5,mew=0.5,mec="black")

plt.xlabel(r"$r$ [$\AA$]")
plt.ylabel(r"$E$ [Hartree]")
plt.legend()
plt.show()
```


−0.6
−0.7
−0.8
−0.9
−1.0
−1.1
E [Hartree]
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
r [Å]
HF - sto-6g
FCI - sto-6g
VQE - sto-6g

[ ]: