AdaQuantum – User Guide

Algorithm by Paul Knott, Rosanna Nichols, Lana Mineh, Jesús Rubio, Lewis O'Driscoll & Jonathan Matthews

https://arxiv.org/pdf/1812.01032.pdf and https://arxiv.org/abs/1812.03183

Introduction and notes

With this guide a user should be able to set-up, run, and then begin to generate results using the algorithm AdaQuantum. It is recommended that you read our paper that introduced the algorithm before reading this guide. We are happy to provide further guidance on using the algorithm (please email Paul.Knott@Nottingham.ac.uk). We also interested in collaborating to extend the algorithm, for example by adding more figures of merit or toolbox elements. And any feedback on the algorithm or on this guide would be very welcome.

We do not always use the standard terminology in our algorithm, and the terminology in our paper doesn't always match our Matlab code. So here are some of the main terms and keywords we use: Figure of merit (FOM) = Fitness function. Fitness value (FV) = Objective value. Genome = DNA.

Setting up Matlab

1. After downloading AdaQuantum, add the parent folder and all subfolders to Matlab's path
2. Install Matlab's global optimisation toolbox.

Creating a toolbox and figure of merit

We first have to create a toolbox and FOM. This is done via a user interface (UI) by running '*Create_toolbox_and_FOM_with_UI*'. The first UI allows you to select which quantum optics elements – states, measurements and operators – you would like the algorithm to optimise over. You can also select:

- The value of loss in the detectors (0 = no loss, 1 = all photons lost)
- The number of modes in the experiment to be designed (it's recommended to stick to 2 modes)
- The number of operators (labelled *'m'* in our paper). We recommend 2-5 operators.
- The maximum truncation. As discussed in our paper, in stage 3 of the GA the algorithm will first simulate the circuit specified by the input DNA at a very low truncation, then repeat this, increasing the truncation on each iteration, until either the average number of photons in the final state converges, or until the maximum truncation is reached. If the average number of photons has not converged at maximum truncation then the fitness function of the state is severely penalised. This ensures that the final results outputted by the algorithm are accurate. For the results in our paper, we often needed Max. Truncation to be 120-140, but this will change depending on what states you are looking for. The first consideration is that larger states will require a larger truncation (we generally looked for states with less than 1.5 photons on average).
- Loss on the output state

After creating a toolbox you should save it (preferably in the *Toolbox* folder) by clicking *Save settings* on the UI. We have created a sample toolbox that works well for running the algorithm, called *Sample_toolbox*.

After creating and saving a toolbox, click *Next*. You can now create a figure of merit (FOM). A good FOM to start with is *'QFI / nbar'*. This is the quantum Fisher information (QFI) divided by the average number of photons, nbar. The QFI is unbounded, and optimising for QFI alone can create states that are too large for the Max. Truncation. In contrast, QFI/nbar is more stable and reliable.

Ticking the box *'Penalise low heralding probability'* means that states that have a low probability of being heralded are not considered. It is recommended to tick this, as states with a tiny heralding probability can have number distributions with contributions beyond the Max. Truncation. By default, probabilities less than 0.001 are rejected – this can be changed in *'main_3stage'* (see below) by changing the value of *'heralding_accepted'*.

Finally, you can choose an augmenting function. The algorithm will multiply the FOM by the value of the augmenting function for the average photon number, nbar, of the state in question. This is used to force the algorithm to find states with certain values of nbar. If this augmenting function is not used, then the algorithm can find huge states that are too large for the Max. Truncation. As an example, if you choose the *Exponential fall-off* augmenting function with default values (click *Preview* for a preview), and the algorithm produces a state with 1 photon on average, then the FOM will be multiplied by approx 0.8. We generally restrict to states with less than 1.5 photons.

After saving the FOM, click *Exit*. We have created a sample FOM that works well (when used with *Sample_toolbox*) for running the algorithm, called *Sample_FOM*.

Running the algorithm

Our genetic algorithm has three stages:

1. First, a large number (*Random_initial_population_size*) of random DNAs are produced and evaluated. A collection of DNAs with the best FVs are selected for the next stage. Due to the nature of the problem, many of the FVs are zero, so this stage rules out many of these zero FVs, and gives the genetic algorithm (GA) a better starting population. In this stage the quantum simulation is only approximate, but it is fast.

2. In the second stage, the GA uses a medium-sized population (*Population_size_s2*). The simulation is less approximate than state 1, and hence slower. This stage only runs for a small, fixed number of generations, usually 10.

3. In the final stage, the simulation is accurate but slow. Here the population (*Population_size_s3*) is smaller. The GA runs until the FOM doesn't improve over *StallGen_s3* generations, OR if the total generations reaches *Generations_s3*.

Now open *'main_3stage'*. The first thing to do is modify the *'Selected Toolbox'* and *'Selected FOM'* that are loaded (this is the toolbox and FOM that will be optimised over). Or just keep the default, *Sample_toolbox* and *Sample_FOM*. We recommend that the only things you modify initially are the population sizes of the 3 stages of the algorithm. These population sizes greatly affect how long it takes to run the algorithm to produce results. For the default population sizes of 100, 30 and 20, for stages 1, 2 and 3 respectively, the algorithm runs in around 2 minutes on my laptop. In contrast, for the results in our paper the population sizes for the three stages were around 1e7, 1e5, and 2e4. This ran for 4 days total on 16-cores of the University of Nottingham's HPC. Huge population sizes such as these were necessary to overcome local minima and find states with the large QFIs presented in our paper, but interesting results can still be found by just running for a few hours.

It takes some experimentation to find suitable population sizes for a given computer. We recommend that stages 1 and 2 run for time **t** each, and stage 3 running for time **2*t**. (See below for a different *main* function that allows the time to be controlled more precisely.)

We won't introduce all the options that can be modified in *'main_3stage'*, as they are all commented in the code. But some of the most important options are:

- The genetic algorithm hyperparameters are introduced below. These determine how quickly and effectively the algorithm searches.
- We recommend that the truncation sizes of the different stages are left as default (30 for stage 1 and 80 for stage 2; stage 3 is selected in the UI, but we'd recommend 120-140).
- The command *'p = gcp'* sets up parallel processing on Matlab. This greatly speeds up the running time.

Results produced by the algorithm

By default, the algorithm saves the entire workspace after stages 2 and 3. This uses more memory, but we find that it makes it easier to fully analyse the output. The results are stored in folders labelled with the date and time that the results were produced. The algorithm also prints the best state that was found, along with its fitness value (FV; the value produced by the FOM), to the Matlab command window (and saves this info in *'Useful stuff print.txt'*). In the saved workspace, the following variables are noteworthy:

- *Winning DNA*: This is the DNA that gave the best FV. Running the command: *'DNAtoDisplay(WinningDNA, SelectedToolbox, SelectedFigureOfMerit, num_modes, max_ops)'* prints out the details of the state corresponding to this DNA.

- *nbar_post_measurement*: This is the average number of photons in the final state

- *heralding probability*: This is the probability of successfully heralding the final state

- *output density operator*: This is the density matrix of the final state. If this is a pure state, it can be converted to a vector using *'DensityOperatorToStateVector(output_density_operator)'*

- *FigureOfMeritValue*: This is the FV. It encompasses the augmenting function and other penalties within the algorithm. To get the original FV you desire (without penalties nor augmentation), pass the output density operator through the FOM function. For example, if you want to know the QFI of your state, run the command: '*QFI(output_density_operator)*'

Other ways to run the algorithm

This guide focusses on running the basic algorithm using *'main_3stage'*. There are other versions of *main*, for different purposes, which are briefly introduced next. But note that these codes aren't commented thoroughly, and in this guide we don't give a comprehensive introduction to them.

'main_3stage_timed': In this main, each stage of the algorithm runs for a fixed time. This is preferable when e.g. running on a High Performance Computing facility, where you have to specify how long to run the code for. You choose the time allowed for each stage in the '*%% Timing'* section. You still need to set the population sizes for stages 2 and 3, which need to be adjusted so that these stages each run for a reasonable number of generations (e.g. 10 generations for stage 2 and 100 generations for stage 3).

'main_other_solvers': This is an old version of the algorithm, and might be out-of-date in places. But the main benefit of this is that it runs with different global optimisation methods: *Particle swarm*, *pattern search*, and *simulated annealing*. These can be selected by changing *use_algorithm*. In our runs, our 3-stage genetic algorithm performed best, but we didn't do a comprehensive comparison, and it may be that our hyperparameters weren't optimal (simulated annealing showed the most promise).

'main_3stage_timed_BMSE_1trial' and 'main_3stage_timed_BMSE_n_trials': These are specifically designed to run the Bayesian mean squared error (BMSE) FOM, for 1- and n- trials, respectively. The BMSE FOM is extremely slow, and to get decent results we had to make a specialised *main*. The important difference is that for the BMSE *main* the first stage uses the QFI as a FOM. This allows the algorithm to search in the right regions, before moving onto the BMSE. The FOM in *'main_3stage_timed_BMSE_n_trials'* is particularly slow, so in stage 2 we use an approximate (but faster) version of the FOM.

Modifying and optimising the genetic algorithm (GA)

Before modifying the hyperparameters and GA options we strongly recommend reading Matlab's documentation on their GA, and most importantly the Genetic Algorithm Options documentation here. Also see our paper and this blog. The main options and hyperparameters to modify are:

- The mutation function. Try *@mutationPower_Paul* and *@mutationPower_selection*
- The crossover function. Try *@crossoversinglepoint* or *@crossovertwopoint*, or just specify nothing to use Matlab's default
- The crossover fraction, *Cross*. Recommended to try: from 0.2 to 0.8
- The power of the mutation, *mutPower*. Recommended to try: from 1 to 30
- The tournament size, *tourn_size*. Recommended to try: 2, 4, 8, 16, 32.

The default values given in *'main_3stage'* were found by optimising for the FOM *'QFI / nbar'* and toolbox *'Sample_toolbox'*, which produced some of the results in Fig 4 of our paper. These default values therefore might not be at all suited for the other FOMs, though they did produce decent results for us!

Despite all these options and hyperparameters, by far the best way to improve your results is to increase the populations sizes (and therefore running time). As a challenging, try re-producing the results in Fig 4 of our paper with a small population size, say below 1000. For us, this was impossible, though we might have simply not found the best hyperparameters/options.

Adding FOMs to the algorithm

Take these steps to add a new FOM to the algorithm. The UI should automatically update to include your new FOM:

1. Create a function for your FOM that takes as input a density operator, and produces as output a number (the fitness value, FV). The algorithm is designed to maximise the FV, so if, for example, your FOM wants to be minimised, then you need to e.g. take the reciprocal of the original FV.
2. Put your function in the folder *Toolbox/Figures of Merit*
3. Clear the workspace
4. Load *Toolbox/Full Toolbox* (by double clicking on it)
5. Open the variable *FiguresOfMerit* (by double clicking on it)
6. Double click on cell (1,1). This should open a table with all the FOMs in.

7. Insert your FOM into the table by adding a new row to the bottom. The name goes in the first column, the number of modes it acts on in the second column, and the function handle in the third. Ensure your new row is the same format as the others.
8. Save the whole workspace as *Full Toolbox*. You need to overwrite the old one, so consider backing it up.
9. Now run '*Create_toolbox_and_FOM_with_UI*'. Your FOM should automatically appear below the other FOMs.

Adding States, Operators and Measurements to the Toolbox

This is similar to adding a FOM, but with a few more considerations. The functions for the states, operators and measurements need to be of a very specific form. A full guide to add these will be extremely long and convoluted, so we don't provide all the details here. We expect the easiest way to add a new state, operator or measurement is to modify an existing function or by combining and modifying a number of our existing functions.

For example, for states:

- Function form: *[state] = NameOfState(Settings_to_be_optimised, Settings_fixed, trunc)*
- *Settings_to_be_opimised* is a vector, where each element gives the value of that setting. For example, a coherent state has 2 free parameters, the magnitude and phase, so if we want a coherent state with magnitude 1 and phase 2 we choose: *Settings_to_be_opimised = alpha_abs_arg = [1, 2]*.
- *Settings_fixed* are fixed settings, that the algorithm won't optimise over. We don't have any fixed settings for our states, but say for the measurements we can include the loss in the measurement as a fixed setting. We want to specify, not optimise over, the loss, which is why it is a fixed setting.
- *trunc* = truncation
- Note that all our states in the folder *States* are single-mode. To add two-mode states, you need to specify the operator that produces the state: see the *'Two mode squeezed state'* for an example of this.
- All states are pure states in the Fock basis, in vector form (we only introduce mixed states during the measurement stage)

Some other notes on the functions:

- Operators must take *input_state* (which is a pure state vector) as their first argument, and output an *output_state*. This allows us to use the *expmv* function, as discussed in our paper.
- Operators need to specify which modes they act on, and how many modes the total state has
- The output of the *measurements*' functions are POVM elements (see our paper)

To add your function for your new state, operator or measurement to the toolbox, follow a similar procedure as adding FOMs:

1. Put your function in the appropriate folder
2. Clear the workspace
3. Load *Toolbox/Full Toolbox* (by double clicking on it)
4. Open the variable *FullToolbox* (by double clicking on it)
5. Locate the correct cell. Open the variable *ToolboxTypes* to see which cell the states/measurements/operators go in

6. Insert your new entry into the table, following the same format as previous entries. Open the variable *ToolboxHeadings* to see which details of the functions go where
7. Save the workspace as *Full Toolbox*. You need to overwrite the old one, so consider backing it up.
8. Now run '*Create_toolbox_and_FOM_with_UI'*. Your new element should automatically appear in the UI.