VQE Screening 3

September 18, 2020

1 VQE Screening 3

In [1]:
```
scaffold_codeXX = """
// Ref[2] https://www.asc.ohio-state.edu/perry.6/p5501_sp17/articles/quantum_gates.pdf
// Ref[3] https://arxiv.org/pdf/1901.00015.pdf

const double alpha0 = 3.14159265359;
const double b = 0*3.14159265359, c = 0*3.14159265359, d = 0*3.14159265359;
// Note that I kept b and c equal to zero and treated d as the parameter of the circuit

module initialRotations(qbit reg[2]) {
  H(reg[0]);

  Rx(reg[1], alpha0); // Start of creating Psi_in
  CNOT(reg[1], reg[2]);
  H(reg[1]); // End of creating Psi_in
  // Note that Psi_in is entangled

  Rz(reg[1], d); // This is a simple parameterised unitary
  (which is also the exponential of a Pauli matrix)

  // This the controlled version of the parameterised unitary
  Rz(reg[1], (d-b)/2); // Start of Fig. 2.29 of Ref[2]
  with gate parameters b, c, d defined above
  CNOT(reg[0], reg[1]);
  Rz(reg[1], -(d+b)/2);
  Ry(reg[1], -c/2);
  CNOT(reg[0], reg[1]);
  Ry(reg[1], c/2);
  Rz(reg[1], b); // End of Fig. 2.29 of Ref[2]

}

module entangler(qbit reg[2]) {
  H(reg[1]);
  CNOT(reg[1], reg[2]);
```

```
  H(reg[2]);
  CNOT(reg[2], reg[1]);
}


module prepareAnsatz(qbit reg[2]) {
  initialRotations(reg);
  //entangler(reg);
  // I missed out the entangling gates because Psi_in is already entangled
}

module measure(qbit reg[2], cbit result[3]) {
  result[0] = MeasX(reg[0]);
  result[1] = MeasX(reg[1]);

  S(reg[0]);
  H(reg[0]);

  result[2] = MeasZ(reg[2]);
}

int main() {
  qbit reg[3];
  cbit result[3];

  prepareAnsatz(reg);
  measure(reg, result);

  return 0;
}


"""
```

```
In [2]: scaffold_codeYY = """
        const double alpha0 = 3.14159265359;
        const double b = 0*3.14159265359, c = 0*3.14159265359, d = 0*3.14159265359;
        // Note that I kept b and c equal to zero and treated d as the parameter of the circuit

        module initialRotations(qbit reg[2]) {
          H(reg[0]);

          Rx(reg[1], alpha0); // Start of creating Psi_in
          CNOT(reg[1], reg[2]);
          H(reg[1]); // End of creating Psi_in
          // Note that Psi_in is entangled

          Rz(reg[1], d); // This is a simple parameterised unitary
```

```
  (which is also the exponential of a Pauli matrix)

  // This the controlled version of the parameterised unitary
  Rz(reg[1], (d-b)/2); // Start of Fig. 2.29 of Ref[2]
  with gate parameters b, c, d defined above
  CNOT(reg[0], reg[1]);
  Rz(reg[1], -(d+b)/2);
  Ry(reg[1], -c/2);
  CNOT(reg[0], reg[1]);
  Ry(reg[1], c/2);
  Rz(reg[1], b); // End of Fig. 2.29 of Ref[2]

}

module entangler(qbit reg[2]) {
  H(reg[1]);
  CNOT(reg[1], reg[2]);

  H(reg[2]);
  CNOT(reg[2], reg[1]);
}


module prepareAnsatz(qbit reg[2]) {
  initialRotations(reg);
  //entangler(reg);
  // I missed out the entangling gates because Psi_in is already entangled
}

module measure(qbit reg[2], cbit result[3]) {
  Rx(reg[1], 1.57079632679);
  Rx(reg[2], 1.57079632679);

  result[0] = MeasZ(reg[0]);
  result[1] = MeasZ(reg[1]);

  S(reg[0]);
  H(reg[0]);
  result[2] = MeasZ(reg[2]);
}

int main() {
  qbit reg[3];
  cbit result[3];

  prepareAnsatz(reg);
  measure(reg, result);
```

```
  return 0;
}
"""
```

```
In [3]: scaffold_codeZZ = """
const double alpha0 = 3.14159265359;
const double b = 0*3.14159265359, c = 0*3.14159265359, d = 0*3.14159265359;
// Note that I kept b and c equal to zero and treated d as the parameter of the circuit

module initialRotations(qbit reg[2]) {
  H(reg[0]);

  Rx(reg[1], alpha0); // Start of creating Psi_in
  CNOT(reg[1], reg[2]);
  H(reg[1]); // End of creating Psi_in
  // Note that Psi_in is entangled

  Rz(reg[1], d); // This is a simple parameterised unitary
  (which is also the exponential of a Pauli matrix)

  // This the controlled version of the parameterised unitary
  Rz(reg[1], (d-b)/2); // Start of Fig. 2.29 of Ref[2]
  with gate parameters b, c, d defined above
  CNOT(reg[0], reg[1]);
  Rz(reg[1], -(d+b)/2);
  Ry(reg[1], -c/2);
  CNOT(reg[0], reg[1]);
  Ry(reg[1], c/2);
  Rz(reg[1], b); // End of Fig. 2.29 of Ref[2]

}

module entangler(qbit reg[2]) {
  H(reg[1]);
  CNOT(reg[1], reg[2]);

  H(reg[2]);
  CNOT(reg[2], reg[1]);
}


module prepareAnsatz(qbit reg[2]) {
  initialRotations(reg);
  //entangler(reg);
  // I missed out the entangling gates because Psi_in is already entangled
}

module measure(qbit reg[2], cbit result[3]) {
```

```
  result[0] = MeasZ(reg[0]);
  result[1] = MeasZ(reg[1]);

  S(reg[0]);
  H(reg[0]);
  result[2] = MeasZ(reg[2]);
}

int main() {
  qbit reg[3];
  cbit result[3];

  prepareAnsatz(reg);
  measure(reg, result);

  return 0;
}


"""
```

---

2 Executing it

```
In [4]: from scaffcc_interface import ScaffCC
        openqasmXX = ScaffCC(scaffold_codeXX).get_openqasm()
        openqasmYY = ScaffCC(scaffold_codeYY).get_openqasm()
        openqasmZZ = ScaffCC(scaffold_codeZZ).get_openqasm()
        print(openqasmXX)
        print(openqasmYY)
        print(openqasmZZ)
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg reg[3];
creg result[3];
h reg[0];
rx(3.141593e+00) reg[1];
cx reg[1],reg[2];
h reg[1];
rz(0.000000e+00) reg[1];
rz(0.000000e+00) reg[1];
cx reg[0],reg[1];
rz(-0.000000e+00) reg[1];
ry(-0.000000e+00) reg[1];
cx reg[0],reg[1];
ry(0.000000e+00) reg[1];
```

```
rz(0.000000e+00) reg[1];
h reg[0];
measure reg[0] -> result[0];
h reg[1];
measure reg[1] -> result[1];
s reg[0];
h reg[0];
measure reg[2] -> result[2];
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg reg[3];
creg result[3];
h reg[0];
rx(3.141593e+00) reg[1];
cx reg[1],reg[2];
h reg[1];
rz(0.000000e+00) reg[1];
rz(0.000000e+00) reg[1];
cx reg[0],reg[1];
rz(-0.000000e+00) reg[1];
ry(-0.000000e+00) reg[1];
cx reg[0],reg[1];
ry(0.000000e+00) reg[1];
rz(0.000000e+00) reg[1];
rx(1.570796e+00) reg[1];
rx(1.570796e+00) reg[2];
measure reg[0] -> result[0];
measure reg[1] -> result[1];
s reg[0];
h reg[0];
measure reg[2] -> result[2];
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg reg[3];
creg result[3];
h reg[0];
rx(3.141593e+00) reg[1];
cx reg[1],reg[2];
h reg[1];
rz(0.000000e+00) reg[1];
rz(0.000000e+00) reg[1];
cx reg[0],reg[1];
rz(-0.000000e+00) reg[1];
ry(-0.000000e+00) reg[1];
```

```
cx reg[0],reg[1];
ry(0.000000e+00) reg[1];
rz(0.000000e+00) reg[1];
measure reg[0] -> result[0];
measure reg[1] -> result[1];
s reg[0];
h reg[0];
measure reg[2] -> result[2];
```

2.0.1 Execute on a Simulator

In [5]:
```
from qiskit import Aer,QuantumCircuit, execute
Aer.backends()
```

Out[5]:
```
[<QasmSimulator('qasm_simulator') from AerProvider()>,
 <StatevectorSimulator('statevector_simulator') from AerProvider()>,
 <UnitarySimulator('unitary_simulator') from AerProvider()>]
```

In [6]:
```
simulator = Aer.get_backend('qasm_simulator')
vqe_circXX = QuantumCircuit.from_qasm_str(openqasmXX)
vqe_circYY = QuantumCircuit.from_qasm_str(openqasmYY)
vqe_circZZ = QuantumCircuit.from_qasm_str(openqasmZZ)
num_shots = 100000
sim_resultXX = execute(vqe_circXX, simulator, shots=num_shots).result()
sim_resultYY = execute(vqe_circYY, simulator, shots=num_shots).result()
sim_resultZZ = execute(vqe_circZZ, simulator, shots=num_shots).result()

countsXX = sim_resultXX.get_counts()
countsYY = sim_resultYY.get_counts()
countsZZ = sim_resultZZ.get_counts()

expected_valueXX = (countsXX.get('000', 0) - countsXX.get('001', 0)
 - countsXX.get('010', 0) + countsXX.get('011', 0) - countsXX.get('100', 0)
  + countsXX.get('101', 0) + countsXX.get('110', 0) - countsXX.get('111', 0))
  / num_shots
expected_valueYY = (countsYY.get('000', 0) - countsYY.get('001', 0)
 - countsYY.get('010', 0) + countsYY.get('011', 0) - countsXX.get('100', 0)
  + countsXX.get('101', 0) + countsXX.get('110', 0) - countsXX.get('111', 0))
  / num_shots
expected_valueZZ = (countsZZ.get('000', 0) - countsZZ.get('001', 0)
 - countsZZ.get('010', 0) + countsZZ.get('011', 0) - countsXX.get('100', 0)
  + countsXX.get('101', 0) + countsXX.get('110', 0) - countsXX.get('111', 0))
  / num_shots
```

```
        expected_value = 0.5 - 0.5 * expected_valueXX - 0.5 * expected_valueYY +
         0.5 * expected_valueZZ
        print('The derivative of the lowest eigenvalue with respect to rotation angle, which is :
        %s' % expected_value)

        #print(expected_valueXX)
        #print(expected_valueYY)
        #print(expected_valueZZ)
```

```
The derivative of the lowest eigenvalue with respect to rotation angle,
 which is : -0.00041000000000002146
```

---

3 Circuit Visualization

```
In [7]: from qiskit.tools.visualization import circuit_drawer
        circuit_drawer(vqe_circXX, scale=.4)
```

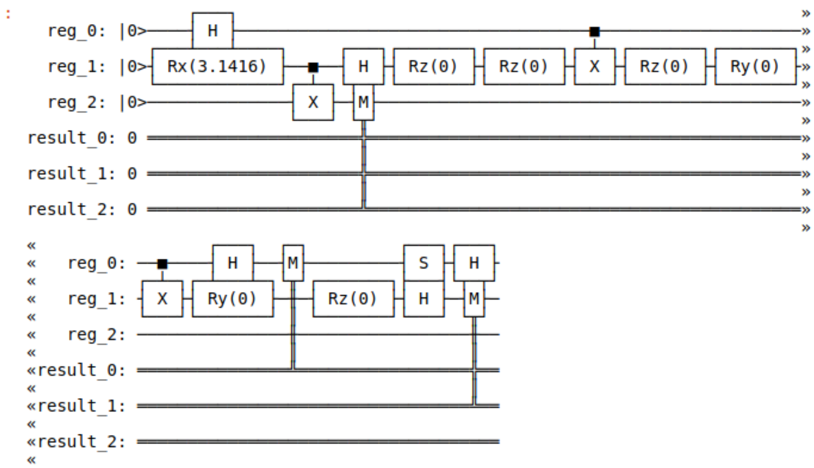
```
Out[7]: <qiskit.visualization.text.TextDrawing at 0x7f789c582eb8>
```

```
In [8]: from qiskit.tools.visualization import circuit_drawer
        circuit_drawer(vqe_circYY, scale=.4)
```

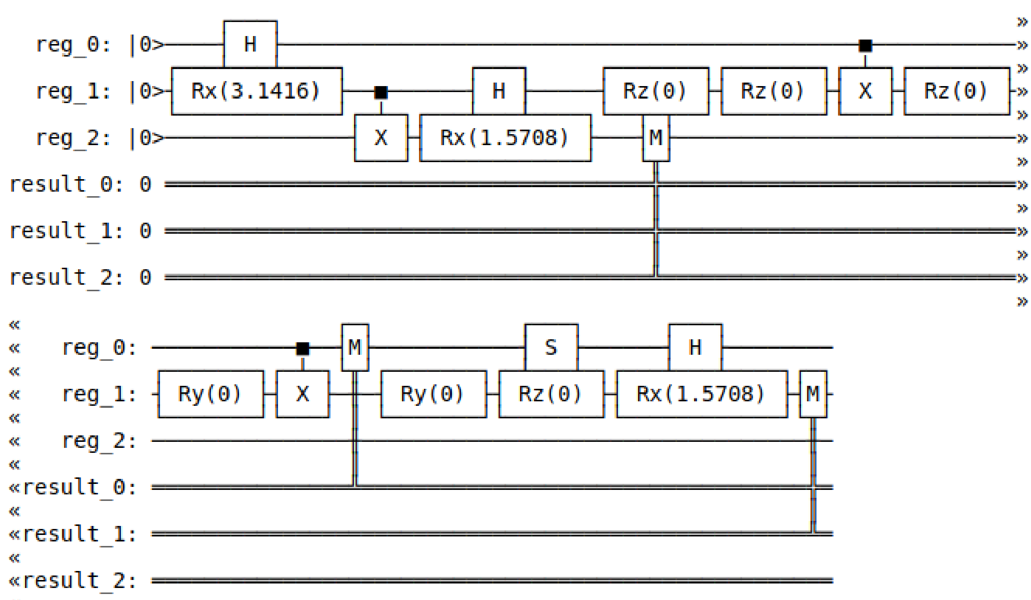
```
Out[8]: <qiskit.visualization.text.TextDrawing at 0x7f784560bf28>
```

```
In [9]: from qiskit.tools.visualization import circuit_drawer
        circuit_drawer(vqe_circZZ, scale=.4)
```

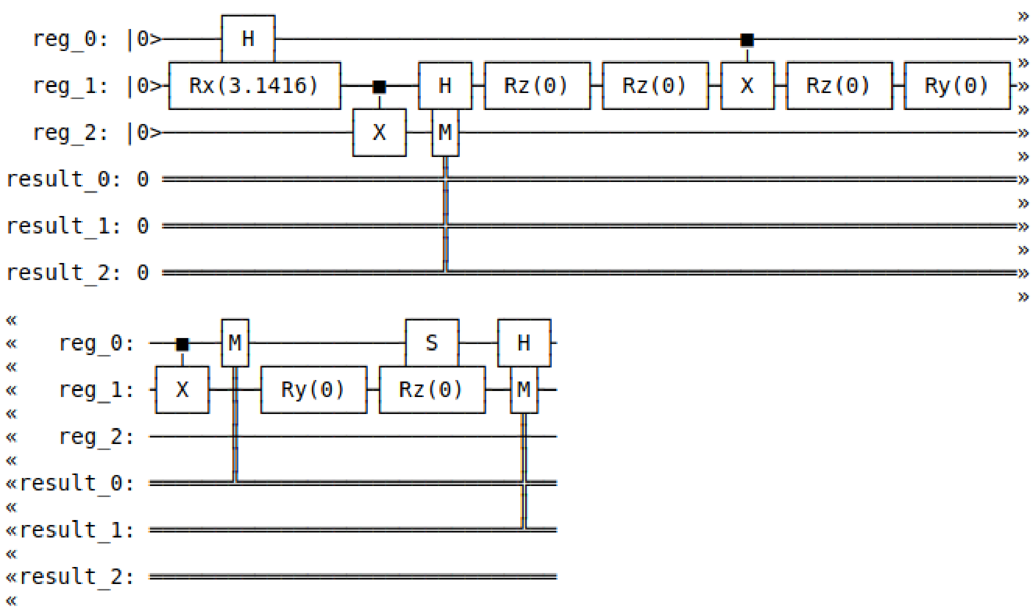
```
Out[9]: <qiskit.visualization.text.TextDrawing at 0x7f784560be80>
```



vqe_circXX



vqe_circYY


reg_0: |0>
reg_1: |0>
reg_2: |0>
result_0: 0
result_1: 0
result_2: 0
H
Rx(3.1416)
X
H
M
Rz(0)
Rz(0)
X
Rz(0)
Ry(0)
reg_0:
reg_1:
reg_2:
result_0:
result_1:
result_2:
X
M
Ry(0)
S
Rz(0)
H
M

vqe_circZZ