Supplementary Material

I. STEP-BY-STEP WALKTHROUGH OF ALGORITHM

Here we provide a step-by-step walkthrough of the AlphaZero algorithm with particular focus on the Monte-Carlo tree search (MCTS). A MCTS consists of nodes, which represent states (s_t) connected by edges which represent state-action pairs (s_t, a_t). The nodes are connected by the edges such that if we go from state s_t to s_{t+1} by selecting action a_t, then the edge (s_t, a_t) connects the node (s_t) with the node (s_{t+1}). Each edge contain four numbers: a visit count $N(s_t, a_t)$, a total action value $W(s_t, a_t)$, a mean action value $Q(s_t, a_t)$, and a prior probability $P(s_t, a_t)$. In order to illustrate how all of this is connected we provide a simple example using an action space of just three actions a_1, a_2, and a_3. We illustrate how the MCTS is gradually built in Supplementary Fig. 1.

A. Step One

First, we start from an empty tree where we add the root state (initial state) s_0. In our own implementation this would be the initial unitary $\hat{U}_0$, which was the identity matrix. The root state is fed into the neural network $\mathrm{nn}(s_0) = (\mathbf{p}, v)$, where $\mathbf{p} = (p_1, p_2, p_3)^T$ denotes the three probabilities corresponding to each of the three actions. We initialize the edges as $N(s_0, a_j) = W(s_0, a_j) = Q(s_0, a_j) = 0$ and $P(s_0, a_j) = p_j$ for $j = 1, 2, 3$. This is illustrated in Supplementary Fig. 1 under step one.

B. Step Two

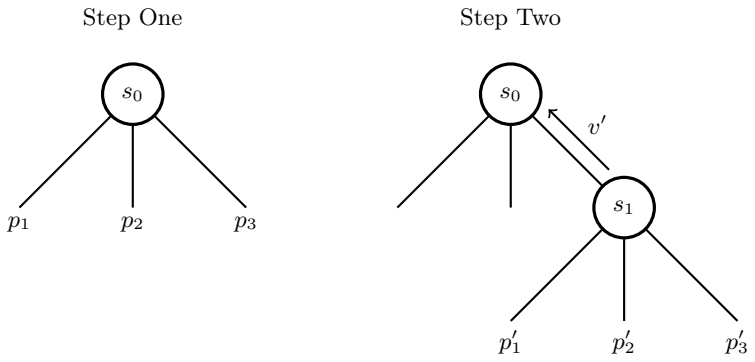
We set the current state s to be the root state s_0 and calculate an uncertainty for each edge based on a combination of prior probabilities and visit count:

$$U(s_0, a_j) = c_{\mathrm{puct}} P(s_0, a_j) \frac{\sqrt{\sum_{i=1}^{3} N(s_0, a_i)}}{1 + N(s_0, a_j)} \text{ for } j = 1, 2, 3. \tag{1}$$

Here c_{puct} is a parameter that governs the level of exploration. The uncertainty is higher for less-visited states, but the neural network may compensate for this by assigning a higher probability $P(s_0, a_j)$ to actions it has previously learned are preferable. Now we select an action that maximizes the sum of uncertainty and mean action-value $a = \mathrm{argmax}_j(Q(s_0, a_j) + U(s_0, a_j))$. Note that initially both the uncertainty and mean action-value are zero, and so the very first action is drawn at random. Let us say that $a = a_3$, and so we obtain a new state, s_1, via this action. In our algorithm this corresponds to multiplying the unitaries $\hat{U}_1 = \hat{U}(a_3)\hat{U}_0$. We add s_1 to the tree and initialize its edges by feeding it to the neural network similar as in the step one i.e. $\mathrm{nn}(s_1) = (\mathbf{p}', v')$ where $N(s_1, a_j) = W(s_1, a_j) = Q(s_1, a_j) = 0$ and $P(s_1, a_j) = p'_j$ for $j = 1, 2, 3$. But this time we also use the value v' given by the neural network in a back-pass where we update all edges we encountered in the forward search. In this case, it is only edge (s_0, a_3), where we update its numbers as $N(s_0, a_3) \leftarrow N(s_0, a_3) + 1$, $W(s_0, a_3) \leftarrow W(s_0, a_3) + v'$, and $Q(s_0, a_3) = W(s_0, a_3)/N(s_0, a_3)$ i.e. in this example $N(s_0, a_3) = 1$ and $W(s_0, a_3) = Q(s_0, a_3) = v'$.

C. Step j

The next steps in the algorithm resembles the previous step. We start from the root state and traverse down the tree by selecting actions according to $a_t = \mathrm{argmax}_a(Q(s_t, a) + U(s_t, a))$. This is done until we reach a not-previously visited node (leaf node) similar to step two where we had not encountered s_1 before. Remember that we are working with discretized actions so each state is uniquely determined by the series of actions that led to said state. This allows us algorithmically to distinguish different states. The leaf node is added to the tree, the state is fed to the neural network, and its edges are initialized. Then we use the value v to update all the edges we have visited in this particular search. Both the initialization and updates are done as in the previous step.



Supplementary Fig. 1. A simple example of how a MCTS works (see text).

D. Last step

After a pre-set number of searches have been conducted as described above, we calculate a policy based on the visit count of the root state

$$\pi(a_j|s_0) = \frac{N(s_0, a_j)^{1/\tau}}{\sum_i N(s_0, a_i)^{1/\tau}}, \tag{2}$$

here τ denotes a parameter that also governs the level of exploration, which we anneal from unity towards zero during the simulations. We draw an action from the policy, and update the root state accordingly. We discard the rest of the tree belonging to the other nodes in the second layer, but keep the part that is still reachable from the new root node. We save the old root state and the policy in a replay buffer (database). At the end of the episode we also save the final score z, which was the fidelity.
After each episode we train the neural network (update the network parameters) such that its prediction becomes closer to the stored data i.e. $\mathrm{nn}(s) = (\mathbf{p}, v)$ should approach $(\boldsymbol{\pi}, z)$ for all $s, \boldsymbol{\pi}, z$ stored in the replay buffer.

II. WALL TIME TO REACH A SPECIFIC INFIDELITY THRESHOLD

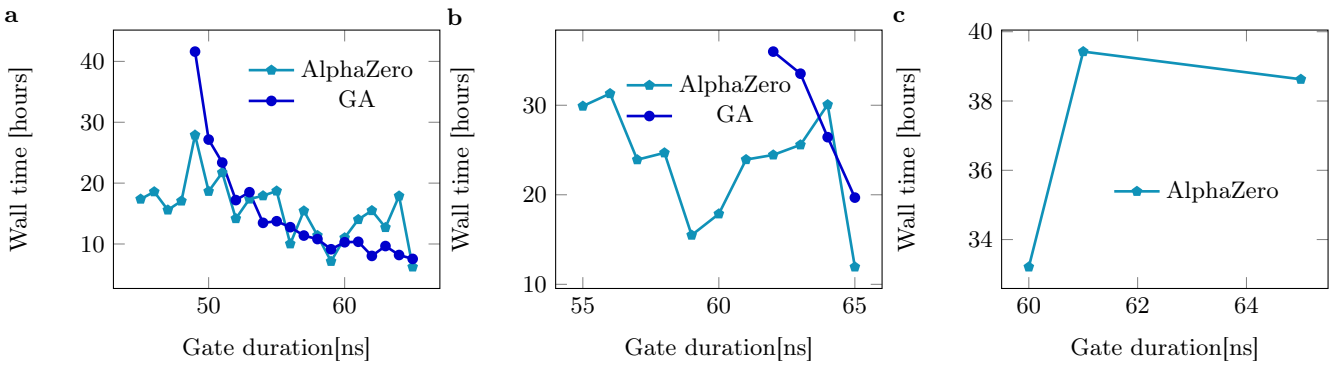
In the main part of our paper, we investigated how low infidelities the different optimization algorithms could reach for a fixed wall time. However, advanced optimization algorithms such as AlphaZero take a long time *to learn* how to solve the optimization problem, and hence its initial performance is very poor. Therefore we now supply a different analysis that investigates how long a wall time is needed to reach a predefined infidelity threshold. We repeat the analysis for all of the three optimization tasks in the main paper.

A. Optimization in the discrete case (SFQ)

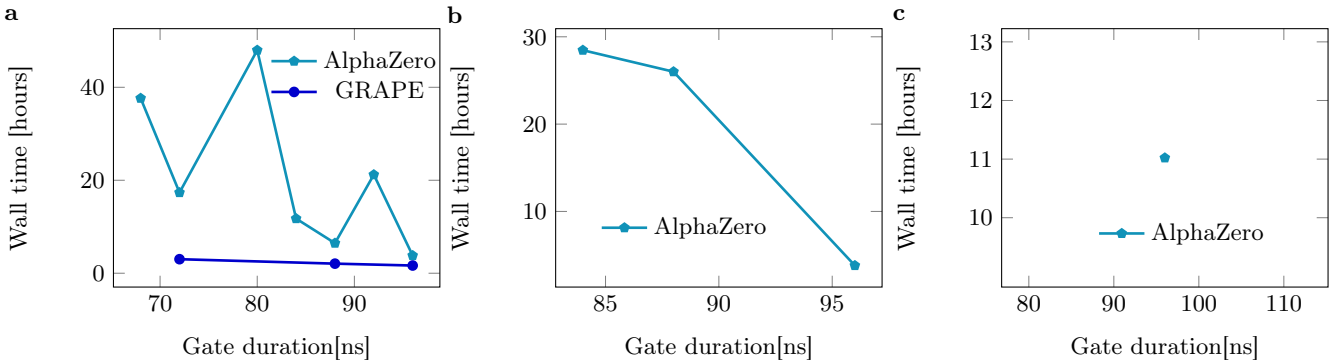
First, we consider the optimization in the discrete case (SFQ). Supplementary Fig. 2 shows the wall time needed to reach a specific gate duration for three different infidelity thresholds. For instance Supplementary Fig. 2a shows the wall time needed to reach an infidelity of 5.0×10^{-2}. In general, the genetic algorithm (GA) and AlphaZero are compatible with respect to the wall time needed. However, AlphaZero, unlike the GA, is able to reach this infidelity when the gate duration falls below 50ns. A similar tendency is seen in Supplementary Fig.2b where the threshold is 1.0×10^{-2}. The last infidelity threshold of 1.0×10^{-3} is plotted in Supplementary Fig. 2c. Here, only AlphaZero is able to find any solutions, but not at each depicted gate duration. This is due to the fluctuations in the AlphaZero results that can be seen in the main paper.

B. Optimization of constrained continuous pulses (Gaussian filtering)

Here we consider the constrained optimization of continuous pulses i.e. Gaussian Filtering. Supplementary Fig. 3 shows the wall time needed to reach specific gate durations for three different infidelity thresholds. For an infidelity of



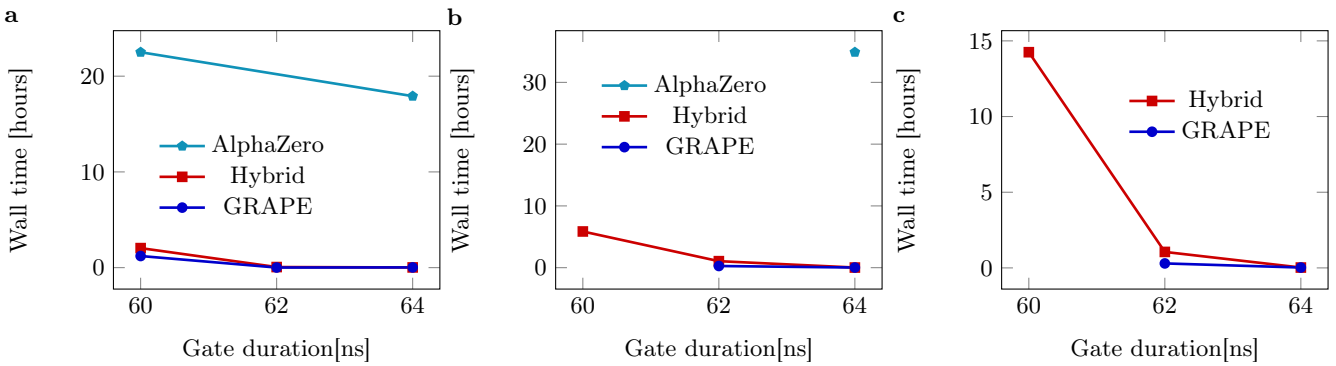
Supplementary Fig. 2. Optimization in the discrete case (SFQ) for three different infidelity thresholds: **a** 5.0×10^{-2}, **b** 1.0×10^{-2}, and **c** 1.0×10^{-3}.



Supplementary Fig. 3. Optimization of constrained continuous pulses (Gaussian filtering) for three different infidelity thresholds: **a** 1.0×10^{-1}, **b** 5.0×10^{-2}, and **c** 1.0×10^{-2}.

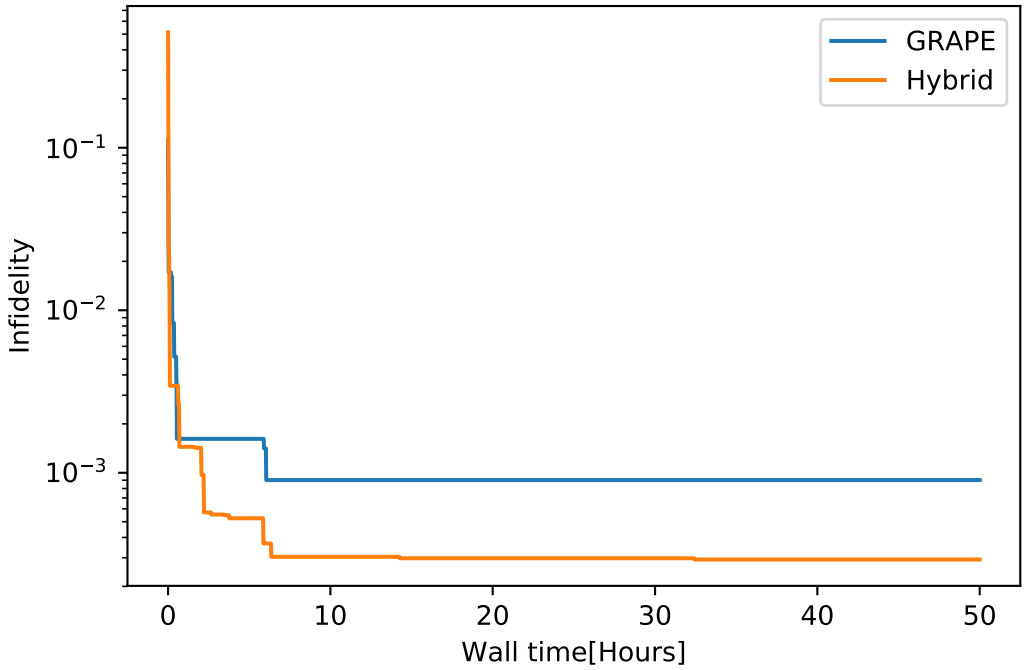
1.0×10^{-1} as displayed in Supplementary Fig. 3a GRAPE is a lot faster when it does succeed. However, at the other thresholds at 5.0×10^{-2} and 1.0×10^{-2}, which is plotted in Supplementary Fig. 3b and c, only AlphaZero succeeds.

C. Optimization on continuous (piecewise constant) pulses



Supplementary Fig. 4. Optimization on continuous (piecewise constant) pulses for three different infidelity threshold: **a** 1.0×10^{-3}, **b** 0.5×10^{-3}, and **c** 0.3×10^{-3}.

At last we consider optimization of piecewise constant pulses. The results can be seen in Supplementary Fig.4. GRAPE is generally faster and the AlphaZero algorithm slower. But at the threshold of 0.5×10^{-3} and 0.3×10^{-3},



Supplementary Fig. 5. Comparison between GRAPE and the Hybrid algorithm at 60ns for the optimization problem of creating piecewise constant pulses.

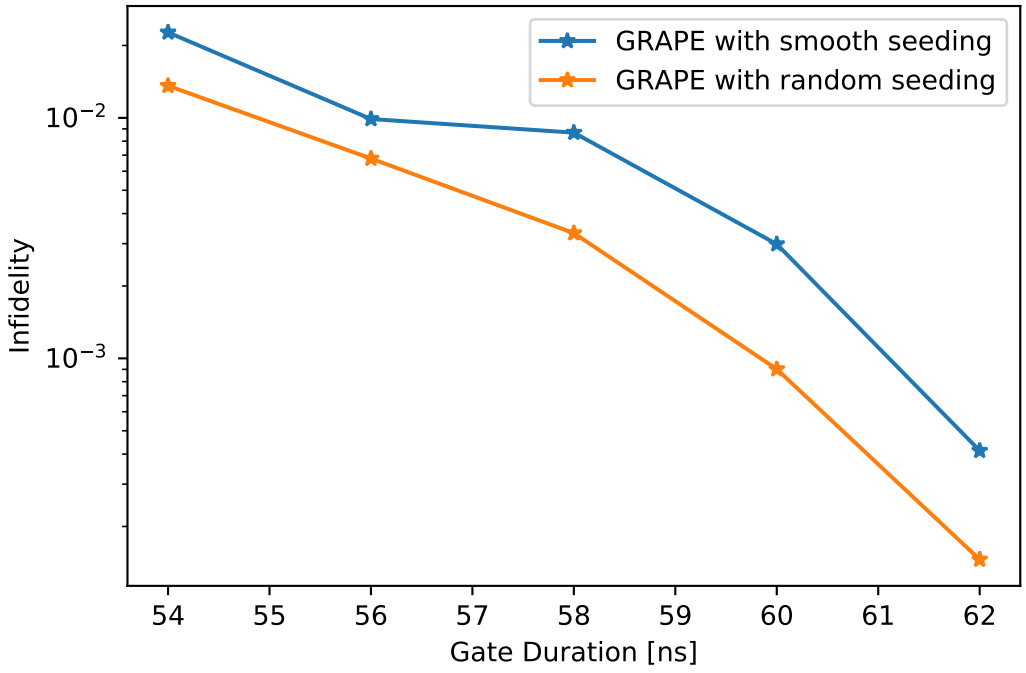
which is plotted in Supplementary Fig. 4b and c respectively, the Hybrid is able to find solutions at 60ns, while GRAPE is not.

In order to compare the convergence rates between the Hybrid and GRAPE algorithm, we also plot the best found infidelities at 60ns in Supplementary Fig. 5 as function of the wall time consumption. We see that the Hybrid significantly outperforms GRAPE during the first eight hours of wall time, where after both algorithms essentially saturate, although the Hybrid still finds minor improvements.

In the next section we briefly comment on the seeding strategy employed by GRAPE.

III. COMPARISON OF SEEDING STRATEGIES

In the main paper we exclusively applied GRAPE with random seeds. This was due to the fact that the optimization problems we considered did not have analytical solutions that could be used as a seeding strategy, unless one would go to significantly larger gate durations. Therefore, we instead consider seeding with smooth pulses in order to compare against random seeding. Specifically, we draw the seeds from a seventh order polynomial where we draw the roots at random. We also constrain the polynomial such that it starts and end with zero. The results of 50 hour wall time simulations for the optimization on piecewise constant pulses using the two strategies can be seen in Supplementary Fig. 6. We see that the random seeding strategy outperforms the smooth one, thus supporting the tabula rasa strategy employed in the main paper.



Supplementary Fig. 6. Comparison of GRAPE using random seeding and seeding with smooth initial guesses. Each data point corresponds to the best infidelity obtained in a 50 hour wall time simulation for the problem of generating piecewise constant pulses.