Revisiting QAOA—More Layers and Noisy Execution

Aaron Stringer-Usdan and Tyler Rotello

May 2023

We investigated QAOA earlier in the course as part of a homework assignment, but only for one layer of the algorithm and simulated without noise. In [1] the authors assert that the optimal choice of parameters for $d+1$ layers will never produce a worse result than the optimal choice of parameters for d layers in QAOA. This is because it's always possible to just choose the same parameters for the first d layers and then take both parameters to be zero for the final layer so that the circuit for $d+1$ is equivalent to the circuit for d. We can think of this as the algorithm reaching the adiabatic limit. Since QAOA is discretized, it is probably not the case that the limit is ever reached exactly for a finite number of layers, but the point is that in theory, adding more layers improves the algorithm's performance. However, unless the algorithm is actually able to prepare an exact superposition of only ground states in a finite number of layers, any cost function we might care to choose—either the probability of measuring a ground state or the expectation value of the energy—must approach the optimum asymptotically, and this means diminishing returns on performance increases with more layers added. Each layer of the circuit requires the same number of gates, though, which means the total number of errors on average continues to increase with the number of layers. We would therefore expect that for any finite error rate there should be a point at which adding more layers actually makes the algorithm perform worse, and investigating this is a major motivation. Examining the point at which it no longer becomes viable to add more depth and still expect a return has been the subject of previous work[2] and the time to solution has been shown to grow with noise on QAOA[3]. For QAOA to be a useful NISQ algorithm, we need to know that on noisy hardware with one layer it doesn't perform worse than random guessing and get even worse from there. Although QAOA is not supposed to be single-error fragile, it is valuable to see its performance when multiple errors are almost a certainty.

The problem Hamiltonians used for our investigations are all linear combinations of Z-product operators. This is a reasonable choice because many optimization problems are classical in nature. As such, any valid potential solution to such a problem must be a computational basis state, rather than a superposition. Any classical cost function which is encoded as a Hamiltonian must have only computational basis states as eigenvectors, and must therefore be expressible as a linear combination of Z-products. In particular, we've applied QAOA to MAX-3-XOR-SAT, the same as in the homework assignment earlier in the course. The MAX-3-XOR-SAT problem is NP-Complete, making it an interesting problem to approximate (the 3-XOR-SAT decision problem is in P, but we are concerned with the optimization problem version, which is NP-C). As in the homework, our $H_D = -\sum_{j=1}^{qubits} \sigma_j^x$

and in the algorithm we alternate between applications of H_D and H_P for a single layer, making $\big|\psi\big\rangle \to e^{i\beta H_D} e^{i\alpha H_P} \big|\psi\big\rangle$. In the algorithm we we use the uncomputing trick in order to allow a 3 qubit interaction by applying three C-NOT's with the target qubit being an ancilla and the controls being the three qubits in a given term of the Hamiltonian, applying a Z rotation on the ancilla and then applying the C-NOT's in reverse order as they were originally applied. As before, one clause of the XOR-SAT problem is represented by a three-body Z interaction in the Hamiltonian, with coefficient -1 if the clause is negated and coefficient 1 otherwise. And of course in the output, qubits in the $\big|1\big\rangle$ state correspond to variables assigned to true, and qubits in the $\big|0\big\rangle$ state correspond to variables assigned to false. A clause of the problem is satisfied by a bitstring if and only if the corresponding interaction term in the Hamiltonian has a value of -1 in the basis state which represents the bitstring in question. Because this is an XOR-SAT problem, this occurs when an odd number of the variables in the clause are assigned true, or equivalently when an odd number of the qubits in the interaction are in the $\big|1\big\rangle$ state. This is how, as before, finding a state which minimizes the energy is equivalent to finding a bitstring which maximizes the number of clauses satisfied. However, unlike previously, we have varied the system size between 6 and 9 qubits to see if and how the algorithm's performance is influenced by the number of qubits, which is equivalent to the number of variables appearing in the clauses of the problem instance.

Since adding a qubit that does not interact with the others does not change the number of variables in the corresponding XOR-SAT problem, we need to be certain that every qubit is included in at least one interaction term to actually see the impact of adding more qubits. For Z-product interactions, the ordering of qubits does not matter since Z operators on different qubits commute. Therefore, a unique M-body interaction is determined by an unordered M-tuple of distinct qubit indices. Thus there are in total ${}_NC_M$ such interaction terms possible. If we choose interactions at random (which we did when generating our problem Hamiltonians), the only way to be certain that every qubit is included is by choosing enough interaction terms that it's impossible for any qubit to be excluded. The number of terms that all exclude a single particular qubit is ${}_{N-1}C_M$, so as long as we choose more terms than that, at least one term must include the hypothetically excluded qubit. We chose ${}_{N-1}C_3 + 2$ terms in all of our Hamiltonians. Since we generated our problem instances by randomly choosing interactions/clauses, there is no direct connection between the Hamiltonians we used for different system sizes aside from being instances of the same problem. We did of course use the same Hamiltonian for both noiseless and noisy simulation at a given system size.

The problem instances we used were as follows:

For $N = 6$:

$$(x_0 \veebar x_1 \veebar x_4) \wedge (x_3 \veebar x_4 \veebar x_5) \wedge \neg(x_0 \veebar x_3 \veebar x_4) \wedge \neg(x_0 \veebar x_4 \veebar x_5)$$
$$\wedge\neg(x_0 \veebar x_2 \veebar x_3) \wedge \neg(x_0 \veebar x_3 \veebar x_5) \wedge \neg(x_0 \veebar x_1 \veebar x_3) \wedge (x_1 \veebar x_2 \veebar x_4)$$
$$\wedge(x_0 \veebar x_1 \veebar x_5) \wedge \neg(x_0 \veebar x_2 \veebar x_5) \wedge \neg(x_2 \veebar x_3 \veebar x_4) \wedge (x_1 \veebar x_4 \veebar x_5)$$

For $N = 7$:

$$
\begin{aligned}
&(x_0 \veebar x_1 \veebar x_4) \wedge (x_4 \veebar x_5 \veebar x_6) \wedge \neg(x_0 \veebar x_1 \veebar x_2) \wedge (x_1 \veebar x_2 \veebar x_3) \\
&\wedge(x_1 \veebar x_2 \veebar x_6) \wedge (x_0 \veebar x_2 \veebar x_4) \wedge \neg(x_2 \veebar x_5 \veebar x_6) \wedge \neg(x_2 \veebar x_4 \veebar x_5) \\
&\wedge(x_2 \veebar x_3 \veebar x_4) \wedge \neg(x_0 \veebar x_3 \veebar x_5) \wedge \neg(x_3 \veebar x_4 \veebar x_5) \wedge (x_0 \veebar x_4 \veebar x_5) \\
&\wedge\neg(x_1 \veebar x_3 \veebar x_4) \wedge (x_1 \veebar x_4 \veebar x_6) \wedge \neg(x_0 \veebar x_3 \veebar x_6) \wedge \neg(x_0 \veebar x_1 \veebar x_5) \\
&\wedge\neg(x_2 \veebar x_3 \veebar x_5) \wedge (x_0 \veebar x_4 \veebar x_6) \wedge (x_3 \veebar x_4 \veebar x_6) \wedge \neg(x_0 \veebar x_1 \veebar x_3) \\
&\wedge\neg(x_0 \veebar x_1 \veebar x_6) \wedge \neg(x_0 \veebar x_2 \veebar x_5)
\end{aligned}
$$

Because the number of clauses continues to get even larger for $N = 8$ and $N = 9$, we will just include verbatim what the program generated when choosing these problem instances. It outputs a list of tuples, each tuple of the form (i, j, k, V) and representing a clause/interaction. i, j, k are indices chosen from $\{0, 1, 2, \ldots, N-1\}$ and are distinct from one another, and V is ± 1. Such a tuple indicates that the Hamiltonian includes a term $VZ_iZ_jZ_k$, or equivalently that the XOR-SAT instance includes a clause $(x_i \veebar x_j \veebar x_k)$ for $V = 1$ or $\neg(x_i \veebar x_j \veebar x_k)$ for $V = -1$. These problem instances are as follows:

For $N = 8$:

```
[(0, 5, 7, 1), (2, 3, 6, 1), (0, 4, 6, 1), (3, 5, 6, -1),
(0, 4, 5, 1), (0, 3, 6, 1), (0, 1, 7, 1), (0, 4, 7, -1),
(1, 5, 6, 1), (3, 6, 7, -1), (1, 6, 7, 1), (0, 3, 5, 1),
(0, 1, 2, -1), (1, 3, 7, -1), (1, 2, 5, 1), (1, 4, 5, 1),
(1, 4, 6, -1), (1, 2, 7, -1), (1, 2, 6, -1), (1, 3, 4, 1),
(2, 5, 6, 1), (0, 3, 4, 1), (1, 5, 7, -1), (0, 2, 6, -1),
(0, 6, 7, -1), (2, 3, 4, -1), (0, 2, 3, -1), (3, 4, 5, -1),
(1, 3, 6, 1), (1, 4, 7, -1), (2, 6, 7, -1), (0, 5, 6, -1),
(0, 2, 4, 1), (4, 6, 7, -1), (1, 2, 3, -1), (5, 6, 7, -1),
(1, 2, 4, -1)]
```

For $N = 9$:

```
[(3, 4, 7, 1), (0, 6, 8, 1), (2, 4, 7, -1), (1, 7, 8, 1),
(0, 5, 6, -1), (2, 3, 6, -1), (1, 3, 4, -1), (3, 4, 5, -1),
(1, 2, 5, -1), (3, 7, 8, 1), (0, 1, 3, -1), (1, 4, 7, -1),
(3, 6, 8, 1), (4, 5, 6, -1), (0, 5, 7, -1), (2, 5, 6, 1),
(0, 1, 2, 1), (3, 5, 8, -1), (2, 5, 8, -1), (0, 3, 6, -1),
(0, 3, 7, 1), (1, 6, 7, -1), (0, 6, 7, -1), (1, 2, 7, -1),
(1, 4, 6, 1), (1, 2, 4, -1), (3, 5, 6, -1), (3, 4, 8, 1),
(4, 6, 7, 1), (0, 1, 8, -1), (1, 5, 6, -1), (0, 4, 6, 1),
(1, 5, 7, 1), (0, 1, 4, 1), (0, 2, 8, -1), (2, 3, 7, 1),
(0, 2, 3, 1), (0, 1, 6, 1), (6, 7, 8, 1), (0, 2, 7, -1),
(5, 7, 8, 1), (3, 6, 7, -1), (5, 6, 7, -1), (1, 2, 6, 1),
(2, 5, 7, 1), (1, 3, 8, -1), (2, 4, 6, -1), (0, 4, 8, -1),
(0, 1, 7, 1), (0, 1, 5, -1), (4, 5, 8, 1), (2, 4, 5, -1),
(1, 2, 8, -1), (0, 5, 8, -1), (1, 2, 3, 1), (3, 4, 6, -1),
(2, 3, 8, -1), (2, 6, 8, 1)]
```

In addition to varying system size, we compared the effects of adding more layers to the QAOA circuit at each system size, both with noise and without. With more layers, it becomes inefficient to perform a grid search over the algorithm parameters α and β, as the number of grid search points needed will grow exponentially in the number of layers. Additionally, it is not necessarily efficient in terms of actual runtimes (as opposed to asymptotic behavior) to use knowledge of the problem to determine the parameters either, contrary to the claim made in [1]. While it is true that for the example of MaxCut on graphs of bounded degree, it is possible to do classical preprocessing which does not increase in cost with the overall size of the graph, that preprocessing seems to require checking if pairs of subgraphs are isomorphic. Since there's no known efficient algorithm for checking isomorphisms (this, too, is an NP-C problem), this is at least exponential in the size of the subgraphs. Since the size of the subgraphs is directly tied to the QAOA circuit depth, and moreover grows faster than linear in the circuit depth, this can get very inefficient very quickly. It should be noted that such preprocessing *can* be worthwhile for one-layer circuits, as illustrated in [1]. Since we can't rely on grid search or knowledge of the problem for our deeper circuits, we have used a gradient descent approach to optimize the parameters of all layers of the algorithm. Ideally, we would want to optimize for the probability of measuring one of the true ground states of the problem Hamiltonian, and for this small problem, we could actually directly solve for the ground state and then optimize parameters that way. However, the practical use-case of the algorithm is when the true ground state is both unknown and intractable to find directly, so this would not be a good demonstration of the algorithm's performance. Instead, we used the expectation value of the problem Hamiltonian in the final state of the circuit as the cost function for the gradient descent; the expectation value is generally accessible even when minimizing a Hamiltonian with unknown ground state on a real quantum device, and so better reflects how the algorithm will actually perform. We performed gradient descent only using our ideally-simulated circuits, in order to find the best parameters possible. The initial guess for the optimization was informed by the observation that, in the grid search over just one layer, there was consistently a minimum for the expectation value of many Hamiltonians in the region near $\alpha \sim -0.1$ and $\beta \sim 0.1$. Once an optimal set of parameters for the combination of problem Hamiltonian and layers was found, we used that same set of parameters to run a noisy version of the same circuit, and calculate the expectation value from the final state of that circuit for comparison.

The results of our circuit can be seen in Figure 1, where the x-axis represents the number of layers and y-axis the Expectation Value, there are two lines the 'Best EV' is from the noiseless circuit and 'Average Noisy EV' is the noisy circuit. In the figure it is evident that for a noiseless circuit, as the number of layers increases the expectation value gets better. This is to be expected as the more layers generally the better the results should be. In the noisy circuit, the expectation values do not get better with increased number of layers. This is expected due to the prevalence of noise in our simulation, the more layers in our circuit the more gates and so more room for error. This is also consistent with the results found in [2], where the authors found if the noise rate is too high, there is no gain in performance from having a deeper circuit. It is also expected that with an increased system size that the expectation values will get more negative, corresponding to the maximum number of satisfiable clauses typically increasing the more clauses there are; this is seen in the results from the noiseless circuit. In the noisy circuit it is seen that increased system size has

relatively no impact, this is probably due to more gates with increased system sizes and so the impact is offset by gate error in the 2-qubit gates. This is also a useful demonstration of circuit length limitations due to where we currently stand with error. It was also decided not to implement noise mitigation strategies within our circuit for a couple of reasons. In QAOA the profit of implementing noise mitigation strategies is relatively minimal. This is even more exacerbated when we consider the overhead of implementing noise mitigation, noise mitigation uses gates which also contain error so any profit/noise reduction gained is offset from the error produced by the noise mitigation itself. In addition to this, gates cost money, and so noise mitigation seems even less reasonable. We chose to represent this problem for a few reasons, chief among them being to see the affects of noise on an algorithm that is not single error fragile and how noise impacts how complex/deep a circuit can be.

Number of 2-qubit gates for varying system size in simulation:

System size = 6 number of 2-qubit gates per layer = 72 * max 4 layers * 1000 shots = 288000 2Q gates

System size = 7 number of 2-qubit gates per layer = 132 * max 4 layers * 1000 shots = 528000 2Q gates

System size = 8 number of 2 qubit gates per layer = 222 * 4 * 1000 shots = 888000 2Q gates

System size = 9 number of 2 qubit gates per layer = 348 * 4 * 1000 shots = 1392000 2Q gates

Due to these large 2-qubit gate counts we will only pick one system size to run and run it with 4 layers and enough shots to get good data, we are open to input on the number of shots actually needed.

References

[1] E. Farhi, J. Goldstone, and S. Gutmann, "A Quantum Approximate Optimization Algorithm," 2014. Available: https://arxiv.org/pdf/1411.4028.pdf

[2] J. Marshall, F. Wudarski, S. Hadfield, and T. Hogg, "Characterizing local noise in QAOA circuits," IOP SciNotes, vol. 1, no. 2, p. 025208, Aug. 2020, doi: https://doi.org/10.1088/2633-1357/abb0d7.

[3] P. C. Lotshaw et al., "Scaling quantum approximate optimization on near-term hardware," Scientific Reports, vol. 12, no. 1, Jul. 2022, doi: https://doi.org/10.1038/s41598-022-14767-w.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 26 13:27:55 2023

@author: Aaron
"""

from math import comb
from itertools import combinations
from random import choice
import numpy as np
from scipy.optimize import minimize
from matplotlib import pyplot as plt
from qiskit import quantum_info as qi
from qiskit import QuantumCircuit
from qiskit import QuantumRegister, AncillaRegister
from qiskit.circuit.quantumregister import AncillaQubit
from qiskit import Aer, transpile
from qiskit_aer import noise, AerSimulator

def Z_product(angle, circuit, qubits, ancilla):
    # this is to allow passing the ancilla as an int index or an AncillaQubit
    # instance.
    if type(ancilla) is int:
        anc = circuit.ancillas[ancilla]
    else:
        anc = ancilla
    # likewise for the non-ancilla qubits
    if type(qubits[0]) is int:
        # circuit.qubits includes instances of AncillaQubit. This is to make
        # sure that this function works whether the ancillas come before or
        # after the other qubits.
        qs = [q for q in circuit.qubits if not isinstance(q, AncillaQubit)]
        qs = [qs[qubit] for qubit in qubits]
    else:
        qs = qubits

    circuit.cx(qs, anc)

    # i.e. len(qs) mod 2 == 1, i.e. len(qs) is odd
    if len(qs)%2:
        circuit.rz(2*angle, anc)
    else:
        circuit.rz(-2*angle, anc)
```

```
    # applies the cx in reverse order to before
    circuit.cx(qs[-1::-1], anc)

def find_GS(Hp):
    matrix_rep = Hp.to_matrix()
    # For a small Hamiltonian, we can just get the eigensystem directly
    vals, vecs = np.linalg.eig(matrix_rep)
    # the ground state energy is just the lowest eigenvalue of H
    Egs = np.min(vals)
    # Hp is diagonal in the computational basis for this problem which means
    # all its eigenvectors are basis states. The index in vals of an eigenvalue
    # is equal to the numeric value of the bitstring representing the basis
    # state corresponding to that eigenvalue, so we only need to save the
    # indices to have all the information we need about the eigenstates.
    # nonzero returns a tuple of arrays, one for each dimension of the input,
    # even if there's only one dimension. So we take the zeroth element of
    # that tuple to get the array of indices we actually want.
    ground_states = np.nonzero(vals == Egs)[0]

    return Egs, ground_states

def choose_interactions(N, terms, bodies):
    # list of all combinations of bodies-many qubits chosen from N qubits
    all_interactions = list(combinations(range(N),bodies))
    # list to store the combinations we choose to put in the Hamiltonian
    interactions = []
    for i in range(terms):
        # randomly choose an available interaction
        index = choice(range(len(all_interactions)))
        # pop it from the list of combinations so we can't re-use it later
        interaction = all_interactions.pop(index) + (choice([1,-1]),)
        interactions.append(interaction)

    return interactions

def Z_product_string(N, indices):
    # it's easiest to start with all elements I and then change the appropriate
    # elements to Z. Even in the worst case, this is still linear in N.
    segments = ['I' for i in range(N)]

    # As long as indices is a tuple taken from the list generated by
    # choose_interactions, i is guaranteed to be a valid index for segments.
    for i in indices:
        segments[i] = 'Z'
```

```
    # The reason for reversing here is that when qiskit takes strings
    # representing multi-qubit operators, the single-qubit operator acting
    # on the zeroth qubit is the last element of the string, and the operator
    # acting on the Nth qubit is the first element of the string.
    return ''.join(reversed(segments))

# This is useful not for actually running the algorithm, but for finding the
# true ground state energy and for finding the expectation value of Hp in
# the state produced by the algorithm.
def make_Hp(N, interactions):
    # this makes a list that can be passed to the SparsePauliOp.from_list()
    # method.
    term_list = [(Z_product_string(N,i[:-1]), i[-1]) for i in interactions]

    Hp = qi.SparsePauliOp.from_list(term_list)
    return Hp

def make_QAOA_circuit(N, interactions, params):
    qr = QuantumRegister(N, 'q')
    anc = AncillaRegister(1, 'anc')
    circ = QuantumCircuit(qr, anc)

    # initialize the primary qubits in a uniform superposition, and the
    # ancilla in the 0 state.
    circ.reset(circ.qubits)
    circ.h(qr)
    for i in range(params.shape[0]):
        alpha = params[i,0]
        beta = params[i,1]
        # First we apply the Hp-derived gates
        for interaction in interactions:
            qubits = interaction[:-1]
            V = interaction[-1]
            theta = V*alpha
            # This function adds gates to the circuit in-place, so there's
            # no need to assign or append anything
            Z_product(theta, circ, qubits, 0)

        # now we apply the X rotations.
        circ.rx(2*beta, qr)

    return circ

def make_cost_function(N, interactions):
    Hp = make_Hp(N, interactions)
```

```
    simulator = Aer.get_backend('aer_simulator')

    def cost_function(params):
        # scipy.optimize.minimize takes a 1D vector of parameters, but the
        # function that constructs the circuit takes a 2D array with each row
        # being an alpha-beta pair
        grouped_params = np.reshape(params, (int(params.shape[0]/2),2))
        circ = make_QAOA_circuit(N, interactions, grouped_params)
        circ.snapshot('out')

        circ_comp = transpile(circ, simulator)
        job = simulator.run(circ_comp, shots=1)
        result = job.result()

        state = qi.Statevector(result.data()['snapshots']['statevector']['out'][0])

        return np.real(state.expectation_value(Hp))

    return cost_function

def make_aria_noise_model():
    model = noise.NoiseModel(['u', 'cx'])

    # Error probabilities
    prob_1 = 0.0005  # 1-qubit gate
    prob_2 = 0.005   # 2-qubit gate

    # Depolarizing quantum errors
    error_1 = noise.depolarizing_error(prob_1, 1)
    error_2 = noise.depolarizing_error(prob_2, 2)

    model.add_all_qubit_quantum_error(error_1, ['u'])
    model.add_all_qubit_quantum_error(error_2, ['cx'])

    return model

def optimize_noiseless_QAOA(N, interactions, max_layers):
    cost_fn = make_cost_function(N, interactions)
    best_expcs = np.zeros(max_layers)
    solutions = []

    for i in range(max_layers):
        # The 'magic numbers' given for the initial guess reflect the fact that
        # there was consistently a minimum in the expectation value of H_p for
        # negative alpha and positive beta in low dimensions when doing grid
```

```
        # search for the QAOA assignment.
        result = minimize(cost_fn, np.array([-0.1,0.1]*(i+1)),
                          bounds=[(-np.pi/2,np.pi/2),(-np.pi/2,np.pi/2)]*(i+1))

        best_expcs[i] = result.fun
        solutions.append(np.reshape(result.x, (int(len(result.x)/2),2)))

    return best_expcs, solutions

def run_noisy_circuits(N, interactions, best_params, shots):
    noise_model = make_aria_noise_model()
    simulator = AerSimulator(noise_model=noise_model)

    avg_expcs = np.empty((len(best_params),))

    Hp = make_Hp(N, interactions)

    for i,params in enumerate(best_params):
        circ = make_QAOA_circuit(N, interactions, params)
        circ.snapshot('out')

        circ_comp = transpile(circ, simulator)
        result = simulator.run(circ_comp, shots=shots).result()

        states = result.data()['snapshots']['statevector']['out']

        expc_total = 0

        for s in states:
            state = qi.Statevector(s)
            expc_total += np.real(state.expectation_value(Hp))

        avg_expcs[i] = expc_total/len(states)

    return avg_expcs

def main():
    fig, axs = plt.subplots(2,2)
    x = 0
    y = 0
    all_interactions = []
    all_solutions = []
    for i in range(4):
        N = 6+i
        bodies = 3
```

```
        terms = comb(N-1, bodies) + 2
        max_layers = 4
        interactions = choose_interactions(N,terms,bodies)
        all_interactions.append(interactions)

        best_expcs, solutions = optimize_noiseless_QAOA(N, interactions, max_layers)
        avg_expcs = run_noisy_circuits(N, interactions, solutions, 1000)
        all_solutions.append(solutions)

        layer_counts = list(range(1, len(avg_expcs)+1))

        fig.suptitle("Increasing")
        x=x%2
        y=y%2
        fig.tight_layout()
        axs[x,y].plot(layer_counts, avg_expcs, label='Average Noisy EV')
        axs[x,y].plot(layer_counts, best_expcs, label='Best EV')

        axs[x,y].legend()
        axs[x,y].set_title('N = {0}'.format(6+i))
        x+=i%2
        y+=1

    plt.show()

    return all_interactions, all_solutions
```

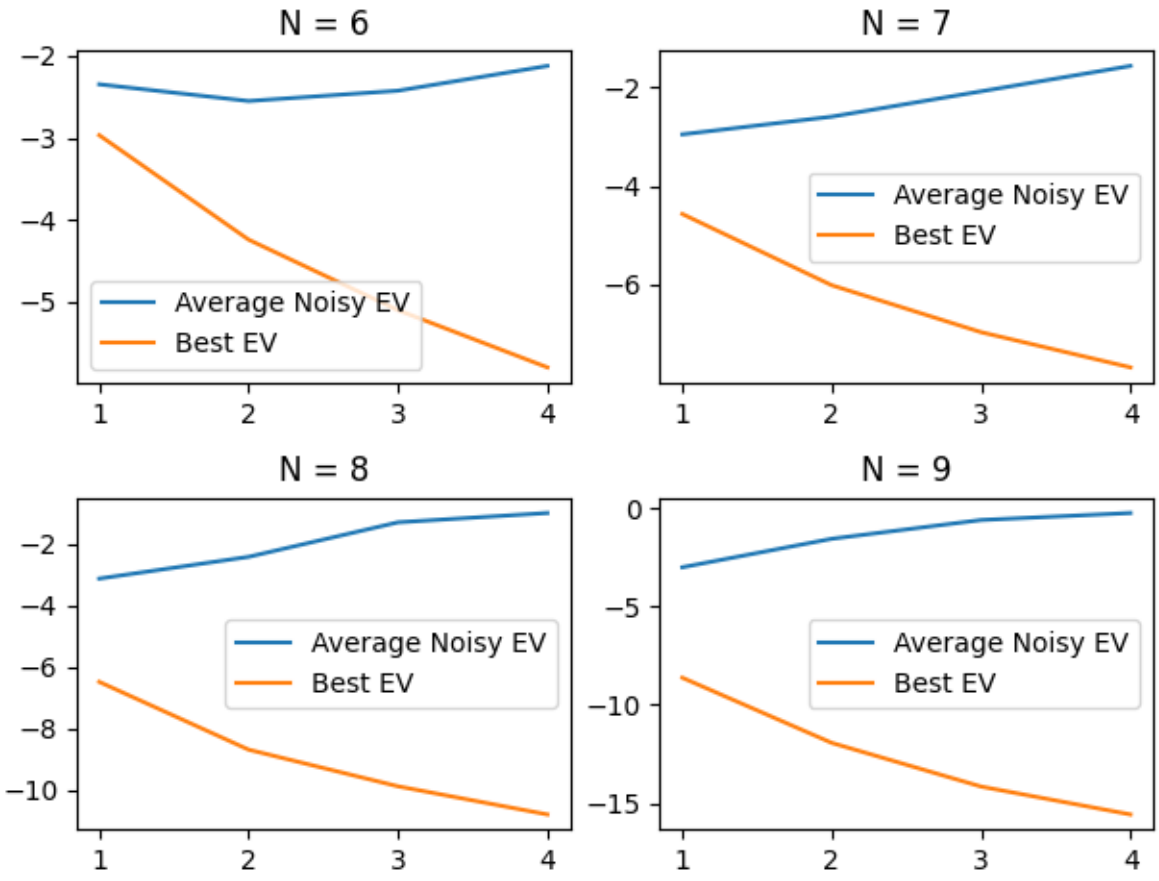

Figure 1: Increasing System Size