Quandary: Optimal Control for Open and Closed Quantum Systems

Stefanie Günther* N. Anders Petersson*

last updated May 20, 2024

Installation

Read the `README.md`! In short:

1. Install Petsc (`https://petsc.org/`), export `$PETSC_DIR` and `$PETSC_ARCH`, and append `$PETSC_DIR/$PETSC_ARCH` to your `$LD_LIBRARY_PATH`.
2. Compile the quandary executable with

   ```
   > make -j quandary
   ```

   and add it to your `$PATH`.
3. For the python interface, add the Quandary folder to your `$PYTHONPATH`.

Quick start

The C++ Quandary executable takes a configuration input file. As a quick start, test it with

```
> ./quandary config_template.cfg                    (serial execution)
> mpirun -np 4 ./quandary config_template.cfg       (on 4 cores)
```

You can silence Quandary by adding the `--quiet` command line argument.

Results are written as column-based text files in the output directory. Gnuplot is an excellent plotting tool to visualize the written output files, see below. The `config_template.cfg` is currently set to run a CNOT optimization test case. It lists all available options and configurations, and is filled with comments that should help users to set up new simulation and optimization runs, and match the input options to the equations found in this document.

Test the python interface by running one of the examples in `/examples/pythoninterface/`, e.g.

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA.

```
> python3 example_swap02.py
```

The python interpreter will start background processes on the C++ executable using a config file written by the python interpreter, and gathers quandary's output results back into the python shell for plotting.

Contents

1 Introduction **3**

2 Model equation **4**

2.1 Rotational frame approximation 5

2.2 Control pulses 6

2.2.1 Storage of the control parameters 7

2.2.2 Alternative control parameterization based on B-spline amplitudes and time-constant phases 8

2.3 Interfacing to Python environemnt 8

3 The Optimal Control Problem **9**

3.1 Fidelity 10

3.2 Objective function 10

3.3 Optimization targets 11

3.3.1 Pure target states 11

3.3.2 Arbitrary target state 12

3.3.3 Gate optimization 12

3.3.4 Essential and non-essential levels 13

3.4 Initial conditions 13

3.4.1 Pure-state initialization 13

3.4.2 Basis states 14

3.4.3 Diagonal density matrices (aka all pure states) 14

3.4.4 Ensemble state for pure-state optimization 15

3.4.5 Three initial states for gate optimization 15

3.4.6 $N + 1$ initial states for gate optimization 16

3.4.7 Reading an initial state from file 16

3.5 Penalty terms and leakage prevention 16

4 Implementation **17**

4.1 Vectorization of Lindblad's master equation 17

4.1.1 Real-valued system and state storage 18

4.2 Sparse-matrix vs. matrix-free solver 18

4.3 Time-stepping 19

4.3.1 Implicit Midpoint Rule (`IMR`) 19

4.3.2 Higher-order compositional IMR (`IMR4`, or `IMR8`) 19

4.3.3 Choice of the time-step size 19

4.4 Gradient computation via discrete adjoint backpropagation 20

4.5 Optimization algorithm 20

5 Parallelization **21**

6 Output and plotting the results **21**

6.1 Output options with regard to state evolution 22

6.2 Output with regard to simulation and optimzation 23

6.3 Plotting 23

7 Testing **23**

A Appendix: Details for the real-valued, vectorized Hamiltonian **24**

B Summary of all C++ configuration options **25**

C Summary of all python interface options **29**

1 Introduction

Quandary numerically simulates and optimizes the time evolution of closed and open quantum systems. The underlying dynamics are modelled by either Schroedinger's equation (for closed systems), or Lindblad's master equation (for open systems that interact with the environment). Quandary solves the respective ordinary differential equation (ODE) numerically by applying a time-stepping integration scheme, and applies a gradient-based optimization scheme to determine optimal control pulses that drive the quantum system to a desired target. The target can be a unitary, i.e. optimizing for pulses that realize a logical quantum operation, or state preparation that aims to drive the quantum system from one (or multiple) initial state to a desired target state, such as for example the ground state of zero energy level, or for the creation of entangled state.

Quandary is designed to solve optimal control problems in larger (potentially open) quantum systems, targeting modern high performance computing (HPC) platforms. Quandary utilizes distributed memory computations using the message passing paradigm that enables scalability to large number of compute cores. Implemented in C++, Quandary is portable and its object-oriented implementation allows developers to extend the predefined setup to suit their particular simulation and optimization requirements. For example, customized gates for Hamiltonian simulations can easily be added to supplement Quandary's predefined gate set. The Python interface allows for greater flexibility where custom Hamiltonian models can be used.

This document outlines the mathematical background and underlying equations, and summarizes their implementation and usage in Quandary. We also refer to our publications [3, 4].

2 Model equation

Quandary models composite quantum systems consisting of Q subsystems, with n_k energy levels for the k-th subsystem, $k = 0, \dots, Q-1$. The Hilberspace dimension is hence the product of each subsystem dimensions: $N = \prod_{k=0}^{Q-1} n_k$.

The default system Hamiltonian model for the composite system is

$$H_d := \sum_{k=0}^{Q-1} \left(\omega_k a_k^\dagger a_k - \frac{\xi_k}{2} a_k^\dagger a_k^\dagger a_k a_k + \sum_{l>k} \left(J_{kl} \left(a_k^\dagger a_l + a_k a_l^\dagger \right) - \xi_{kl} a_k^\dagger a_k a_l^\dagger a_l \right) \right) \tag{1}$$

where $\omega_k \geq 0$ denotes the $0 \to 1$ transition frequency and $\xi_k \geq 0$ is the self-Kerr coefficient of subsystem k, and the cross resonance coefficients are $J_{kl} \geq 0$ ("dipole-dipole interaction") and $\xi_{kl} \geq 0$ ("zz-coupling"). Here, $a_k \in \mathbb{C}^{N \times N}$ denotes the lowering operator acting on subsystem k, which is defined as

$$\begin{aligned} a_0 &:= a^{(n_0)} \otimes I_{n_1} \otimes \cdots \otimes I_{n_{Q-1}} \\ a_1 &:= I_{n_0} \otimes a^{(n_1)} \otimes \cdots \otimes I_{n_{Q-1}} \\ &\vdots \\ a_{Q-1} &:= I_{n_0} \otimes I_{n_1} \otimes \cdots \otimes a^{(n_{Q-1})} \end{aligned} \quad \text{with} \quad a^{(n_k)} := \begin{pmatrix} 0 & 1 & & & \\ & 0 & \sqrt{2} & & \\ & & \ddots & \ddots & \\ & & & & \sqrt{n_k - 1} \\ & & & & 0 \end{pmatrix} \in \mathbb{R}^{n_k \times n_k} \tag{2}$$

where $I_{n_k} \in \mathbb{R}^{n_k \times n_k}$ is the identity matrix.

The action of external control fields on the quantum system is modelled through the control Hamiltonian

$$H_c(t) := \sum_{k=0}^{Q-1} f^k(\vec{\alpha}^k, t) \left(a_k + a_k^\dagger \right) \tag{3}$$

where $f^k(\vec{\alpha}^k, t)$ are real-valued, time-dependent control functions that are parameterized by real-valued parameters $\vec{\alpha}^k \in \mathbb{R}^d$, which can be either specified, or optimized for.

For a **closed quantum system** (no environmental interactions), the quantum state is described by a complex-valued vector $\psi \in \mathbb{C}^N$, with $\|\psi\| = 1$. For a given initial state $\psi(t = 0)$, the evolution of the state vector is modelled through **Schroedinger's equation**

$$\dot{\psi}(t) = -iH(t)\psi(t), \quad \text{with} \quad H(t) := H_d + H_c(t). \tag{4}$$

Open quantum systems take interactions with the environment into account, allowing to model decoherence and noise in the system. In that case, the state of the quantum system is described by its density matrix $\rho \in \mathbb{C}^{N\times N}$, and the time-evolution is modelled by **Lindblad's master equation**:

$$\dot{\rho}(t) = -\, i(H(t)\rho(t) - \rho(t)H(t)) + \mathcal{L}(\rho(t)), \tag{5}$$

where again $H(t) = H_d + H_c(t)$, and where $\mathcal{L}(\rho(t))$ denotes the Lindbladian collapse operators to model system-environment interactions. The Lindbladian operator $\mathcal{L}(\rho(t))$ is assumed to be of the form

$$\mathcal{L}(\rho(t)) = \sum_{k=0}^{Q-1}\sum_{l=1}^{2} \mathcal{L}_{lk}\rho(t)\mathcal{L}_{lk}^{\dagger} - \frac{1}{2}\left(\mathcal{L}_{lk}^{\dagger}\mathcal{L}_{lk}\rho(t) + \rho(t)\mathcal{L}_{lk}^{\dagger}\mathcal{L}_{lk}\right) \tag{6}$$

where the collapse operators $\mathcal{L}_{lk}$ model decay and dephasing processes in the subsystem k with

- Decay ("T_1"): $\mathcal{L}_{1k} = \frac{1}{\sqrt{T_1^k}} a_k$
- Dephasing ("T_2"): $\mathcal{L}_{2k} = \frac{1}{\sqrt{T_2^k}} a_k^{\dagger} a_k$

The constants $T_l^k > 0$ correspond to the half-life of process l on subsystem k. Typical T_1 decay time is between $10 - 100$ microseconds (us). T_2 dephasing time is typically about half of T1 decay time.

All the above constants and system parameters can be specified in the first part of the configuration file that Quandary's executable takes as an input, compare `config_template.cfg`. Note that the main choice here is which equation should be solved for and which representation of the quantum state will be used (either Schroedinger with a state vector $\psi \in \mathbb{C}^N$, or Lindblad's equation for a density matrix $\rho \in \mathbb{C}^{N\times N}$). In the configuration file, this choice is determined through the option `collapse_type`, where `none` will result in Schroedinger's equation and any other choice will result in Lindblad's equation being solved for. Further note, that choosing `collapse_type` $\neq$ `none`, together with a collapse time $T_l^k = 0.0$ will omit the evaluation of the corresponding term in the Lindblad operator (6) (but will still solve Lindblad's equation for the density matrix).

Note: In the remainder of this document, the quantum state will mostly be denoted by ρ, independent of which equation is solved for. Depending on the context, ρ can then either denotes the density matrix $\rho \in \mathbb{C}^{N\times N}$, or the state vector $\psi \in \mathbb{C}^N$. A clear distinction between the two will only be made explicit if necessary.

2.1 Rotational frame approximation

Quandary uses the rotating wave approximation in order to slow down the time scales in the solution of Schroedinger's or Lindblad's master equations. To that end, the user can specify the rotation frequencies ω_k^r for each oscillator. Under the rotating frame wave approximation, the Hamiltonians

are transformed to

$$\tilde{H}_d(t) := \sum_{k=0}^{Q-1} (\omega_k - \omega_k^r)\, a_k^\dagger a_k - \frac{\xi_k}{2} a_k^\dagger a_k^\dagger a_k a_k - \sum_{l>k} \xi_{kl} a_k^\dagger a_k a_l^\dagger a_l$$
$$+ \sum_{k=0}^{Q-1} \sum_{l>k} J_{kl} \left(\cos(\eta_{kl} t) \left(a_k^\dagger a_l + a_k a_l^\dagger \right) + i \sin(\eta_{kl} t) \left(a_k^\dagger a_l - a_k a_l^\dagger \right) \right) \tag{7}$$

$$\tilde{H}_c(t) := \sum_{k=0}^{Q-1} \left(p^k(\vec{\alpha}^k, t)(a_k + a_k^\dagger) + i q^k(\vec{\alpha}^k, t)(a_k - a_k^\dagger) \right) \tag{8}$$

where $\eta_{kl} := \omega_k^r - \omega_l^r$ are the differences in rotational frequencies between subsystems.

Note that the eigenvalues of the rotating frame Hamiltonian become significantly smaller in magnitude by choosing $\omega_k^r \approx \omega_k$ (so that the first term with $a_k^\dagger a_k$ drops out). This slows down the time variation of the state evolution, hence bigger time-step sizes can be chosen when solving the master equation numerically. We remark that the rotating wave approximation ignores terms in the control Hamiltonian that oscillate with frequencies $\pm 2\omega_k^r$. Below, we drop the tildes on $\tilde{H}_d$ and $\tilde{H}_c$ and use the rotating frame definition of the Hamiltonians to model the system evolution in time.

Using the rotating wave approximatino, the real-valued laboratory frame control functions are written as

$$f^k(\vec{\alpha}^k, t) = 2\mathrm{Re}\left(d^k(\vec{\alpha}^k, t) e^{i\omega_k^r t} \right), \quad d^k(\vec{\alpha}^k, t) = p^k(\vec{\alpha}^k, t) + i q^k(\vec{\alpha}^k, t) \tag{9}$$

where the rotational frequencies ω_k^r act as carrier waves to the rotating-frame control functions $d^k(\vec{\alpha}^k, t)$.

2.2 Control pulses

The time-dependent rotating-frame control functions $d^k(\vec{\alpha}^k, t)$ are parameterized using N_s^k piecewise quadratic B-spline basis functions $B_s(t)$ acting as envelope for N_f^k carrier waves:

$$d^k(\vec{\alpha}^k, t) = \sum_{f=1}^{N_f^k} \sum_{s=1}^{N_s^k} \alpha_{s,f}^k B_s(t) e^{i\Omega_k^f t}, \quad \alpha_{s,f}^k = \alpha_{s,f}^{k(1)} + i\alpha_{s,f}^{k(2)} \in \mathbb{C} \tag{10}$$

The amplitudes $\alpha_{s,f}^{k(1)}, \alpha_{s,f}^{k(2)} \in \mathbb{R}$ are the control parameters (*design* variables) that Quandary can optimize in order to realize a desired system behavior, giving a total number of $2\sum_k N_s^k N_f^k$ real-valued optimization variables. (Note that the number of carrier wave frequencies N_f^k as well as the number of spline basis functions N_s^k can be different for each subsystem k.) $\Omega_k^f \in \mathbb{R}$ denote the carrier wave frequencies in the rotating frame which can be choosen to trigger certain system frequencies. The corresponding Lab-frame carrier frequencies become $\omega_k^r + \Omega_k^f$. Those frequencies can be chosen to match the transition frequencies in the lab-frame system Hamiltonian. For example, when $\xi_{kl} << \xi_k$, the transition frequencies satisfy $\omega_k - n\xi_k$. Thus by choosing $\Omega_k^f = \omega_k - \omega_k^r - n\xi_k$, one triggers transition between energy levels n and $n+1$ in subsystem k. Choosing effective carrier wave frequencies is quite important for optimization performance. We recommend to have a look at [5] for details on how to choose them.

Using trigonometric identities, the real and imaginary part of the rotating-frame control $d^k(\vec{\alpha}^k, t) = p^k(\vec{\alpha}^k, t) + iq^k(\vec{\alpha}^k, t)$ are given by

$$p^k(\vec{\alpha}^k, t) = \sum_{f=1}^{N_f^k} \sum_{s=1}^{N_s} B_s(t) \left(\alpha_{s,f}^{k(1)} \cos(\Omega_f^k t) - \alpha_{lsf}^{k(2)} \sin(\Omega_f^k t) \right) \tag{11}$$

$$q^k(\vec{\alpha}^k, t) = \sum_{f=1}^{N_f^k} \sum_{s=1}^{N_s} B_s(t) \left(\alpha_{s,f}^{k(1)} \sin(\Omega_f^k t) + \alpha_{s,f}^{k(2)} \cos(\Omega_f^k t) \right) \tag{12}$$

Those relate to the Lab-frame control $f^k(\vec{\alpha}^k, t)$ through

$$f^k(t) = 2 \sum_{f=1}^{N_f^k} \sum_{s=1}^{N_s} B_s(t) \left(\alpha_{s,f}^{k(1)} \cos((\omega_k^r + \Omega_f^k)t) - \alpha_{s,f}^{k(2)} \sin((\omega_k^r + \Omega_f^k)t) \right) \tag{13}$$

$$= 2p^k(\vec{\alpha}^k, t) \cos(\omega_k^r t) - 2q^k(\vec{\alpha}^k, t) \sin(\omega_k^r t) \tag{14}$$

$$= 2\mathrm{Re}\left(d^k(\vec{\alpha}^k, t) e^{i\omega_k^r t} \right) \tag{15}$$

2.2.1 Storage of the control parameters

The control parameters $\boldsymbol{\alpha}$ are stored in the Quandary code in the following order: List oscillators first $(\vec{\alpha}^0, \ldots, \vec{\alpha}^{Q-1})$, then for each $\vec{\alpha}^k \in \mathbb{R}^{2N_s N_f^k}$, iterate over all carrierwaves $\vec{\alpha}^k = (\alpha_1^k, \ldots, \alpha_{N_f}^k)$ with $\alpha_f^k \in \mathbb{R}^{2N_s^k}$, then each α_f^k iterates over spline basis functions listing first all real then all imaginary components: $\alpha_f^k = \alpha_{1,f}^{k(1)}, \ldots, \alpha_{N_s^k,f}^{k(1)}, \alpha_{1,f}^{k(2)}, \ldots, \alpha_{N_s^k,f}^{k(2)}$. Hence there are a total of $2\sum_k N_s^k N_f^k$ real-valued optimization parameters, which are stored in the following order:

$$\boldsymbol{\alpha} := \left(\vec{\alpha}^0, \ldots, \vec{\alpha}^{Q-1} \right), \in \mathbb{R}^{2\sum_k N_s^k N_f^k} \quad \text{where} \tag{16}$$

$$\vec{\alpha}^k = \Bigg(\alpha_{1,1}^{k(1)}, \ldots, \alpha_{N_s^k,1}^{k(1)}, \ldots, \alpha_{1,N_f^k}^{k(1)}, \ldots, \alpha_{N_s^k,N_f^k}^{k(1)} \tag{17}$$

$$\alpha_{1,1}^{k(2)}, \ldots, \alpha_{N_s^k,1}^{k(2)}, \ldots, \alpha_{1,N_f^k}^{k(2)}, \ldots, \alpha_{N_s^k,N_f^k}^{k(2)} \Bigg) \tag{18}$$

iterating over Q subsystems first, then N_f^k carrier wave frequencies, then N_s^k splines, listing first all real parts then all imaginary parts. To access an element $\alpha_{s,f}^{k(i)}$, $i = 0, 1$, from storage $\boldsymbol{\alpha}$:

$$\alpha_{s,f}^{k(i)} = \boldsymbol{\alpha}[\left(\sum_{j=0}^{k-1} 2N_s^j N_f^j \right) + f * 2N_s^k + s + i * N_s], \tag{19}$$

Note: this ordering of the controls is compatible with the order of control parameters in the Juqbox.jl software [5].

When executing Quandary, the control parameter $\boldsymbol{\alpha}$ can be either specified (e.g. a constant pulse, a pi-pulse, or pulses whose parameters are read from a given file), or can be optimized for (Section 3).

In order to guarantee that the optimizer yields control pulses that are bounded with $|p^k(t)| \leq c^k_{max}$, $|q^k(t)| \leq c^k_{max}$ for given bounds c^k_{max} for each subsystem $k = 0, \ldots, Q-1$, box constraints are implemented as:

$$|\alpha^{k(1)}_{s,f}| \leq \frac{c^k_{max}}{N^k_f} \quad \text{and} \quad |\alpha^{k(2)}_{s,f}| \leq \frac{c^k_{max}}{N^k_f}. \tag{20}$$

2.2.2 Alternative control parameterization based on B-spline amplitudes and time-constant phases

As an alternative to the above parameterization, we can parameterize only the *amplitudes* of the control pulse with B-splines, and add a time-constant phase per carrierwave:

$$d(t) = \sum_f e^{i\Omega_f t} a_f(t) e^{ib_f} \quad \text{where} \quad a_f(t) = \sum_s \alpha_{f,s} B_s(t) \tag{21}$$

$$\Rightarrow d(t) = \sum_f \sum_s \alpha_{f,s} B_s(t) e^{i\Omega_f t + b_f} \tag{22}$$

where the control parameters are $b_f \in [-\pi, \pi]$ (phases for each carrier wave) and the amplitudes $\alpha_{f,s} \in \mathbb{R}$ for $s = 1, \ldots, N_s$, $f = 1, \ldots, N_f$. Hence for Q oscillators, we have a total of $\sum_q (N^q_s + 1) N^q_f$ control parameters.

The rotating frame pulses are then given by

$$p(t) = \sum_f \sum_s \alpha_{f,s} \cos(\Omega_f t + b_f) B_s(t) \tag{23}$$

$$q(t) = \sum_f \sum_s \alpha_{f,s} \sin(\Omega_f t + b_f) B_s(t) \tag{24}$$

2.3 Interfacing to Python environemnt

You can use the Python interface for Quandary to simulate and optimize from within a python environment (version ≥ 3). It eases the use of Quandary, and adds some additional functionality, such as automatic computation of the required number of time steps, automatic choice of the carrier frequencies, and it allows for custom Hamiltonian models to be used (system and control Hamiltonian operators H_d and H_c). A good place to start is to have a look into the example `example_swap02.py`. This test case optimizes for a 3-level SWAP02 gate that swaps the state of the zero and the second energy level of a 3-level qudit.

All interface functions are defined in `quandary.py`, which should be imported at the beginning of the script. Hence, you should either have this file in your working directory, or append your python path with the location of that file. Most importantly, the pythoninterface defines the `Quandary` dataclass that gathers all configuration options and sets defaults. Default values are overwritten by user input either through the constructor call through `Quandary(<membervar>=<value>)` directly, or by accessing the member variables after construction and calling `config.update()` afterwards (compare `example_swap02.py`).

After setting up the configuration, you can evoke simulations or optimizations with `quandary.simulate/optimize(` Check out `help(Quandary)` to see all available user functions. Under the hood, those function writes

the required Quandary configuration files (`config.cfg`, etc.) to a data directory, then evokes (multiple) subprocesses to execute parallel C++ Quandary on that configuration file through the shell, and then loads the results from Quandary's output files back into the python interpreter. Plotting scripts are also provided, see the example scripts.

In addition to the standard Hamiltonian models as described in Section 2, the python interface allows for user-defined Hamiltonian operators H_d and H_c. Those are provided to Quandary through optional arguments to the python interface configuration `QuandaryConfig`. If those user-defined Hamiltonians are given, Quandary replaces the Hamiltonian operators in (7) (system Hamiltonian) and (8) (control Hamiltonian operators $a \pm a'$) by the provided matrices.

- The system Hamiltonian H_d must be real-valued, and time-independent. The units of the system Hamiltonian should be angular frequency (multiply 2π).
- For each oscillator, a maximum of one complex-valued control operator can be specified. Those should be provided in terms of their real and imaginary parts separately, e.g. the standard model control operators would be specified as $H^{re}_{c,k} = a_k + a_k^\dagger$ and $H^{im}_{c,k} = a_k - a_k^\dagger$, for each oscillator k. The real parts will be multiplied by the control pulses $p_k(t)$, while the imaginary parts will be multiplied by $iq_k(t)$ for each oscillator k, similar to the model in (8). Control Hamiltonian operators should be 'unit-free', since those units come in through the multiplied control pulses p and q.
- Note: While control Hamiltonian operators are optional, the system Hamiltonian is always required. (Set it to zero otherwise.)
- Note: The matrix-free solver can not be used when custom Hamiltonians are provided. The code will therefore be slower.

The python interface is set up such that it automatically computes the time-step size for discretizing the time domain, as well as the carrier wave frequencies that trigger system resonances. Note that the carrier wave frequency analysis are tailored for the standard Hamiltonian model, and those frequencies might need to be adapted when custom Hamiltonian operators are used (read the screen output). You can always check the written configuration file `config.cfg`, and the log to see what frequencies are being used, and potentially modify them.

To switch between the Schroedinger solver and the Lindblad solver, the optional T_1 decay and T_2 dephasing times can be passed to the python QuandaryConfig. For the Lindblad solver, the same collapse terms as defined in (6) will be added to the dynamical equation.

3 The Optimal Control Problem

In the most general form, Quandary can solve the following optimization problem

$$\min_{\boldsymbol{\alpha}} J(\{\rho_i^{target}, \rho_i(T)\}) + \gamma_1 \|\boldsymbol{\alpha}\|_2^2 + Penalty \tag{25}$$

where $\rho_i(T)$ denotes one or multiple quantum states evaluated at a final time $T > 0$, which solve either Lindblad's master equation (5) or Schroedinger's equation (4) in the rotating frame for initial conditions $\rho_i(0)$, as specified in Section 3.4, $i = 1, \ldots, n_{init}$. The first term in (25) minimizes an

objective function J (see Section 3.2) that quantifies the discrepancy between the realized states $\rho_i(T)$ at final time T driven by the current control $\boldsymbol{\alpha}$ and the desired target ρ_i^{target}, see Section 3.3. The second term is a standart Tikhonov regularization that can be added with parameter $\gamma_1 \geq 0$ in order to regularize the optimization problem (stabilize optimization convergence) by favoring solutions with small norm. Additional penalty terms, e.g. for leakage prevention, can be added, compare 3.5.

3.1 Fidelity

As a measure of optimization success, Quandary reports on the fidelity computed from

$$F = \begin{cases} \frac{1}{n_{init}} \sum_{i=1}^{n_{init}} \mathrm{Tr}\left(\left(\rho_i^{target}\right)^\dagger \rho_i(T)\right) & \text{if Lindblad} \\ \left|\frac{1}{n_{init}} \sum_{i=1}^{n_{init}} (\psi_i^{target})^\dagger \psi_i(T)\right|^2 & \text{if Schroedinger} \end{cases} \tag{26}$$

The fidelity is an average of Hilbert-Schmidt overlaps of the target states and the evolved states: for the density matrix, the Hilbert-Schmidt overlap is $\langle \rho^{target}, \rho(t)\rangle = \mathrm{Tr}\left(\left(\rho^{target}\right)^\dagger \rho(T)\right)$, which is *real* if both states are density matrices (which is always the case in Quandary, see definition of basis matrices). For the state vector (and Schroedingers solver), the Hilbert-Schmidt overlap is $\langle \psi^{target}, \psi(T)\rangle = (\psi^{target})^\dagger \psi$, which is complex. Note that in the fidelity above (and also in the corresponding objective function J_{trace}, the absolute value is taken *outside* of the sum, hence relative phases are taken into account.

Further note that this fidelity is averaged over the chosen initial conditions, so the user should be careful how to interpret this number. E.g. if one optimizes for a logical gate while choosing the three initial condition as in Section 3.4.5, the fidelity that is reported during optimization will be averaged over those three initial states, which is not sufficient to estimate the actual average fidelity over the entire space of potential intial states. It is advised to recompute the average fidelity **after** optimization has finished by propagating all basis states B_{kj} to final time T using the optimized control parameter, or by propagating only $N+1$ initial states to get an estimate thereof.

3.2 Objective function

The following objective functions can be used for optimization in Quandary (config option `optim_objective`):

$$J_{frobenius} = \sum_{i=1}^{n_{init}} \frac{\beta_i}{2} \left\|\rho_i^{target} - \rho_i(T)\right\|_F^2 \tag{27}$$

$$J_{trace} = \begin{cases} 1 - \sum_{i=1}^{n_{init}} \frac{\beta_i}{w_i} \mathrm{Tr}\left((\rho_i^{target})^\dagger \rho_i(T)\right) & \text{if Lindblad} \\ 1 - \left|\sum_{i=1}^{n_{init}} \beta_i (\psi_i^{target})^\dagger \psi_i(T)\right|^2 & \text{if Schroedinger} \end{cases} \tag{28}$$

$$J_{measure} = \sum_{i=1}^{n_{init}} \beta_i \mathrm{Tr}\left(N_m \rho(T)\right) \tag{29}$$

Here, β_i denote weights with $\sum_{i=1}^{n_{init}} \beta_i = 1$ that can be used to scale the contribution of each initial/target state i (default $\beta_i = 1/n_{init}$). $J_{Frobenius}$ measures (weighted average of) the frobenius norm between target and final states. J_{Trace} measures the (weighted) infidelity in terms of the

Hilbert-Schmidt overlap, compare the definition of fidelity in eq. (26). Here, $w_i = \mathrm{Tr}\left(\rho_i(0)^2\right)$ is the purity of the initial state. Both those measures are common for optimization towards a unitary gate transformation, for example. $J_{measure}$ is (only) useful when considering pure-state optimization, see Section 3.3.1. Here, $m \in \mathbb{N}$ is a given integer, and N_m is a diagonal matrix with diagonal elements being $|k-m|, k = 0, \dots N-1$

The distinction for Lindblad vs. Schroedingers solver is made explicit for J_{trace} above. The other two measures apply naturally to either the density matrix version solving Lindblad's equation, or the state vector version solving Schroedinger's equation with $\|\rho^{target} - \rho(T)\| = \|\psi^{target} - \psi(T)\|$ and $\mathrm{Tr}\left(N_m \rho(T)\right) = \psi(T)^\dagger N_m \psi(T)$.

3.3 Optimization targets

Here we describe the target states ρ^{target} that are realized in Quandary (C++ config option `optim_target`). Two cases are considered: State preparation, where the target state is the same for all initial conditions, and gate optimization, where the target state is a unitary transformation of the initial condition.

3.3.1 Pure target states

State preparation aims to drive the system from either one specific or from any arbitrary initial state to a common desired (fixed) target state. Quandary can optimize towards *pure* target states of the form

$$\psi^{target} = \boldsymbol{e}_m \quad \text{or} \quad \rho^{target} = \boldsymbol{e}_m \boldsymbol{e}_m^\dagger, \quad \text{for} \quad m \in \mathbb{N}_0 \quad \text{with} \quad 0 \leq m < N \tag{30}$$

where $\boldsymbol{e}_m$ denotes the m-th unit vector in $\mathbb{R}^N$. [1] The integer m refers to the $|m\rangle$-th state of the entire system under consideration with dimension N, which can be a composite of Q subsystems. In the configuration file however, the pure target state is specified by defining the desired pure target for *each* of the subsystems individually. For a composite system of Q subsystems with n_k levels each, a composite target pure state is specified by a list of integers m_k with $0 \leq m_k < n_k$ representing the pure target state in each subsystem k. The composite pure target state is then

$$\psi^{target} = |m_0 m_1 m_2 \dots m_{Q-1}\rangle \quad \text{aka} \quad \psi^{target} = \boldsymbol{e}_{m_0} \otimes \boldsymbol{e}_{m_1} \otimes \dots \otimes \boldsymbol{e}_{m_{Q-1}} \tag{31}$$

for unit vectors $\boldsymbol{e}_{m_k} \in \mathbb{R}^{n_k}$, and $\rho^{target} = \psi^{target}(\psi^{target})^\dagger$ for the density matrix. The composite-system index m is computed inside Quandary, from

$$m = m_0 \frac{N}{n_0} + m_1 \frac{N}{n_0 n_1} + m_2 \frac{N}{n_0 n_1 n_2} + \dots + m_{Q-1} \tag{32}$$

Depending on the choice for the initial conditions, optimization towards a pure target state can be used to realize either a simple state-to-state transfer (choosing one specific initial condition, $n_{init} = 1$), or to realize the more complex task of state preparation that drives *any* initial state to a common pure target state. For $m = 0$, the target state represents the ground state of the system

[1] We note that considering pure states of that specific form ($\boldsymbol{e}_m$ or $\boldsymbol{e}_m \boldsymbol{e}_m^\dagger$) is not a restriction, because any other pure target state can be transformed to this representation using a unitary change of coordinates (compare the Appendix in [4] for a more detailed description).

under consideration, which has important applications for quantum reset as well as quantum error correction. Driving *any* initial state to a common target will require to couple to a dissipative bath, which should be accounted for in the model setup. In the latter case, typically a full basis of initial conditions needs to be considered during the optimization ($n_{init} = N^2$ for density matrices). However, it is shown in [4], that if one chooses the objective function $J_{measure}$ with corresponding measurement operator N_m (see eq. (29)), one can reduce the number of initial conditions to only *one* being an ensemble of all basis states, and hence $n_{init} = 1$ independent of the system dimension N. Compare [4] for details, and Section 3.4.4.

3.3.2 Arbitrary target state

A specific (non-pure) target state ρ^{target} can also be used a a target. For the C++ code, such a target state is read from file. File format: The vectorized density matrix (columnwise vectorization) in the Lindblad case, or the state vector in the Schroedinger case, one real-valued number per row, first list all real parts, then list all imaginary parts (hence either $2N^2$ lines with one real number each, or $2N$ lines with one real number each). The configuration option should be pointing to the file location. For the Python interface, the target state can be passes as a numpy array.

3.3.3 Gate optimization

Quandary can be used to realize logical gate operations. In that case, the target state is not fixed across the initial states, but instead is a unitary transformation of each initial condition. Let $V \in \mathbb{C}^{N\times N}$ be a unitary matrix presenting a logical operation, the goal is to drive any initial state $\rho(0)$ to the unitary transformation $\rho^{target} = V\rho(0)V^\dagger$, or, in the Schroedinger case, drive any initial state $\psi(0)$ to the unitary transformation $\psi(T) = V\psi(0)$. In the C++ code, some default target gates that are currently available:

$$V_X := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad V_Y := \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad V_Z := \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad V_{Hadamard} := \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{33}$$

$$V_{CNOT} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad V_{SWAP} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad V_{QFT} = ... \tag{34}$$

as well as a multi-qubit generalization of the SWAP and the CNOT gate for general Q subsystems: The SWAP-0Q gate swaps the state of the first and the last qubit, while leaving all other oscillators in their respective initial state, and the CQNOT gate performs a NOT operation on the last qubit if all other qubits are in the one-state. New, user-defined gates can be added to the C++ code by augmenting the `Gate` class in the corresponding `.cpp` and `.hpp` files. The target gate matrix can also be read from file. The file is a simple text file that contains the vectorized target unitary matrix (columnwise vectorization), first all real parts, then all imaginary parts (giving a total of $2N^2$ real-valued numbers). It should be specified in the essential dimensions. For the python interface, the target unitary matrix is passed to Quandary through a numpy array.

For gate optimization, the first two objective function $J_{frobenius}$ and J_{trace} are appropriate. Since *any* initial quantum state should be transformed by the control pulses, typically a basis for the initial states should be considered ($n_{init} = N$ for Schroedinger solver, and $n_{init} = N^2$ for Lindblad

solver). In the Lindblad solver case, it has however been shown in [1] that it is enough to optimize with only three specific initial states ($n_{init} = 3$), independent of the Hilbert space dimension N. Those three states are set up in such a way that they can distinguish between any two unitaries in that Hilbert space. The three initial states are readily available in Quandary, see Section 3.4. Note that when optimizing with only those three initial states, it turns out that the choice of the weights β_i that weight the contribution from each initial state in the overall objective function strongly influences the optimization convergence. For faster convergence, it is often benefitial to emphasize on the first of the three initial conditions ($\rho_1(0)$ in Section 3.4.5), hence choosing β_1 (much) bigger than β_2 and β_3 (e.g. $\beta = \{20, 1, 1\}$ often works better than $\beta = \{1, 1, 1\}$, try it yourself). We refer to [1] for details. Note that the weights will be scaled internally such that $\sum_i \beta_i = 1$.

Target gates will by default be rotated into the computational frame (see Section 2). Alternatively, the user can specify the rotation of the target gate through the configuration option `gate_rot_freq` (list of floats).

3.3.4 Essential and non-essential levels

It is often useful, to model the quantum system with more energy levels than the number of levels that the target gate is defined on. For example when optimizing for a SWAP gate on two qubits, with $V_{SWAP} \in \mathbb{C}^{4\times 4}$, one might want to model each qubit with more than two energy levels in order to (a) model the (infinite dimensional) system with more accuracy by including more levels and (b) allow the system to transition through higher energy levels in order to achieve the target at final time T. In that case, the *essential* levels denote the levels that the target gate is defined on. To this end, Quandary provides the option to specify the number of essential energy levels n_k^e in addition to the number of energy levels n_k, where $n_k^e \leq n_k$ for each subsystem k. The quantum dynamics are then modelled with (more) energy levels with $N = \prod_k n_k$ and $\rho(t) \in \mathbb{C}^{N\times N}$ (or $\psi \in \mathbb{C}^N$), while the gate is defined in the essential level dimensions only: $V \in \mathbb{C}^{N_e \times N_e}$, $N_e = \prod_k n_k^e$. In the example above, $n_0^e = n_1^e = 2$ and hence $V_{SWAP} \in \mathbb{C}^{4\times 4}$, but one can choose the number of energy levels n_0 and n_1 to be bigger than 2 to when modelling the system dynamics.

To compute the objective function at final time T, the essential-dimensional gate is projected upwards to the full dimensions $\tilde{V} \in \mathbb{C}^{N\times N}$ by inserting identiy blocks for rows/columns that correspond to a non-essential level of either of the subsystems. Hence, a realization of the gate $\tilde{V}$ will not alter the occupation of higher (non-essential) energy level compared to their initial occupation at $t = 0$.

3.4 Initial conditions

The initial states $\rho_i(0)$ which are accounted for during optimization in eq. (25) can be specified with the configuration option `initialcondition`. Below are the available choices.

3.4.1 Pure-state initialization

One can choose to simulate and optimize for only one specific pure initial state (then $n_{init} = 1$). The initial density matrix is then composed of Kronecker products of pure states for each of the

subsystems. E.g. for a bipartite system with $n_1 \otimes n_2$ levels, one can propagate any initial pure state

$$\psi(0) = |m_1\rangle \otimes |m_2\rangle \quad \text{for}\, m_1 \in \{0, \dots, n_1 - 1\}, m_2 \in \{0, \dots, n_2 - 1\} \tag{35}$$

$$\text{and} \quad \rho(0) = \psi(0)\psi(0)^\dagger \tag{36}$$

Note that, again, in this notation $|m_1\rangle = \boldsymbol{e}_{m_1} \in \mathbb{R}^{n_1}$. The configuration input takes a list of the integers m_k for each subsystem.

3.4.2 Basis states

To span any possible initial state, an entire basis of states can be used as initial conditions. For open systems using the density matrix representation (Lindblad solver), the $n_{init} = N^2$ basis states as defined in [4] are implemented:

$$B^{kj} := \frac{1}{2}\left(\boldsymbol{e}_k\boldsymbol{e}_k^\dagger + \boldsymbol{e}_j\boldsymbol{e}_j^\dagger\right) + \begin{cases} 0 & \text{if}\, k = j \\ \frac{1}{2}\left(\boldsymbol{e}_k\boldsymbol{e}_j^\dagger + \boldsymbol{e}_j\boldsymbol{e}_k^\dagger\right) & \text{if}\, k < j \\ \frac{i}{2}\left(\boldsymbol{e}_j\boldsymbol{e}_k^\dagger - \boldsymbol{e}_k\boldsymbol{e}_j^\dagger\right) & \text{if}\, k > j \end{cases} \tag{37}$$

for all $k, j \in \{0, \dots, N-1\}$. These density matrices represent N^2 pure, linear independent states that span the space of all density matrices in this Hilberspace. For closed systems using the state vector representation (Schroedinger's solver), the basis states are the unit vector in $\mathbb{C}^N$, hence $n_{init} = N$ initial states $\boldsymbol{e}_i \in \mathbb{R}^N, i = 0, \dots N-1$.

When composite systems of multiple subsystems are considered, the user can provide a consecutive list of integer ID's to determine in which of the subsystems the basis states should be spanned. Other subsystems will then be initialized in the ground state.

Note: The basis states are spanned in the *essential dimensions* of the system, if applicable.

In order to uniquely identify the different initial conditions in the Quandary code and in the output files, a unique index $i \in \{0, \dots, N^2 - 1\}$ is assigned to each basis state with

$$B^i := B^{k(i),j(i)}, \quad \text{with} \quad k(i) := i \bmod N, \quad \text{and} \quad j(i) := \left\lfloor \frac{i}{N} \right\rfloor$$

(columnwise vectorization of a matrix of matrices $\left\{B^{kj}\right\}_{kj}$).

3.4.3 Diagonal density matrices (aka all pure states)

For density matrices (Lindblad solver), one can choose to propagate only those basis states that correspond to pure states of the form $\boldsymbol{e}_k\boldsymbol{e}_k^\dagger$, i.e. propagating only the B^{kk} in (37) for $k = 0, \dots, N-1$, and then $n_{init} = N$. For the Schroedinger solver, this is equivalent to all basis states.

Again, when composite systems of multiple subsystems are considered, the user can provide a consecutive list of integer ID's to determine in which of the subsystems the diagonal states should be spanned. Other subsystems will then be initialized in the ground state.

Note: the diagonal states are spanned in the *essential dimensions* of the system, if applicable.

3.4.4 Ensemble state for pure-state optimization

Only valid for the density matrix version, solving Lindblad's master equation.

For pure-state optimization using the objective function $J_{measure}$ (29), one can use the ensemble state

$$\rho_s(0) = \frac{1}{N^2} \sum_{i,j=0}^{N-1} B^{kj} \tag{38}$$

as the only initial condition for optimization or simulation ($\Rightarrow n_{init} = 1$). Since Lindblad's master equation is linear in the initial condition, and $J_{measure}$ is linear in the final state, propagating this single initial state yields the same target value as if one propagates all basis states spanning that space and averages their measure at final time T (compare [4]). To specify the ensemble state in Quandary for composite quantum systems with multiple subsystems, on can provide a list of integer ID's that determine in which of the subsystems the ensemble state should be spanned. Other subsystems will be initialized in the ground state.

To be precise: the user specifies a list of consecutive ID's $< k_0 >, \ldots, < k_m >$ with $0 \leq k_j \leq Q-1$ and $k_{j+1} = k_j + 1$, the ensemble state $\rho_s(0)$ will be spanned in the dimension given by those subsystems, $N_s = \prod_{j=0}^{m} n_{k_j}$ and $\rho_s(0) \in \mathbb{C}^{N_s \times N_s}$ with basis matrices B^{kj} spanned in $\mathbb{C}^{N_s \times N_s}$. The initial state that Quandary propagates is then given by

$$\rho(0) = \boldsymbol{e}_0 \boldsymbol{e}_0^\dagger \otimes \underbrace{\rho_s(0)}_{\in \mathbb{C}^{N_s \times N_s}} \otimes \boldsymbol{e}_0 \boldsymbol{e}_0^\dagger \tag{39}$$

where the first $\boldsymbol{e}_0$ (before the kronecker product) is the first unit vector in $\mathbb{R}^{\prod_{k=0}^{k_0-1}}$ (i.e. ground state in all preceding subsystems), and the second $\boldsymbol{e}_0$ (behind the kronecker products) is the first unit vector in the dimension of all subsequent systems, $\mathbb{R}^{\prod_{k=k_m+1}^{Q-1}}$.

Note: The ensemble state will be spanned in the *essential* levels of the (sub)system, if applicable, and will then be lifted up to the full dimesion by inserting zero rows and columns.

3.4.5 Three initial states for gate optimization

Only valid for the density matrix version, solving Lindblad's master equation.

When considering gate optimization, it has been shown in [1] that it is enough to consider only three specific initial states during optimization ($n_{init} = 3$), independent of the Hilber space dimension. Those three initial states are given by

$$\rho(0)_1 = \sum_{i=0}^{N-1} \frac{2(N-i+1)}{N(N+1)} \boldsymbol{e}_i \boldsymbol{e}_i^\dagger \tag{40}$$

$$\rho(0)_2 = \sum_{ij=0}^{N-1} \frac{1}{N} \boldsymbol{e}_i \boldsymbol{e}_j^\dagger \tag{41}$$

$$\rho(0)_3 = \frac{1}{N} I_N \tag{42}$$

where $I_N \in R^{N\times N}$ is the identity matrix. They are readily implemented in Quandary. Note that it is important to choose the weights $\beta_i, i = 1, 2, 3$ in the objective function appropriately to achieve fast convergence.

Note: The three initial states are spanned in the *full* dimension of the system, including non-essential levels. The theory for gate optimization with three initial states had been developed for considering *only* essential levels (the gate is defined in the same dimension as the system state evolution), and at this point we are not certain if the theory generalizes to the case when non-essential levels are present. It is advised to optimize on the full basis if non-essential levels are present (or work on the theory, and let us know what you find.). The same holds for $N + 1$ initial states below.

3.4.6 $N + 1$ initial states for gate optimization

Only valid for the density matrix version, solving Lindblad's master equation.

The three initial states from above do not suffice to estimate the fidelity of the realized gate (compare [1]). Instead, it is suggested in that same paper to choose $N+1$ initial states to compute the fidelity. Those $N + 1$ initial states consist of the N diagonal states B^{kk} in the Hilbertspace of dimension N, as well as the totally rotated state $\rho(0)_2$ from above. Quandary offers the choice to simulate (or optimize) using those initial states, then $n_{init} = N + 1$.

Note: The $N + 1$ initial states are spanned in the *full* dimension of the system, including non-essential levels, see above for 3-state initialization.

3.4.7 Reading an initial state from file

A specific initial state can also be read from file ($\Rightarrow n_{init} = 1$). Format: one column being the vectorized density matrix (vectorization is columnwise), or the state vector, first all real parts, then all imaginary parts (i.e. number of lines is $2N^2$ or $2N$, with one real-valued number per line).

This option is useful for example if one wants to propagate a specific *non-pure* initial state. In that case, one first has to generate a datafile storing that state (e.g. by simulating a system and storing the output), which can then be read in as initial condition.

3.5 Penalty terms and leakage prevention

Three optional penalty terms can be added to the objective function:

$$Penalty = \gamma_2/T \int_0^T P\left(\{\rho_i(t)\}\right) \mathrm{d}t + \gamma_3/T \int_0^T \|\partial_{tt}\mathrm{Pop}(\rho_i(t))\|^2 \mathrm{d}t + \gamma_4/T \int_0^T \sum_k |d^k(\alpha^k, t)|^2 \, dt \tag{43}$$

The first penalty term can be added with $\gamma_2 \geq 0$ to achieve a desired behavior over the entire time-domain $0 \leq t \leq T$, rather than only at the final time. If extra (non-essential levels are considered through the optimization for at least one oscillator ($n_k^e < n_k$ for at least k, compare Sec. 3.3.4),

then this term can be used to prevent leakage to higher energy levels that are not modelled. In particular, in that case the occupation of all *guard levels* are penalized through

$$P(\rho(t)) = \sum_r \|\rho(t)_{rr}\|_2^2 \tag{44}$$

where r iterates over all indices that correspond to a guard level (i.e. last non-essential energy level) of at least one of the subsystems, and $\rho(t)_{rr}$ denotes their corresponding population.

The second penalty term can be added with parameter $\gamma_3 \geq 0$ to encourage solutions whose populations vary slowly in time by penalizing the second derivative of the populations of the state.

The third penalty term can be added with parameter $\gamma_4 \geq 0$ to encourage small control pulse amplitudes by penalizing the control pulse energy. This term can be useful if hardware bounds are given for the control pulse amplitudes: Rather than include amplitude bounds on control pulse directly, which often leads to more non-convex optimization problems and convergence deterioration, one can utilize this penalty term to favor short control pulses with small amplitudes. Compare also [2] for its usage to determine minimal gate durations.

Note: All regularization and penalty coefficients γ_i should be chosen small enough so that they do not dominate the final-time objective function J. This might require some fine-tuning. It is recommended to always add $\gamma_1 > 0$, e.g. $\gamma_1 = 10^{-4}$, and add other penalties only if needed.

4 Implementation

4.1 Vectorization of Lindblad's master equation

When solving Lindblad's master equation (5), Quandary uses a vectorized representation of the density matrix with $q(t) := \text{vec}(\rho(t)) \in \mathbb{C}^{N^2}$ (columnwise vectorization). Using the relations

$$\text{vec}(AB) = (I_N \otimes A)\text{vec}(B) = (B^T \otimes I_N)\text{vec}(A) \tag{45}$$

$$\text{vec}(ABC) = (C^T \otimes A)\text{vec}(B) \tag{46}$$

for square matrices $A, B, C \in \mathbb{C}^{N\times N}$, the vectorized form of the Lindblad master equation is given by:

$$\dot{q}(t) = M(t)q(t) \quad \text{where} \tag{47}$$

$$M(t) := -i(I_N \otimes H(t) - H(t)^T \otimes I_N) + \sum_{k=0}^{Q-1}\sum_{l=1}^{2} \gamma_{lk}\left(\mathcal{L}_{lk} \otimes \mathcal{L}_{lk} - \frac{1}{2}\left(I_N \otimes \mathcal{L}_{lk}^T\mathcal{L}_{lk} + \mathcal{L}_{lk}^T\mathcal{L}_{lk} \otimes I_N\right)\right) \tag{48}$$

with $M(t) \in \mathbb{C}^{N^2\times N^2}$, and $H(t) = H_d(t) + H_c(t)$ being the rotating frame system and control Hamiltonians as in (7) and (8), respectively.

When solving Schroedinger's equation (4), Quandary operates directly on the state $q(t) := \psi(t) \in \mathbb{C}^N$ and solves (47) with $M(T) := -iH(t)$.

4.1.1 Real-valued system and state storage

Quandary solves the (vectorized) equation (47) in real-valued variables with $q(t) = u(t) + iv(t)$, evolving the real-valued states $u(t), v(t) \in \mathbb{R}^M$ for $M = N$ (Schroedinger's eq.) or $M = N^2$ (Lindblad's eq.) with

$$\dot{q}(t) = M(t)q(t) \quad \Leftrightarrow \quad \begin{bmatrix} \dot{u}(t) \\ \dot{v}(t) \end{bmatrix} = \begin{bmatrix} A(t) & -B(t) \\ B(t) & A(t) \end{bmatrix} \begin{pmatrix} u(t) \\ v(t) \end{pmatrix} \tag{49}$$

for real and imaginary parts $A(t) = \mathrm{Re}\,(M(t))$ and $B(t) = \mathrm{Im}\,(M(t))$.

The real and imaginary parts of $q(t)$ are stored in a colocated manner: For $q = u + iv$ with $u, v \in \mathbb{R}^M$, a vector of size $2M$ is stored that staggers real and imaginary parts behind each other for each component:

$$q = u + iv = \begin{bmatrix} u^1 \\ u^2 \\ \vdots \\ u^M \end{bmatrix} + i \begin{bmatrix} v^1 \\ v^2 \\ \vdots \\ v^M \end{bmatrix} \quad \Rightarrow \quad q_{store} = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ \vdots \\ u_M \\ v_M \end{bmatrix}$$

4.2 Sparse-matrix vs. matrix-free solver

In Quandary, two versions to evaluate the right hand side of Lindblad's equation, $M(t)q(t)$, of the vectorized real-valued system are available:

1. The *sparse-matrix solver* uses PETSc's sparse matrix format (sparse AIJ) to set up (and store) the time-independent building blocks inside $A(t)$ and $B(t)$. Sparse matrix-vector products are then applied at each time-step to evaluate the products $A(t)u(t) - B(t)v(t)$ and $B(t)u(t) + A(t)v(t)$.

   For developers, the appendix provides details on each term within $A(t)$ and $B(t)$ which can be matched to the implementation in the code (class `MasterEq`).

2. The *matrix-free solver* considers the state density matrix $\rho \in C^{N \times N}$ to be a tensor of rank $2Q$ (one axis for each subsystems for each matrix dimension, hence $2 \cdot Q$ axes). Instead of storing the matrices within $M(t)$, the matrix-free solver applies tensor contractions to realize the action of $A(t)$ and $B(t)$ on the state vectors.

   In our current test cases, the matrix-free solver is much faster than the sparse-matrix solver (about 10x), no surprise. However the matrix-free solver is currently only implemented for composite systems consisting of **2, 3, 4, or 5** subsystems.

   The matrix-free solver currently does not parallelize across the system dimension N, hence the state vector is **not** distributed (i.e. no parallel Petsc!). The reason why we did not implement that yet is that Q can often be large while each axis can be very short (e.g. modelling $Q = 12$ qubits with $n_k = 2$ energy levels per qubit), which yields a very high-dimensional tensor with very short axix. In that case, the standard (?) approach of parallelizing the tensor along its axes will likey lead to very poor scalability due to high communication overhead. We have not found a satisfying solution yet - if you have ideas, please reach out, we are happy to collaborate!

4.3 Time-stepping

To solve the (vectorized) master equation (47), $\dot{q}(t) = M(t)q(t)$ for $t \in [0, T]$, Quandary applies a time-stepping integration scheme on a uniform time discretization grid $0 = t_0 < \dots t_N = T$, with $t_n = n\delta t$ and $\delta t = \frac{T}{N}$, and approximates the solution at each discrete time step $q^n \approx q(t_n)$. The time-stepping scheme can be chosen in Quandary through the configuration option `timestepper`.

4.3.1 Implicit Midpoint Rule (`IMR`)

The implicit midpoint rule is a second-order accurate, symplectic time-stepping algorithm with Runge-Kutta scheme tableau $\begin{array}{c|c} 1/2 & 1/2 \\ \hline & 1 \end{array}$. Given a state q^n at time t_n, the update formula to compute q^{n+1} is hence

$$q^{n+1} = q^n + \delta t k_1 \quad \text{where}\, k_1 \,\text{solves} \quad \left(I - \frac{\delta t}{2} M^{n+1/2}\right) k_1 = M^{n+1/2} q^n \tag{50}$$

where $M^{n+1/2} := M(t_n + \frac{\delta t}{2})$. In each time-step, a linear equation is solved to get the stage variable k_1, which is then used it to update q^{n+1}.

4.3.2 Higher-order compositional IMR (`IMR4`, or `IMR8`)

A compositional version of the Implicit Midpoint Rule is available that performs multiple IMR steps in each time step interval, which are composed in such a way that the resulting compositional step is of higher order. Currently, Compared to the standard IMR, the higher-order methods can be very benefitial as it allows for much larger time-steps to be taken to reach a certain accuracy tolerance. Even though more work is done per timestep, the reduction in the number of time-steps needed can be several orders or magnitude and there is hence a tradeoff where the compositional methods outperform the standard IMR scheme.

Currently available is a compositional method of 4-th order that performs 3 sub-steps per time step (`IMR4`), and a compositional method of 8-th order performing 15 sub-steps per time step (`IMR8`).

4.3.3 Choice of the time-step size

The python interface to Quandary automatically computes a time-step size based on the fastest period of the system Hamiltonian. For the C++ code, it needs to be set by the user.

In order to choose a time-step size δt, an eigenvalue analysis of the constant drift Hamiltonian H_d is often useful. Since H_d is Hermitian, there exists a transformation Y such that $Y^\dagger H_d Y = \Lambda \quad$ where $\quad Y^\dagger = Y$ where Λ is a diagonal matrix containing the eigenvalues of H_d. Transform the state $\tilde{q} = Y^\dagger q$, then the ODE transforms to

$$\dot{\tilde{q}} = -i\Lambda\tilde{q} \quad \Rightarrow \dot{\tilde{q}}_i = -i\lambda_i \tilde{q}_i \quad \Rightarrow \tilde{q}_i = a\exp(-i\lambda_i t)$$

Therefore, the period for each mode is $\tau_i = \frac{2\pi}{|\lambda_i|}$, hence the shortest period is $\tau_{min} = \frac{2\pi}{\max_i\{|\lambda_i|\}}$. If we want p discrete time points per period, then $p\delta t = \tau_{min}$, hence

$$\delta t = \frac{\tau_{min}}{p} = \frac{2\pi}{p \max_i\{|\lambda_i|\}} \tag{51}$$

Usually, for the second order scheme we would use something like $p = 40$. The above estimate provides a first idea on how big (small) the time-step size should be, and the user is advised to consider this estimate when running a test case. However, the estimate ignores contributions from the control Hamiltonian, where larger control amplitudes will require smaller and smaller time-steps in order to resolve (a) the time-varying controls themselves and (b) the dynamics induced by large control contributions. A standard Δt test should be performed in order to verify that the time-step is small enough. For example, one can compute the Richardson error estimator of the current approximation error to some true quantity J^* from

$$J^* - J^{\Delta t} = \frac{J^{\Delta t} - J^{\Delta tm}}{1 - m^p} + O(\Delta t^{p+1}) \tag{52}$$

where p is the order of the timestepping scheme (i.e. $p = 2$ for the IMR and $p = 8$ for the compositional IMR8), and $J^{\Delta t}, J^{\Delta tm}$ denote approximations thereof using the time-stepping sizes Δt and Δtm for some factor m.

4.4 Gradient computation via discrete adjoint backpropagation

Quandary computes the gradients of the objective function with respect to the design variables $\boldsymbol{\alpha}$ using the discrete adjoint method. The discrete adjoint approach yields exact and consistent gradients on the algorithmic level, at costs that are independent of the number of design variables. To that end, the adjoint approach propagates local sensitivities backwards through the time-domain while concatenating contributions to the gradient using the chain-rule.

The consistent discrete adjoint time-integration step for adjoint variables denoted by $\bar{q}^n$ is given by

$$\bar{q}^n = \bar{q}^{n+1} + \delta t \left(M^{n+1/2}\right)^T \bar{k}_1 \quad \text{where}\, \bar{k}_1 \,\text{solves} \quad \left(I - \frac{\delta t}{2} M^{n+1/2}\right)^T \bar{k}_1 = \bar{q}^{n+1} \tag{53}$$

The contribution to the gradient ∇J for each time step is

$$\nabla J + = \delta t \left(\frac{\partial M^{n+1/2}}{\partial z} \left(q^n + \frac{\delta t}{2} k_1 \right) \right)^T \bar{k}_1 \tag{54}$$

Each evaluation of the gradient ∇J involes a forward solve of n_{init} initial quantum states to evaluate the objective function at final time T, as well as n_{init} backward solves to compute the adjoint states and the contributions to the gradient. Note that the gradient computation (54) requires the states and adjoint states at each time step. For the Schroedinger solver, the primal states are recomputed by integrating Schroedinger's equation backwards in time, alongside the adjoint computation. For the Lindblad solver, the states q^n are stored during forward propagation, and taken from storage during adjoint backpropagation (since we can't recompute it in case of Lindblad solver, due to dissipation).

4.5 Optimization algorithm

Quandary utilized Petsc's `Tao` optimization package to apply gradient-based iterative updates to the control variables. The `Tao` optimization interface takes routines to evaluate the objective function as well as the gradient computation. In the current setting in Quandary, `Tao` applies a nonlinear

Quasi-Newton optimization scheme using a preconditioned gradient based on L-BFGS updates to approximate the Hessian of the objective function. A projected line-search is applied to ensure that the objective function yields sufficient decrease per optimization iteration while keeping the control parameters within the prescribed box-constraints.

5 Parallelization

Quandary offers two levels of parallelization using MPI.

1. Parallelization over initial conditions: The n_{init} initial conditions $\rho_i(0)$ can be distributed over `np_init` compute units. Since initial condition are propagated through the time-domain for solving Lindblad's or Schroedinger's equation independently from each other, speedup from distributed initial conditions is ideal.
2. Parallel linear algebra with Petsc (sparse-matrix solver only): For the sparse-matrix solver, Quandary utilizes Petsc's parallel sparse matrix and vector storage to distribute the state vector onto `np_petsc` compute units (spatial parallelization). To perform scaling results, make sure to disable code output (or reduce the output frequency to print only the last time-step), because writing the data files invokes additional MPI calls to gather data on the master node.

Strong and weak scaling studies are presented in [4].

Since those two levels of parallelism are orthogonal, Quandary splits the global communicator (MPI_COMM_WORLD) into two sub-communicator such that the total number of executing MPI processes (np_{total}) is split as

$$np_{init} * np_{petsc} = np_{total}.$$

Since parallelization over different initial conditions is perfect, Quandary automatically sets $np_{init} = n_{init}$, i.e. the total number of cores for distributing initial conditions is the total number of initial conditions that are considered in this run, as specified by the configuration option `intialcondition`. The number of cores for distributed linear algebra with Petsc is then computed from the above equation.

It is currently required that the number of total cores for executing quandary is an integer divisor of multiplier of the number of initial conditions, such that each processor group owns the same number of initial conditions.

It is further required that the system dimension is an integer multiple of the number of cores used for distributed linear algebra from Petsc, i.e. it is required that $\frac{M}{np_{petsc}} \in \mathbb{N}$ where $M = N^2$ in the Lindblad solver case and $M = N$ in the Schroedinger case. This requirement is a little annoying, however the current implementation requires this due to the colocated storage of the real and imaginary parts of the vectorized state.

6 Output and plotting the results

Quandary generates various output files for system evolution of the current (optimized) controls as well as the optimization progress. All data files will be dumped into a user-specified folder through the config option `datadir`.

6.1 Output options with regard to state evolution

For each subsystem k, the user can specify the desired state evolution output through the config option `output<k>`:

- `expectedEnergy`: This option prints the time evolution of the expected energy level of subsystem k into files with naming convention `expected<k>.iinit<m>.dat`, where $m = 1, \ldots, n_{init}$ denotes the unique identifier for each initial condition $\rho_m(0)$ that was propagated through (see Section 3.4). This file contains two columns, the first row being the time values, the second one being the expectation value of the energy level of subsystem k at that time point, computed from

$$\langle N^{(n_k)} \rangle = \mathrm{Tr}\left(N^{(n_k)} \rho^k(t)\right) \tag{55}$$

  where $N^{(n_k)} = \left(a^{(n_k)}\right)^\dagger \left(a^{(n_k)}\right)$ denotes the number operator in subsystem k and ρ^k denotes the reduced density matrix for subsystem k, each with dimension $n_k \times n_k$. Note that this equivalent to $\mathrm{Tr}\left(N_k \rho(t)\right)$ with $N_k = I_{n_1} \otimes \cdots \otimes I_{n_{k-1}} \otimes N^{(n_k)} \otimes I_{n_{k+1}} \otimes \cdots \otimes I_Q$ and the full state $\rho(t)$ in the full dimensions $N \times N$.

- `expectedEnergyComposite` Prints the time evolution of the expected energy level of the entire (full-dimensional) system state into files (one for each initial condition, as above): $mboxTr\left(N\rho(t)\right)$ for the number operator N in the full dimensions.

- `population`: This option prints the time evolution of the state populations (diagonal of density matrix, state probabilities) of subsystem k into files named `population<k>.iinit<m>.dat` for each initial condition $m = 1, \ldots, n_{init}$. The files contain $n_k + 1$ columns, the first one being the time values, the remaining n_k columns correspond to the population of each level $l = 0, \ldots, n_k - 1$ of the reduced density matrix $\rho^k(t)$ at that time point. For Lindblad's solver, these are the diagonal elements of the reduced density matrix $(\rho^k_{ll}(t), l = 0, \ldots n_k - 1)$, for Schroedinger's solver it's the absolute values of the reduced state vector elements $|\psi^k_l(t)|^2, l = 0, \ldots n_k - 1$. Note that the reduction to the subsystem k induces a sum over all oscillators to collect contributions to the reduced state.

- `populationComposite`: Prints the time evolution of the state populations of the entire (full-dimensional) system into files (one for each initial condition, as above).

- `fullstate`: Probably only relevant for debugging or very small systems, one can print out the full state $\rho(t)$ or $\psi(t)$ for each time point into the files `rho_Re.iinit<m>.dat` and `rho_Im.iinit<m>.dat`, for the real and imaginary parts of the state, respectively. These files contain $N^2 + 1$ (Lindblad) or $N + 1$ (Schroedinger) columns the first one being the time point value and the remaining ones contain the vectorized density matrix (Lindblad, N^2 elements) or the state vector (Schroedinger, N elements) for each time step. Note that these file become very big very quickly – use with care!

The user can change the frequency of output in time (printing only every j-th time point) through the option `output_frequency`. This is particularly important when doing performance tests, as computing the reduced states for output requires extra computation and communication that might skew performance tests.

6.2 Output with regard to simulation and optimzation

- `config_log.dat` contains all configuration options that had been used for the current run.
- `params.dat` contains the control parameters $\boldsymbol{\alpha}$ that had been used to determine the current control pulses. This file contains one column containing all parameters, ordered as stored, see Section 2.2.
- `control<k>.dat` contain the resulting control pulses applied to subsystem k over time. It contains four columns, the first one being the time, second and third being $p^k(t)$ and $q^k(t)$ (rotating frame controls), and the last one is the corresponding lab-frame pulse $f^k(t)$. Note that the units of the control pulses are in frequency domain (divided by 2π). The unit matches the unit specified with the system paramters such as the qubit gound frequencies ω_k.
- `optim_history.dat` contains information about the optimization progress in terms of the overall objective function and contribution from each term (cost at final time T and contribution from the tikhonov regularization and the penalty term), as well the norm of the gradient and the fidelity, for each iteration of the optimization. If only a forward simulation is performed, this file still prints out the objective function and fidelity for the forward simulation.

Quandary always prints the current paramters and control pulses at the beginning of a simulation or optimization, and in addition at every l-th optimization iteration determined from the `optim_monitor_frequency` configuration option.

6.3 Plotting

The format of all output files are very well suited for plotting with Gnuplot, which is a command-line based plotting program that can output directly to screen, or into many other formats such as png, eps, or even tex. As an example, from within a Gnuplot session, you can plot e.g. the expected energy level of subsystem $k = 0$ for initial condition $m = 0$ by the simple command

```
gnuplot> plot 'expected0.iinit0000.dat' using 1:2 with lines title 'expected energy subsystem 0'
```

which plots the first against the second column of the file 'expected0.iinit0000.dat' to screen, connecting each point with a line. Additional lines (and files) can be added to the same plot by extending the above command with another file separated by comma (only omit the 'plot' keyword for the second command). There are many example scripts for plotting with gnuplot online, and as a starting point I recommend looking into some scripts in the 'quandary/util/' folder.

7 Testing

- Quandary has a set of regression tests. Please take a look at the REGRESSIONTEST.md document in the `quandary/tests` directory for instruction on how to run the regression tests.
- In order to check if the gradient implementation is correct, one can choose to run a Central Finite Difference test. Let the overall objective function be denoted by $F(\boldsymbol{\alpha})$. The Central Finite Difference test compares each element of the gradient $\nabla F(\boldsymbol{\alpha})$ with the following

(second-order accurate) estimate:

$$(\nabla F(\boldsymbol{\alpha}))_i \approx \frac{F(\boldsymbol{\alpha} + \epsilon \boldsymbol{e}_i) - F(\boldsymbol{\alpha} - \epsilon \boldsymbol{e}_i)}{2\epsilon} \qquad \text{(CFD)}$$

for unit vectors $\boldsymbol{e}_i \in \mathbb{R}^d$, and d being the dimension of $\boldsymbol{\alpha}$.

To enable the test, set the flag for the compiler directive `TEST_FD_GRAD` at the beginning of the `src/main.cpp` file. Quandary will then iterate over all elements in α and report the *relative* error of the implemented gradient with respect to the "true" gradient computed from CFD.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-SM-818073.

References

[1] Michael H Goerz, Daniel M Reich, and Christiane P Koch. Optimal control theory for a unitary operation under dissipative evolution. *New Journal of Physics*, 16(5):055012, 2014.

[2] Stefanie Günther and N Anders Petersson. A practical approach to determine minimal quantum gate durations using amplitude-bounded quantum controls. *AVS Quantum Science*, 5(4), 2023.

[3] Stefanie Günther, N. Anders Petersson, and Jonathan L. DuBois. Quandary: An open-source c++ package for high-performance optimal control of open quantum systems. https://arxiv.org/abs/2110.10310, 2021.

[4] Stefanie Günther, N. Anders Petersson, and Jonathan L. DuBois. Quantum optimal control for pure-state preparation using one initial state. *AVS Quantum Science*, 3:043801, 2021.

[5] N Anders Petersson and Fortino Garcia. Optimal control of closed quantum systems via b-splines with carrier waves. *arXiv preprint arXiv:2106.14310*, 2021.

A Appendix: Details for the real-valued, vectorized Hamiltonian

To assemble (evaluate) $A(t) = Re(M(t))$ and $B(t) = Im(M(t))$, consider

$$iH = iH_d(t) + iH_c(t) \tag{56}$$

$$= i\left(\sum_k (\omega_k - \omega_k^{\text{rot}}) a_k^\dagger a_k - \frac{\xi}{2} a_k^\dagger a_k^\dagger a_k a_k - \sum_{l>k} \xi_{kl} a_k^\dagger a_k a_l^\dagger a_l + \sum_{l>k} J_{kl} \cos(\eta_{kl} t) \left(a_k^\dagger a_l + a_k a_l^\dagger\right)\right. \tag{57}$$

$$\left. + \sum_k p^k(\vec{\alpha}^k, t) \left(a_k + a_k^\dagger\right)\right) \tag{58}$$

$$+ \left(\sum_k \sum_{kl} -J_{kl} \sin(\eta_{kl} t) \left(a_k^\dagger a_l - a_k a_l^\dagger\right) - \sum_k q^k(\vec{\alpha}^k, t) \left(a_k - a_k^\dagger\right)\right) \tag{59}$$

Hence $A(t)$ and $B(t)$ are given by

$$A(t) = A_d + \sum_k q^k(\vec{\alpha}^k, t) A_c^k + \sum_{l>k} J_{kl} \sin(\eta_{kl} t) A_d^{kl} \tag{60}$$

$$\text{with} \quad A_d := \sum_k \sum_{j=1,2} \gamma_{jk} \left(\mathcal{L}_{jk} \otimes \mathcal{L}_{jk} - \frac{1}{2} \left(I_N \otimes \mathcal{L}_{jk}^T \mathcal{L}_{jk} + \mathcal{L}_{jk}^T \mathcal{L}_{jk} \otimes I_N \right) \right) \tag{61}$$

$$A_c^k := I_N \otimes \left(a_k - a_k^\dagger \right) - \left(a_k - a_k^\dagger \right)^T \otimes I_N \tag{62}$$

$$A_d^{kl} := I_N \otimes \left(a_k^\dagger a_l - a_k a_l^\dagger \right) - \left(a_k^\dagger a_l - a_k a_l^\dagger \right)^T \otimes I_N \tag{63}$$

and

$$B(t) = B_d + \sum_k p^k(\vec{\alpha}^k, t) B_c^k + \sum_{kl} J_{kl} \cos(\eta_{kl} t) B_d^{kl} \tag{64}$$

$$\text{with} \quad B_d := \sum_k (\omega_k - \omega_k^{\text{rot}}) \left(-I_N \otimes a_k^\dagger a_k + (a_k^\dagger a_k)^T \otimes I_N \right) - \frac{\xi_k}{2} \left(-I_N \otimes a_k^\dagger a_k^\dagger a_k a_k + (a_k^\dagger a_k^\dagger a_k a_k)^T \otimes I_N \right) \tag{65}$$

$$- \sum_{l>k} \xi_{kl} \left(-I_N \otimes a_k^\dagger a_k a_l^\dagger a_l + (a_k^\dagger a_k a_l^\dagger a_l)^T \otimes I_N \right) \tag{66}$$

$$B_c^k := -I_N \otimes \left(a_k + a_k^\dagger \right) + \left(a_k + a_k^\dagger \right)^T \otimes I_N \tag{67}$$

$$B_d^{kl} := -I_N \otimes \left(a_k^\dagger a_l + a_k a_l^\dagger \right) + \left(a_k^\dagger a_l + a_k a_l^\dagger \right)^T \otimes I_N \tag{68}$$

$$\tag{69}$$

The sparse-matrix solver initializes and stores the constant matrices $A_d, A_d^{kl}, A_c^k, B_d, B_d^{kl}, B_c^k$ using Petsc's sparse-matrix format. They are used as building blocks to evaluate the blocks in the system matrix $M(t)$ with

$$A(t) = Re(M(t)) = A_d + \sum_k q^k(\alpha^k, t) A_c^k + \sum_{l>k} J_{kl} \sin(\eta_{kl} t) A_d^{kl} \tag{70}$$

$$B(t) = Im(M(t)) = B_d + \sum_k p^k(\alpha^k, t) B_c^k + \sum_{kl} J_{kl} \cos(\eta_{kl} t) B_d^{kl} \tag{71}$$

at each time t, which are applied to the vectorized, real-valued density matrix using Petsc's sparse MatVec implementation.

The matrix-free solver does not explicitly store the matrices A_d, B_d, A_c^k, B_c^k, etc., but instead only evaluates their action on a vector $q(t)$ using tensor contractions applied to the corresponding dimension of the density matrix tensor.

B Summary of all C++ configuration options

Here is a list of all options available to the C++ Quandary code, this is the same as in `config_template.cfg`.

```
/* ---------------------------------------------------------------- */
/* -------------------------- Testcase  -------------------------- */
/* ---------------------------------------------------------------- */
// Number of levels per subsystem
nlevels = 2, 2
// Number of essential levels per subsystem (Default: same as nlevels)
nessential = 2, 2
// Number of time steps used for time-integration
ntime = 1000
// Time step size (ns). Determines final time: T=ntime*dt
dt = 0.1
// Fundamental transition frequencies (|0> to |1> transition) for each oscillator
    ("\omega_k", multiplying a_k^d a_k, GHz)
transfreq = 4.10595, 4.81526
// Self-kerr frequencies for each oscillator ("\xi_k", multiplying a_k^d a_k^d a_k
     a_k, GHz)
selfkerr = 0.2198, 0.2252
// Cross-kerr coupling frequencies for each oscillator coupling k<->l ("\xi_kl",
    multiplying a_k^d a_k a_l^d a_l, GHz). Format: xi_01, xi_02, xi03, ... ,xi_12,
    xi_13, ...
crosskerr = 0.1
// Dipole-dipole coupling frequencies for each oscillator coupling k<->l ("J_kl",
    multiplying a_k^d a_l + a_k a_l^d, GHz). Format Jkl = J_01, J_02, ..., J12, J13
    , ...
Jkl = 0.0
// Rotational wave approximation frequencies for each subsystem ("\omega_rot", GHz
    ). Note: The target gate rotation can be specified separately with option "
    gate_rot_freq", see below.
rotfreq = 4.10595, 4.81526
// Switch between Schroedinger and Lindblad solver. 'none' solves Schroedinger
    solver (state vector dynamics), all other options solve Lindblads master
    equation (density matrix dynamics)
collapse_type = none
#collapse_type = decay
#collapse_type = dephase
#collapse_type = both
// Time of decay collapse operation (T1) per oscillator (gamma_1 = 1/T_1) (for
    Lindblad solver)
decay_time = 0.0, 0.0
// Time of dephase collapse operation (T2) per oscillator (gamma_2 = 1/T_2) (for
    Lindblad solver)
dephase_time = 0.0, 0.0
// Specify the initial conditions that are to be propagated
initialcondition = basis
#initialcondition = file, <path/to/initial_condition.dat>
#initialcondition = pure, 1, 0
#initialcondition = diagonal, 0
#initialcondition = ensemble, 0
#initialcondition = 3states
#initialcondition = Nplus1
// Apply a pi-pulse to oscillator <oscilID> from <tstart> to <tstop> using a
    control strength of <amp> rad/ns. This ignores the codes control parameters
    inside [tstart,tstop], and instead applies the constant control amplitude |p+iq
    |=<amp> to oscillator <oscilID>, and zero control for all other oscillators.
    Format per pipulse: 4 values: <oscilID (int)>, <tstart (double)>, <tstop (
    double)>, <amp(double)>. For more than one pipulse, just put them behind each
    other.
#apply_pipulse = 0, 0.5, 0.604, 15.10381
#apply_pipulse = 0, 0.5, 0.604, 15.10381, 1, 0.7, 0.804, 15.10381
```

```
/* ------------------------------------------------------------- */
/* ------------------- Optimization options -----------------------*/
/* ------------------------------------------------------------- */
// Define the controllable segments for each oscillator and the type of
    parameterization. Multiple segments can be listed behind each other, with
    corresponding starting and finish times.
// Format: <controltype>, <number of basis functions> [, <tstart>, <tstop>]
control_segments0 = spline, 150
control_segments1 = spline, 150
# control_segments0 = spline, 150, <tstart>, <tstop>
# control_segments0 = spline_amplitude, 150, 1.0
// Decide whether control pulses should start and end at zero. Default: true.
control_enforceBC=false
// Set the initial control pulse parameters (GHz). One option for each segmemt.
    Note: Reading the initialization from file applies to all subsystems, not just
    the one oscillator with that index, i.e. the file should contain all parameters
     for all oscillators in one long column.
control_initialization0 = constant, 0.005
control_initialization1 = constant, 0.005
#control_initialization0 = constant, 0.005
#control_initialization0 = file, ./params.dat
#control_initialization0 = constant, <amp_init>, <phase_init>
// Maximum amplitude bound for the control pulses for each oscillator (GHz). One
    value for each segment.
control_bounds0 = 0.008
control_bounds1 = 0.008
// Carrier wave frequencies for each oscillator 0..Q-1. (GHz)
carrier_frequency0 = 0.0, -0.2198, -0.1
carrier_frequency1 = 0.0, -0.2252, -0.1
// Optimization target
optim_target = gate, cnot
#optim_target = gate, cqnot
#optim_target = gate, swap
#optim_target = gate, swap0q
#optim_target = gate, qft
#optim_target = gate, xgate
#optim_target = gate, hadamard
#optim_target = gate, file, /path/to/target_gate.dat
#optim_target = pure, 0, 0
#optim_target = file, /path/to/target_state.dat
// Frequency of rotation of the target gate, for each oscillator (GHz). Default:
    Use the computational rotating frame (rotfreq).
# gate_rot_freq = [0.0,0.0]
// Objective function measure
optim_objective = Jtrace
# optim_objective = Jfrobenius
# optim_objective = Jmeasure
// Weights for summing up the objective function (beta_i). If less numbers than
    oscillators are given, the last one will e propagated to the remaining ones.
optim_weights = 1.0
# optim_weights = 0.5, 0.5
// Optimization stopping tolerance based on gradient norm (absolute: ||G|| < atol
    )
optim_atol = 1e-7
// Optimization stopping tolerance based on gradient norm (relative: ||G||/||G0||
    < rtol )
optim_rtol = 1e-8
```

```
// Optimization stopping criterion based on the final time cost (absolute: J(T) <
    ftol)
optim_ftol = 1e-5
// Optimization stopping criterion based on the infidelity (absolute: 1-Favg <
    inf_tol)
optim_inftol = 1e-5
// Maximum number of optimization iterations
optim_maxiter = 200
// Coefficient (gamma_1) of Tikhonov regularization for the design variables (
    gamma_1/2 || design ||^2)
optim_regul   = 0.00001
// Coefficient (gamma_2) for adding first integral penalty term (gamma_1 \int_0^T
    P(rho(t) dt )
optim_penalty = 0.0
// integral penalty parameter inside the weight in P(rho(t)) (gaussian variance a)
optim_penalty_param = 0.0
// Coefficient (gamma_3) for penalizing the integral of the second derivative of
    state populations (gamma_3 \int_0^T d^2/dt^2(Pop(rho)) dt )
optim_penalty_dpdm = 0.0
// Coefficient (gamma_4) for penalizing the control pulse energy integral (gamma_4
     \int_0^T p^2 + q^2 dt )
optim_penalty_energy= 0.0
// Switch to use Tikhonov regularization with ||alpha - alpha_0||^2 instead of ||
    alpha||^2
optim_regul_tik0=false


/* ---------------------------------------------------------------- */
/* ------------------- Output and runtypes -----------------------*/
/* ---------------------------------------------------------------- */
// Directory for output files
datadir = ./data_out
// Specify the desired output for each oscillator, one line per oscillator. Format
    : list of either of the following options:
//"expectedEnergy" - time evolution of the expected energy level for this
    oscillator (expected energy of the reduced density matrix)
//"expectedEnergyComposite" - time evolution of expected energy level of the full-
    dimensional composite system
//"population" - time evolution of the energy level populations (probabilities)
    for this oscillator (diagonals of the reduced density matrix)
//"populationComposite" - time evolution of the energy level population (
    probabilities) for the full-dimensional composite system
//"fullstate" - time-evolution of the full state of the composite system (full
    density matrix, or state vector) (note: 'fullstate' can appear in *any* of the
    lines). WARNING: This might result in *huge* output files! Use with care.
output0 = population, expectedEnergy
output1 = population, expectedEnergy
// Output frequency in the time domain: write output every <num> time-step
output_frequency = 1
// Frequency of writing output during optimization: write output every <num>
    optimization iterations.
optim_monitor_frequency = 1
// Runtype options: a forward simulation only, forward simulation and backward
    simulation for gradient, or "optimization" to run a full optimization cycle
#runtype = simulation
#runtype = gradient
runtype = optimization
// Use matrix free solver, instead of sparse matrix implementation. Only available
     for 2,3,4, or 5 oscillators.
```

```
usematfree = true
// Solver type for solving the linear system at each time step
linearsolver_type = gmres
# linearsolver_type = neumann
// Set maximum number of iterations for the linear solver
linearsolver_maxiter = 20
// Switch the time-stepping algorithm. Currently available:
// "IMR" - Implicit Midpoint Rule (IMR) of 2nd order,
// "IMR4" - Compositional IMR of order 2 using 3 stages,
// "IMR8" - Compositional IMR of order 8 using 15 stages,
timestepper = IMR
// For reproducability, one can choose to set a fixed seed for the random number
    generator. Comment out, or set negative if seed should be random (non-
    reproducable)
rand_seed = 1234
```

C Summary of all python interface options

Here is a list of all options available to the python interface, this is the same as in `quandary.py`.

```
# Quantum system specifications
Ne            # Number of essential energy levels per qubit. Default: [3]
Ng            # Number of extra guard levels per qubit. Default: [0] (no guard
    levels)
freq01        # 01-transition frequencies [GHz] per qubit. Default: [4.10595]
selfkerr      # Anharmonicities [GHz] per qubit. Default: [0.2198]
rotfreq       # Frequency of rotations for computational frame [GHz] per qubit.
     Default: freq01
Jkl           # Dipole-dipole coupling strength [GHz]. Formated list [J01, J02,
     ..., J12, J13, ...] Default: 0
crosskerr     # ZZ coupling strength [GHz]. Formated list [g01, g02, ..., g12,
    g13, ...] Default: 0
T1            # Optional: T1-Decay time [ns] per qubit (invokes Lindblad solver
    ). Default: 0
T2            # Optional: T2-Dephasing time [ns] per qubit (invokes Lindlbad
    solver). Default: 0

# Optional: User-defined system and control Hamiltonian operators. Default:
    Superconducting Hamiltonian model
Hsys                # Optional: User specified system Hamiltonian model. Array
    .
Hc_re               # Optional: User specified control Hamiltonian operators
    for each qubit (real-parts). List of Arrays
Hc_im               # Optional: User specified control Hamiltonian operators
    for each qubit (real-parts) List of Arrays
standardmodel       # Internal: Bool to use standard Hamiltonian model for
    superconduction qubits. Default: True

# Time duration and discretization options
T             # Pulse duration (simulation time). Default: 100ns
Pmin          # Number of discretization points to resolve the shortest period
    of the dynamics (determines <nsteps>). Default: 150
nsteps        # Number of time-discretization points (will be computed
    internally based on Pmin, or can be set here)
timestepper   # Time-discretization scheme. Default: "IMR"
```

```
# Optimization targets and initial states options
targetgate          # Complex target unitary in the essential level dimensions
     for gate optimization. Default: none
targetstate         # Complex target state vector for state-to-state
    optimization. Default: none
initialcondition    # Choose from provided initial states at time t=0.0: "
    basis" (all basis states, default), "pure, 0,0,1,..." (one pure initial
    state |001...>), or pass a vector as initial state. Default: "basis"
gate_rot_freq       # Specify frequencies to rotate a target gate (one per
    oscillator). Default: Using the computational frame rotation frequency (
    rotfreq).

# Control pulse options
pcof0               # Optional: Pass an initial vector of control parameters.
    Default: none
pcof0_filename      # Optional: Load initial control parameter vector from a
    file. Default: none
randomize_init_ctrl # Randomize the initial control parameters (will be
    ignored if pcof0 or pcof0_filename are given). Default: True
initctrl_MHz        # Amplitude [MHz] of initial control parameters. Float or
    List[float]. Default: 10 MHz.
maxctrl_MHz         # Amplitude bounds for the control pulses [MHz]. Float or
    List[float]. Default: none
control_enforce_BC  # Bool to let control pulses start and end at zero.
    Default: False
dtau                # Spacing of Bspline basis functions [ns]. The smaller
    dtau, the larger nsplines. Default: 3ns
nsplines            # Number of Bspline basis functions. Default: T/dtau + 2
carrier_frequency   # Carrier frequencies for each oscillator. List[List[float
    ]]. Default will be computed based on Hsys.
cw_amp_thres        # Threshold to ignore carrier wave frequencies whose
    growth rate is below this value. Default: 1e-7
cw_prox_thres       # Threshold to distinguish different carrier wave
    frequencies from each other. Default: 1e-2

# Optimization options
maxiter             # Maximum number of optimization iterations. Default 200
tol_infidelity      # Optimization stopping criterion based on the infidelity.
     Default 1e-5
tol_costfunc        # Optimization stopping criterion based on the objective
    function value. Default 1e-4
costfunction        # Cost function measure: "Jtrace" or "Jfrobenius". Default
    : "Jtrace"
optim_target        # Optional: Set other optimization target string, if not
    specified through the targetgate or targetstate.
gamma_tik0          # Parameter for Tikhonov regularization ||alpha||^2.
    Default 1e-4
gamma_tik0_interpolate # Switch to use ||alpha-alpha_0||^2 instead, where
    alpha_0 is the initial guess. Default: False
gamma_leakage       # Parameter for leakage prevention. Default: 0.1
gamma_energy        # Parameter for integral penality term on the control
    pulse energy. Default: 0.1
gamma_dpdm          # Parameter for integral penality term on second state
    derivative. Default: 0.01

# General options
rand_seed            # Set a fixed random number generator seed. Default: None
     (non-reproducable)
```

```
print_frequency_iter # Output frequency for optimization iterations. (Print
    every <x> iterations). Default: 1
usematfree           # Switch to use matrix-free (rather than sparse-matrix)
    solver. Default: True
verbose              # Switch to turn on more screen output for debugging.
    Default: False

Internal variables.
-------------------
_ninit                : int  # number of initial conditions that are
    propagated
_lindblad_solver      : bool # Flag to determine whether lindblad solver vs
    schroedinger solver
_hamiltonian_filename : str
_gatefilename         : str
_initstatefilename    : str
_initialstate         : List[complex] = field(default_factory=list)


Output parameters, available after Quandary has been executed (simulate or
    optimze)
-----------------------------------------------------
popt         # Optimized control palamters (Bspline coefficients). List[float]
time         # Vector of discretized time points. List[float]
optim_hist   # Optimization history: all fields as in Quandary's output file
    optim_history.dat. Dictionary.
```