Truncated Qubit Operator in VQD

Jupyter Notebook Initialization

User-defined commands are hidden in this markdown cell.

Introduction

The cost of the of the VQD algorithm grows with the number of terms in the transformed Pauli operator. To alliate the cost, we truncate the full operator and analyze the accuracy. By adding and removing particular terms in the qubit operator, we systematically study the relevance and weight of each term to the computed eigenvalues.

Code Initialization

In [1]:
```
# clear all variables
# %reset -f

# import classical modules
import numpy as np
import matplotlib.pyplot as plt

# import quantum modules
from qiskit.circuit import QuantumCircuit, Parameter
from qiskit.algorithms.optimizers import *
from qiskit.circuit.library import TwoLocal, RealAmplitudes, NLocal, EfficientSU2, QAOAAnsatz
from qiskit.primitives import Estimator, Sampler
from qiskit.opflow import I, X, Y, Z
from qiskit_aer.primitives import Estimator as AerEstimator
from qiskit.algorithms.state_fidelities import ComputeUncompute
from qiskit.algorithms.eigensolvers import VQD
from qiskit.quantum_info import SparsePauliOp
from qiskit.algorithms import NumPyMinimumEigensolver
from qiskit.utils import algorithm_globals

# call helper functions
%run vqe-helpers
```

Defining Problem

Recall that the VQD finds the k^{th} lowest (including negative values) eigenvalues. To return sensible results (i.e., eigenvalues of observable quantities), the Schlogl operator must be transformed into a positive-definite hermitian matrix using the transformation

$$Q_{\text{spd}} = Q \cdot Q^T,$$

whose eigenvalues are squares of the original matrix's eigenvalues, such that

$$\lambda_i^Q = \sqrt{\lambda_i^{Q_{\text{spd}}}}$$

where λ_i^Q are the eigenvalues of the original Schlogl operator and $\lambda_i^{Q_{\text{spd}}}$ are the eigenvalues of the transformed semi-positive definite matrix.

Model Parameters

In [2]:
```
# model parameters
a = 1
b = 1
k1 = 3
k2 = 0.6
k3 = 0.25
k4 = 2.95
V = 2.5

# number of qubits
operator_num_qubits = 2
print("Initializing...\n\n","-->",str(operator_num_qubits)+"-qubit matrix")

# Matrix size should be 2^num_qubits
N = 2**operator_num_qubits
print(" --> Matrix size = ",N,"x",N,"\n...")
```

```
Initializing...

 --> 2-qubit matrix
 --> Matrix size =  4 x 4
...
```

Classical Simulation

In [3]:
```
# birth and death rates
lambda_fn = lambda n: ((a*k1*n*(n-1))/V + b*k3*V)
mu_fn = lambda n: ((k2*n*(n-1)*(n-2))/V**2 + n*k4)

# stochastic matrix Q of dimension NxN
Q = TridiagA(lambda_fn, mu_fn, N)

# semi-posotive definite transformation
Qspd = np.dot(Q,Q.T)

# computing the nth and mth eigenvalues and eigenvectors
eig0, vec0,eig1,vec1 = find_eigenvalues_eigenvectors(Qspd, n=1, m=2, hermitian=True)

# classical eigenvalues
c_value0 = np.real(eig0)
c_value1 = -np.sqrt(np.abs(np.real(eig1)))

#*******************************************
print("Classical eigenvalues (at V = ",np.round(V,3),") are ",np.round(c_value0,4),
      " and", np.round(c_value1,4))
```

```
Classical eigenvalues (at V =  2.5 ) are  0.0  and -2.9992
```

Quantum Simulation

Pauli Decomposition

Here, we decompose the classical Hermitian Scholgl matrix into Pauli operators.

In [4]:
```
# transform hermitian matrix to Pauli operator
qubitOp = SparsePauliOp.from_operator(Qspd)
print(qubitOp)

# we can verify the number of qubits in the operator
# It should be the same as 'operator_num_qubits'
print(f"Number of qubits in operator: {qubitOp.num_qubits}")
```

```
SparsePauliOp(['II', 'IX', 'IZ', 'XI', 'XX', 'XZ', 'YY', 'ZI', 'ZX', 'ZZ'],
              coeffs=[ 80.9925505+0.j, -63.392238 +0.j,   8.001875 +0.j,   9.845625 +0.j,
 -27.4459375+0.j,  -8.001875 +0.j, -27.4459375+0.j, -52.455363 +0.j,
  52.455363 +0.j, -27.4459375+0.j])
Number of qubits in operator: 2
```

Truncated Pauli Operator

In [5]:
```
# length of operator
operator_length = len(qubitOp)

print("Number of Terms in Original Operator: ", operator_length)
```

```
Number of Terms in Original Operator:  10
```

In [6]:
```
# indices representing each term in qubit operator
qubitOp_indices = generate_indices(operator_length)
print("Qubit operator indices:\n ",qubitOp_indices)

# every subset in qubit operator
index_subsets = generate_subsets(qubitOp_indices)
```

```
Qubit operator indices:
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [7]:
```
# number of all possible combinations
total_combinations = count_combinations(operator_length)
print("Total combinations of terms in qubit operator: ",total_combinations)
```

```
Total combinations of terms in qubit operator:  1023
```

In [ ]:
```

```

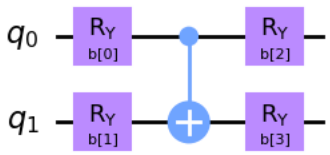
Ansatz: Trial Wavefunction

Next, we define our trial wavefunction as a quantum circuit using the `TwoLocal` function from the `qiskit.circuit` library. The specific anasatz we use here is similar to the one used for the Hydrogen molecule. So, we are not too optimistic about our results. Let's see what such an ansatz gives us...

In [8]:

```
# number of qubits for the ansatz: how sophisticated do you want it to be?
ansatz_num_qubits = operator_num_qubits

# Define the parameters
a = Parameter('a')
b = Parameter('b')
c = Parameter('c')

# ansatzes
ansatz1 = TwoLocal(ansatz_num_qubits, rotation_blocks=['ry'], entanglement="full",
           entanglement_blocks='cx', reps=1, parameter_prefix='a')
ansatz1.assign_parameters({ansatz1.parameters[i]: a for i in range(len(ansatz1.parameters))})

ansatz2 = RealAmplitudes(ansatz_num_qubits, entanglement='circular', reps=1, parameter_prefix='b')
ansatz2.assign_parameters({ansatz2.parameters[i]: a for i in range(len(ansatz2.parameters))})

ansatz3 = EfficientSU2(num_qubits=ansatz_num_qubits, reps=2, su2_gates=['ry'],
        entanglement='sca', parameter_prefix='c')
ansatz3.assign_parameters({ansatz3.parameters[i]: a for i in range(len(ansatz3.parameters))})

# compose quantum circuit
ansatz = QuantumCircuit(ansatz_num_qubits)

#ansatz.compose(ansatz1, inplace=True)
ansatz.compose(ansatz2, inplace=True)
#ansatz.compose(ansatz3, inplace=True)

# draw circuit
ansatz.decompose().draw(output='mpl')
```

Out[8]:



In [9]:

```
# number of states to compute and beta parameters
k = 2

# define callbacks to store intermediate steps
callback_counts = []
callback_values = []
callback_steps = []

def callback(eval_count, params, value, meta, step):
    counts.append(eval_count)
    values.append(value)
    steps.append(step)

# eigenvalue arrays
quantum_eigenvalue0 = np.zeros(total_combinations)
quantum_eigenvalue1 = np.zeros(total_combinations)

# keep track of the truncated operators in for loop
truncation_count = 0

#*******************************
for subset in index_subsets:

    counts = []
    values = []
    steps = []

    truncated_qubitOp = qubitOp[subset]
    print(f"\nTruncation {truncation_count}:\n ",truncated_qubitOp)

    # See qiskit documentation for more info on optimizers
    optimizer = P_BFGS()

    # initialize estimator, sampler, and fidelity
    estimator = Estimator()

    # sampler
    sampler = Sampler()

    # fidelity
    fidelity = ComputeUncompute(sampler)

    # instantiate vqd
    vqd = VQD(estimator, fidelity, ansatz, optimizer, k=k, callback=callback)
    result = vqd.compute_eigenvalues(operator = truncated_qubitOp)
    vqd_values = result.eigenvalues

    # store callbacks
    callback_counts.append(counts)
    callback_values.append(values)
    callback_steps.append(steps)

    value0 = np.real(vqd_values[0])  # vqd state 0
    value1 = np.real(vqd_values[1])  # vqd state 1

    # Check for invalid values in value0 and assign np.nan
    value0 = np.where(np.logical_or(np.isnan(value0), value0 < 0.0), np.nan, value0)

    # Check for invalid values in value1 and assign np.nan
    value1 = np.where(np.logical_or(np.isnan(value1), value1 < 0.0), np.nan, value1)

    # Calculate the square root for valid values
    quantum_eigenvalue0[truncation_count] = np.where(value0 >= 0.0, -np.sqrt(value0), np.nan)
    quantum_eigenvalue1[truncation_count] = np.where(value1 >= 0.0, -np.sqrt(value1), np.nan)


    # next operator
    truncation_count+=1
```

```
Truncation 756:
  SparsePauliOp(['II', 'IZ', 'XX', 'XZ', 'YY', 'ZI', 'ZZ'],
                coeffs=[ 80.9925505+0.j,   8.001875 +0.j, -27.4459375+0.j,  -8.001875 +0.j,
 -27.4459375+0.j, -52.455363 +0.j, -27.4459375+0.j])

Truncation 757:
  SparsePauliOp(['IX', 'IZ', 'XX', 'XZ', 'YY', 'ZI', 'ZZ'],
                coeffs=[-63.392238 +0.j,   8.001875 +0.j, -27.4459375+0.j,  -8.001875 +0.j,
 -27.4459375+0.j, -52.455363 +0.j, -27.4459375+0.j])

Truncation 758:
  SparsePauliOp(['II', 'IX', 'IZ', 'XX', 'XZ', 'YY', 'ZI', 'ZZ'],
                coeffs=[ 80.9925505+0.j, -63.392238 +0.j,   8.001875 +0.j, -27.4459375+0.j,
  -8.001875 +0.j, -27.4459375+0.j, -52.455363 +0.j, -27.4459375+0.j])

Truncation 759:
  SparsePauliOp(['XI', 'XX', 'XZ', 'YY', 'ZI', 'ZZ'],
                coeffs=[  9.845625 +0.j, -27.4459375+0.j,  -8.001875 +0.j, -27.4459375+0.j,
 -52.455363 +0.j, -27.4459375+0.j])
```

Eigenvalues from Truncated Operators

Next, let's see how the different truncations affect the accuracy in eigenvalues. We are going to extract the closest N values to classical results.

In [10]:

```
truncation_result = 0
for idx in range(total_combinations):
    print(f"\nVQD Result (Truncation {truncation_result}):\nState 0 = {quantum_eigenvalue0[idx]}\nState 1 = {qua
    truncation_result+=1
```

```
State 0 = -4.464134176831385
State 1 = -5.994351003132365

VQD Result (Truncation 165):
State 0 = nan
State 1 = nan

VQD Result (Truncation 166):
State 0 = nan
State 1 = -8.189976271390528

VQD Result (Truncation 167):
State 0 = nan
State 1 = nan

VQD Result (Truncation 168):
State 0 = -5.058070992054086
State 1 = -5.338988160685072

VQD Result (Truncation 169):
```

Truncations Closest to Classical Result

In [17]:

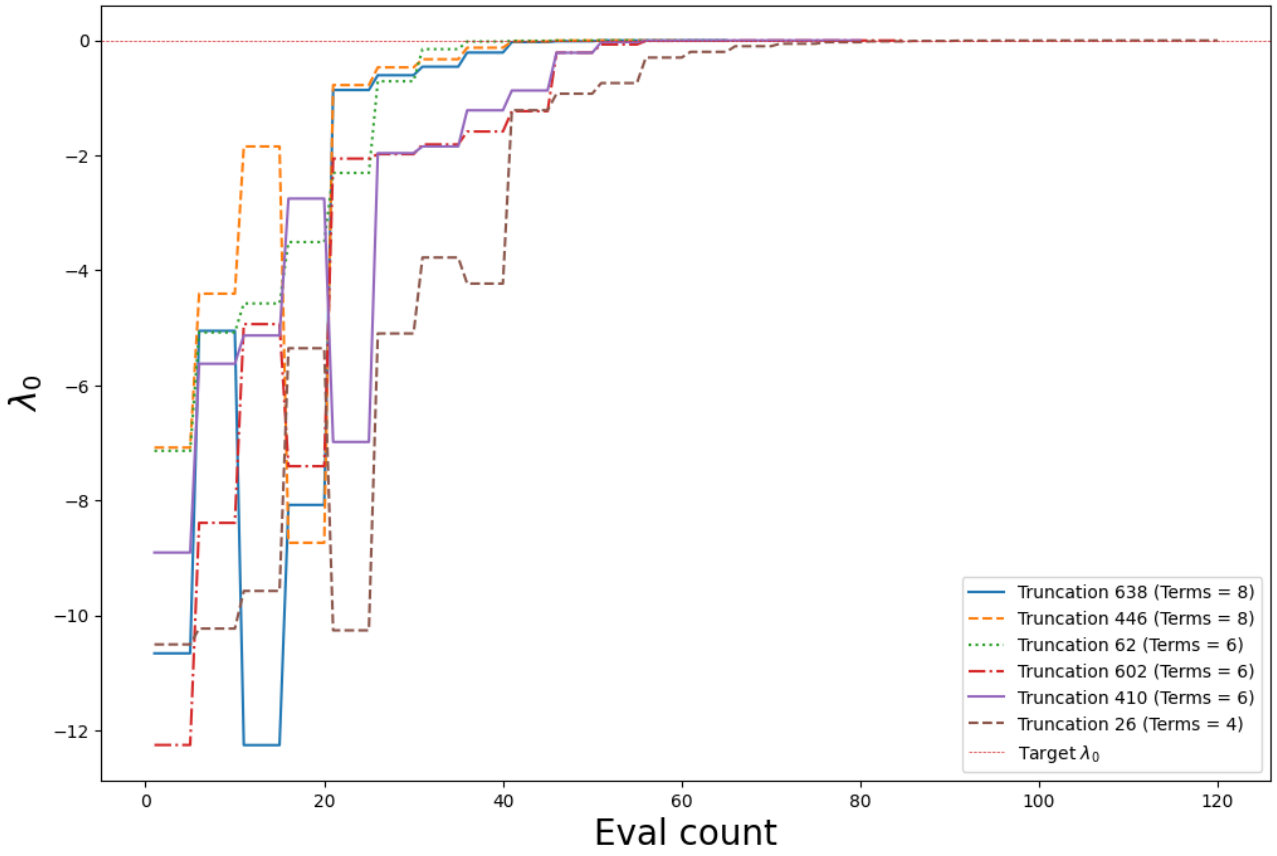
```
# closest to classical results
closest_N = 6

# closest
closest_quantum_values0, closest_indices0 = extract_closest_values(quantum_eigenvalue0, c_value0, closest_N)
closest_quantum_values1, closest_indices1 = extract_closest_values(quantum_eigenvalue1, c_value1, closest_N)

# common indices
find_common_indices(closest_indices0, closest_indices1)

print("\nClosest state 0 indices:\n",closest_indices0)
print("\nClosest state 1 indices:\n",closest_indices1)
print("\nClosest 0 states:\n",closest_quantum_values0)
print("\nClosest 1 states:\n",closest_quantum_values1)
```

```
Closest state 0 indices:
 [638, 446, 62, 602, 410, 26]

Closest state 1 indices:
 [1022, 1010, 982, 546, 70, 462]

Closest 0 states:
 [-6.419738291755496e-08, -6.528735214526733e-08, -1.568045225678781e-07, -1.7444375410916294e-07, -1.855253117915
4269e-07, -1.976862424823888e-07]

Closest 1 states:
 [-2.9992483845360867, -3.072833769744548, -3.072833769744592, -3.384083390526636, -3.3840833905266496, -3.3979840
47667264]
```

Ground State: λ_0

In [23]:

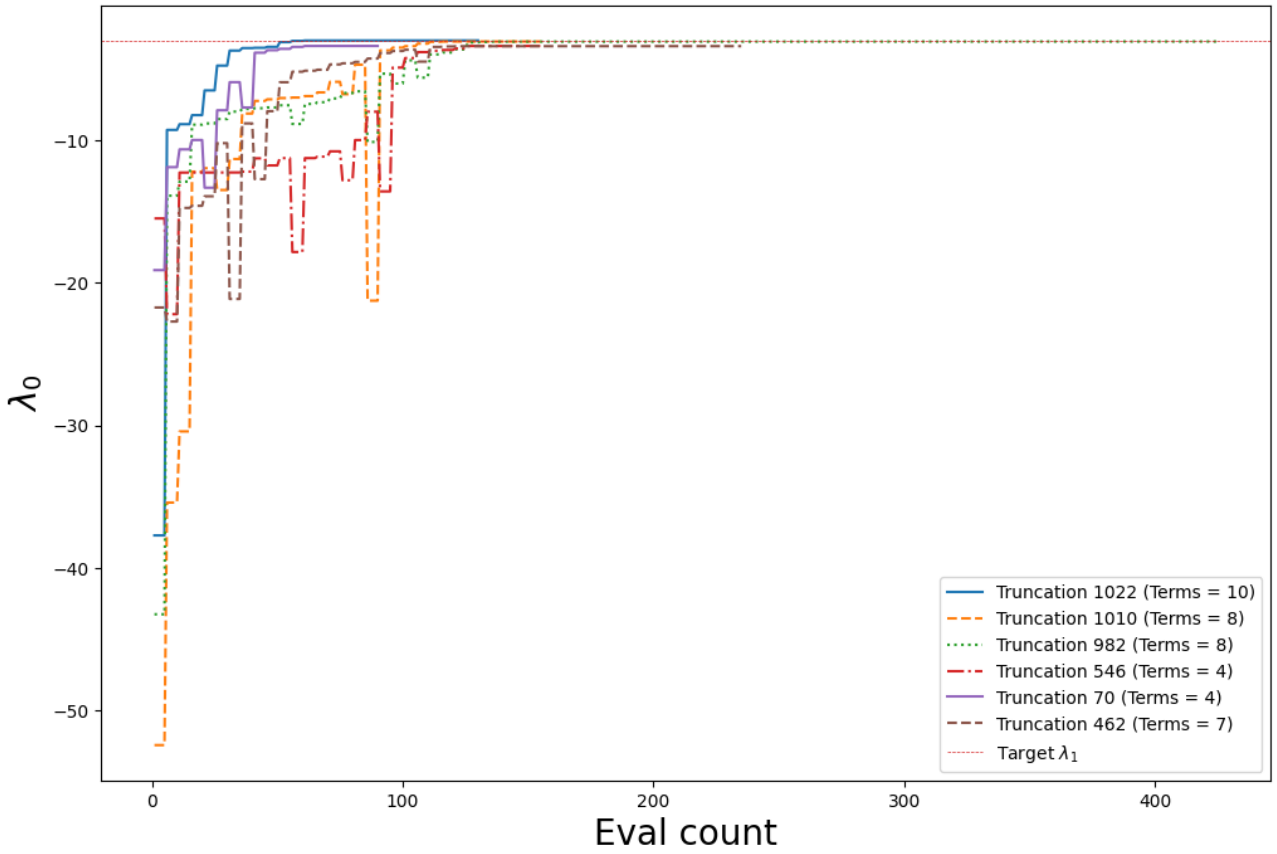
```
(12, 8)

**********

." # Different line styles

ndices0:
len(linestyles)]  # Cycle through line styles

sest_indices]

st_indices]

# Truncate with zero
r2(steps, 2)

sest_indices]

# Extract desired state
ues, _ = split_counts_values(counts, values)

ted_counts)
ted_values)

# Mask for matching elements in steps and twos
# Indices of matching elements

# Use corresponding counts and values
ed_counts), len(_indices))
ed_values), len(_indices))

nts_length]
ues_length]

o.abs(_values)),
=linestyle, label=f"Truncation {closest_indices} (Terms = {len(truncated_qubitOp[index_subsets[closest_indices]])})"

abel="red", ls="--", lw=0.5, label="Target $\lambda_0$")

e=20)
ze=20)
16)

**********
```


0
−2
−4
−6
−8
−10
−12
λ_0
0
20
40
60
80
100
120
Eval count
Truncation 638 (Terms = 8)
Truncation 446 (Terms = 8)
Truncation 62 (Terms = 6)
Truncation 602 (Terms = 6)
Truncation 410 (Terms = 6)
Truncation 26 (Terms = 4)
Target λ_0

First Excited State: λ_1

In [22]:

```
# plotting intermediate steps
plt.rcParams["figure.figsize"] = (12, 8)

#*********************************************

linestyles = ["-", "--", ":", "-."]  # Different line styles

idx = 0
for closest_indices in closest_indices1:
    linestyle = linestyles[idx % len(linestyles)]  # Cycle through line styles

    counts = callback_counts[closest_indices]
    counts = np.asarray(counts)

    steps = callback_steps[closest_indices]
    steps = np.asarray(steps)

    # in steps, replace unwanted state with zero
    twos = replace_digit_with_zero(steps, 1)

    values = callback_values[closest_indices]
    values = np.asarray(values)

    # split counts and values to extract desired state
    _, state1_counts, _, state1_values = split_counts_values(counts, values)

    # plotting state 1
    state1_counts = np.array(state1_counts)
    state1_values = np.array(state1_values)

    mask = np.isin(steps, twos)  # Mask for matching elements in steps and twos
    _indices = np.where(mask)[0]  # Indices of matching elements

    # Use the indices to extract the corresponding counts and values
    counts_length = min(len(state1_counts), len(_indices))
    values_length = min(len(state1_values), len(_indices))

    _counts = state1_counts[:counts_length]
    _values = state1_values[:values_length]

    plt.plot(_counts, -np.sqrt(np.abs(_values)),
             color=f"C{idx}", ls=linestyle, label=f"Truncation {closest_indices} (Terms = {len(truncated_qubitO

    idx+=1

# target state
plt.axhline(y=c_value1, color="tab:red", ls="--", lw=0.5, label="Target $\lambda_1$")

plt.xlabel("Eval count", fontsize=20)
plt.ylabel("$\lambda_0$", fontsize=20)
plt.legend(loc="best", fontsize=10)
plt.show()
#*********************************************
```



In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```