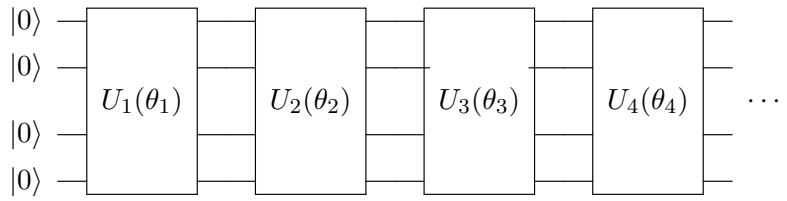
Ameya Thete 25th September 2020

QOSF Quantum Computing Mentorship Program 2020

Task 1

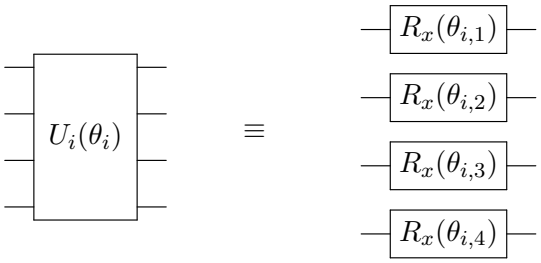
1 Problem Statement

The problem involves implementing the four-qubit quantum state $|\psi(\theta)\rangle$ given below.

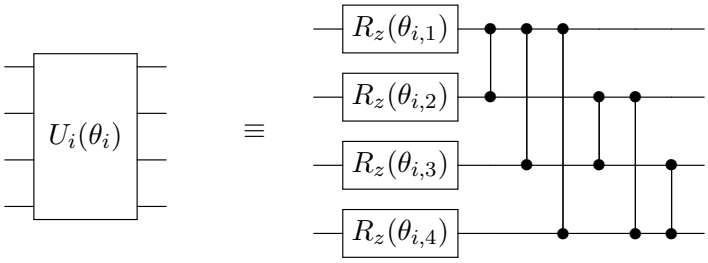


Here, an odd block and an even block together constitute a "layer". The number of layers in the circuit is represented by a parameter L. Each odd and even variational block is represented as follows:

Odd Block



Even Block



The angles $\{\theta_{i,n}\}$ are variational parameters lying in the interval $(0, 2\pi)$ and are randomly initialised.

The task is to **report with a plot**, as a function of the number of layers L, the minimum distance ε which is defined as

$$\varepsilon = \min_{\theta} \| |\psi(\theta)\rangle - |\phi\rangle \|$$

where $|\phi\rangle$ is a randomly generated state vector and $\|\cdot\|$ is the norm of a quantum state. The right set of parameters $\{\theta_{i,n}\}$ can be found via any method.

2 Solution

I have attempted the solution using IBM's Qiskit framework and Python 3.8.3

I will periodically refer to sections of code that can be found in the accompanying Jupyter Notebook titled `QOSF Task-1.ipynb`. I am deliberately not including the code snippets here to avoid cluttering the document.

The first thing to do was to create the variational circuits for the odd and even blocks. This can be achieved easily using Qiskit's `QuantumCircuit` objects and their associated methods. To allow for more fine tuning and to introduce modularity, I created dedicated methods for both the even and odd blocks and a final method called `layer` that stitches the two together. A final method called `assemble_circ` puts the entire circuit together. The number of layers to be created is decided by the number of theta parameters passed to the function.

The next task was to define the cost function to be passed to the optimiser. This is pretty straightforward to implement and the `numpy.linalg.norm` method is able to compute the norm of the difference between the target statevector and the statevector generated by the circuit. A practice that I had to implement here which I am not entirely comfortable with is making the randomised target statevector (`random_state`) a global object. As the cost function computes the cost associated with each depth, every run of the optimiser would generate a new circuit; defining the random statevector within the `cost_function` method would invariable lead to a different random state being generated each time which is completely wrong. I am certain better practices can be used to circumvent this problem, but given the accompanying time constraints, I had to settle for this specific implementation.

Now comes the most interesting part: choosing the optimisers. To do this, I looked at the optimisers that are provided by Qiskit's Aqua library. The reasons for doing this are two fold: first, I wanted to gain a better understanding of the Qiskit framework; and second, I felt that using optimisers shipped with Qiskit would eliminate the need to preprocess the input to the optimisers and outputs generated by them as I was working with a time constraint.

I selected three optimisers for the job: Adam, COBYLA and AQGD (Analytic Quantum Gradient Descent). The reason for selecting these three are because all three optimisers support constrained optimisation, and I had a minimal experience using Adam and GD. All of them were run on the `statevector_simulator` backend. I will categorically cover the performances of each in the proceeding subsections

3 Results

3.1 COBYLA

I ran COBYLA with the following parameters - `maxiter:200, tol:0.0002`. The results I obtained are plotted below:

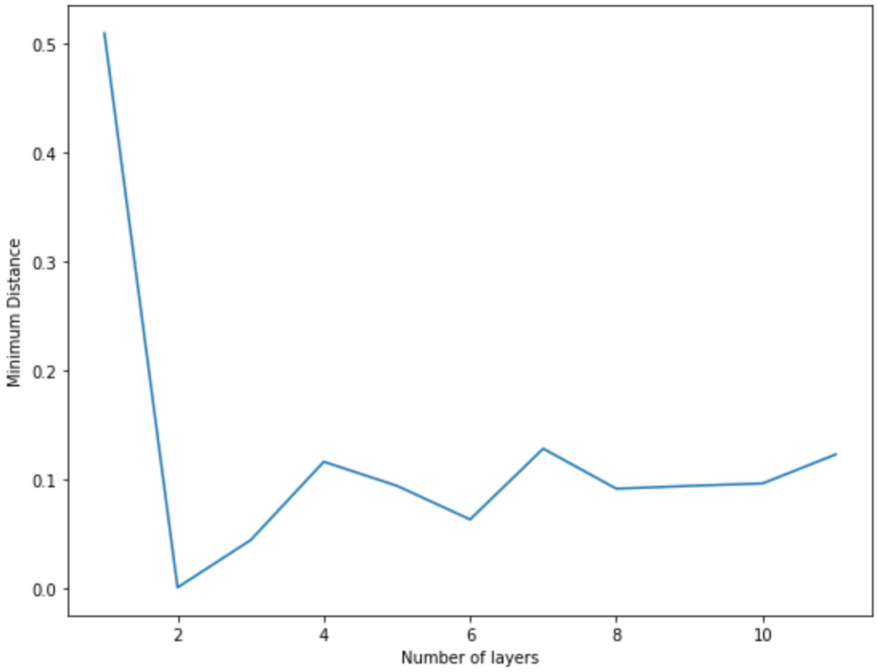


Figure 1: A plot of the minimum distance as a function of depth for COBYLA

As seen from the plot, COBYLA did converge to minimum pretty rapidly — at depths of 2 layers only! Another advantage was that COBYLA ran pretty fast. However, it is apparent that it diverged from the minimum at the very next depth, and subsequently stabilising at the higher-than-minimum distance at larger depths of the circuit. Although I am not sure how COBYLA performs in a quantum environment, it has been shown that COBYLA can have suboptimal results classically. [1] I might be incorrect, but I theorise that this behaviour might have been carried into the quantum regime as well.

3.2 AQGD

Next, I chose to run AQGD with the same parameters as before - `maxiter:200, tol:0.0002`. The results I obtained are plotted below in Figure 2.

As seen from the plot, AQGD converged to a minimum of 0.5 after around 10 iterations. It can also be seen that the intermediate behaviour of the optimiser was not monotonic and kept changing. I believe that this might have been because I ran the optimiser for a much lower number of maximum iterations as it took very long to run which might have affected the behaviour.

[1] Wendorff, Andrew, Emilio Botero, and Juan J. Alonso. "Comparing Different Off-the-Shelf Optimizers' Performance in Conceptual Aircraft Design." 17th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference. 2016.

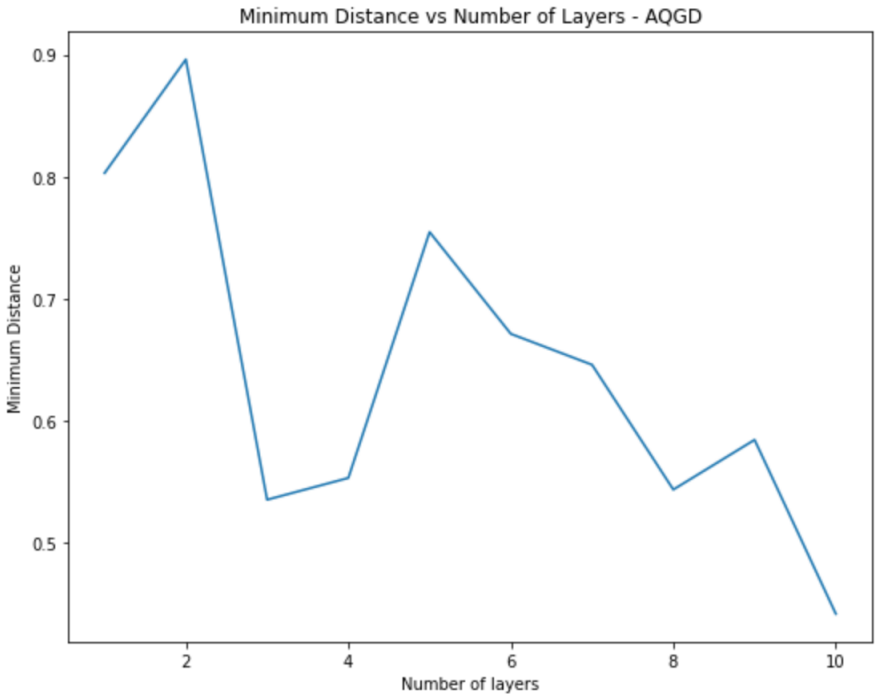


Figure 2: A plot of the minimum distance as a function of depth for AQGD

3.3 Adam

Finally, I ran the Adam optimiser with the following parameters - `maxiter:1000, tol:0.0002`. The results I obtained are plotted below in Figure 2.

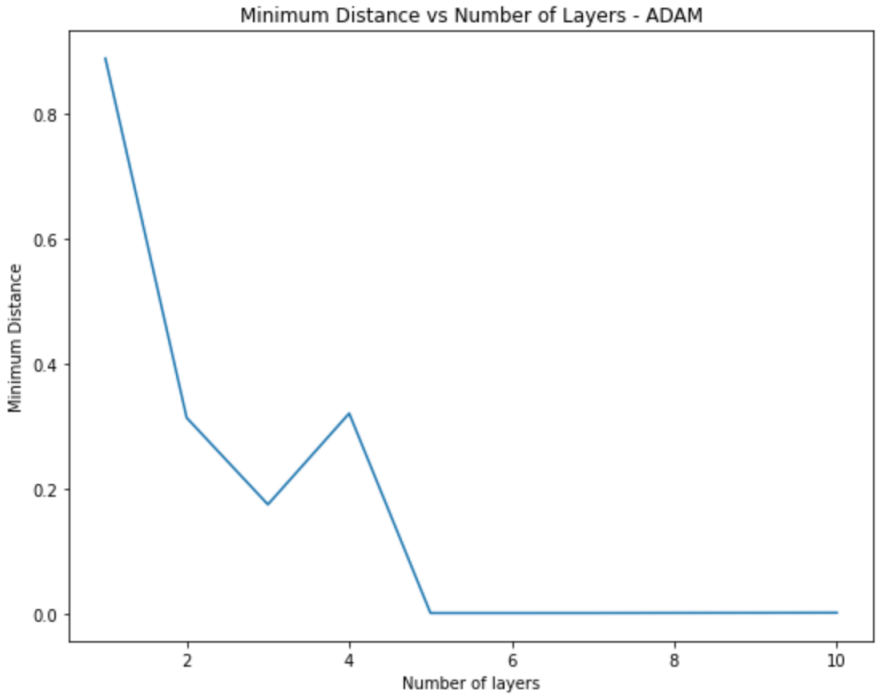


Figure 3: A plot of the minimum distance as a function of depth for Adam (Run 1)

The results I obtained were pretty promising. There is a steady, almost monotonic (except for the localised peak at $L = 4$) decline in the minimum distance as the number of layers increases, and once it reaches a minimum, the value remains stable with the addition of more layers. To test the optimiser once more, I ran it again with the same setting. I obtained a similar result this time.

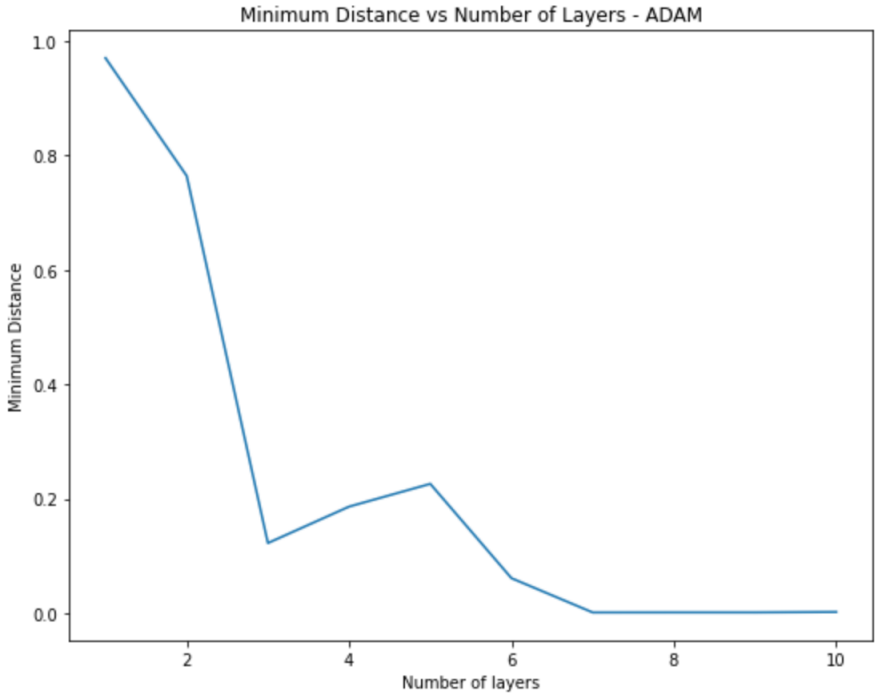


Figure 4: A plot of the minimum distance as a function of depth for Adam (Run 2)

A major advantage I observed with using Adam was that the minimum value remained stable once it was reached and was unaffected by the adding additional layers; however this was offset by the excruciatingly high run times, with the first run taking almost 4 hours! (I first thought this was a issue with my hardware locally, but similar run times were observed even when I ran it on the cloud using Google Colab). I am really not sure why the run times were high.

4 Conclusion

To summarise, I executed the variational circuit using three optimisers: Adam, COBYLA and AQGD. Adam provided the best results (offset by its high run times), next best was COBYLA, which did converge to a minimum of 0, but it was not stable beyond that layer, and finally the least promising was AQGD, which did not converge to a global minimum as its other counterparts, but this might be because I did not let it run for longer termination iterations.

Unfortunately, I could not try out the bonus task as I didn't have the time, but I would really like to see if changing the basis gates for the layer would affect the performance.

I really enjoyed this task as this was the first time I implemented everything from scratch while also researching which components to use. Comparing the results I obtained along with theorising the possible reasons for obtaining those results, and also troubleshooting errors was really interesting too.