Quandary: Optimal Control for Open and Closed Quantum Systems

Stefanie Günther* N. Anders Petersson*

last updated July 27, 2022

Contents

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

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 state. Two optimization objectives are considered: Unitary gate optimization that finds controls to realize a logical quantum gate, and state preparation that aims to drive the quantum system to a desired target state, such as for example the ground state of zero energy level.

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 easily 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.

This document outlines the mathematical background and underlying equations, and summarizes their implementation in Quandary. For installation instructions and execution, please take a look at the `README.md` file. Further, the configuration file `config_template.cfg` is filled with comments that should help users and developers to set up new simulation and optimization runs, and match the input options to the equations found in this document. A full documentation is under development. In the meantime, don't hesitate to direct any questions to guenther5@llnl.gov. We also refer to our publications [2, 3].

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 Hamiltonian describing the composite system is modelled as

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

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

$$\begin{aligned}\tilde{H}_d(t) := &\sum_{k=0}^{Q-1} \left(\omega_k - \omega_k^r\right) a_k^\dagger a_k - \frac{\xi_k}{2} a_k^\dagger a_k^\dagger a_k a_k - \sum_{l>k} \xi_{kl} a_k^\dagger a_k a_l^\dagger a_l \\ &+ \sum_{k=0}^{Q-1} \sum_{l>k} J_{kl} \left(\cos(\eta_{kl} t) \left(a_k^\dagger a_l + a_k a_l^\dagger \right) + i \sin(\eta_{kl} t) \left(a_k^\dagger a_l - a_k a_l^\dagger \right) \right) \qquad (7) \\ \tilde{H}_c(t) := &\sum_{k=0}^{Q-1} \left(p^k(\vec{\alpha}^k, t)(a_k + a_k^\dagger) + i q^k(\vec{\alpha}^k, t)(a_k - a_k^\dagger) \right) \qquad (8)\end{aligned}$$

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

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

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:

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

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 [4] 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 $(\vec{\alpha}^0, \dots, \vec{\alpha}^{Q-1})$, then for each $\vec{\alpha}^k \in \mathbb{R}^{2N_s N_f^k}$, iterate over all carrierwaves $\vec{\alpha}^k = (\alpha_1^k, \dots, \alpha_{N_f}^k)$ 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: $\alpha_f^k = \alpha_{1,f}^{k(1)}, \dots, \alpha_{N_s^k,f}^{k(1)}, \alpha_{1,f}^{k(2)}, \dots, \alpha_{N_s^k,f}^{k(2)}$. 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:

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

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

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

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 software [4].

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:

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

2.3 Python interface for general Hamiltonian systems

The Python interface for Quandary allows to simulate and optimize with user-defined Hamiltonians. To that end, the user defines (arbitrary) system and control Hamiltonian operators in a python script (see below), and Quandary then replace the Hamiltonian model in (7) (system Hamiltonian) and (8) (control Hamiltonian) by the Hamiltonians defined in the script.

To enable the python interface for Quandary, use the configuration option `python_file` to pass the name of the python script to Quandary. (Note: Make sure that the beginning of the configuration file (`nlevels`) matches the dimension of the Hamiltonians defined in the python file.) During Quandary's initialization phase, the python script will be parsed for functions of specific names that return the corresponding Hamiltonians. A description of the interface functions is given below. A good place to start is by having a look at the example in `quandary/results/python_interface_example/`. There, the python script `Hamiltonian.py` provides the interface functions that reproduce the (default) Hamiltonian model and control as in (7), (8) for a system of two coupled qudits, applying a constant control pulse of a certain duration and amplitude to qudit 0 (a non-perfect pi-pulse).

The python interface allows for general Hamiltonians that are composed of a constant (time-independent) system Hamiltonian H_d, (multiple) time-dependent system Hamiltonians $H_d(t)$ and (multiple) time-dependent control Hamiltonians $H_c(t)$:

$$H(t) = H_d + H_d(t) + H_c(t) \tag{21}$$

The time-dependency is realized through real-valued transfer functions that multiply constant matrices, see below.

- **Time-independent system Hamiltonian** H_d (required)
  The constant system Hamiltonian is assumed to be real-valued (for now): $H_d \in \mathbb{R}^{N\times N}$. It contains the time-independent (non-controllable) system Hamiltonian terms (detuning, anharmonicity, etc. and coupling terms, if they are time-constant). Quandary parses the python script for a function `getHd`, which must return a list of floats containing the vectorized matrix H_d (column-major).

  python function
  ```
  def getHd():
      [...]
      return vec(H_d)
  ```

- **Time-dependent system Hamiltonians** $H_d(t)$ (optional)
  Multiple time-dependent system Hamiltonian terms can be defined through these interface functions (e.g. such as the Jaynes-Cummings coupling terms in the rotating frame). Each term, $H_d^l(t) \in \mathbb{C}^{N\times N}, l = 0, 1, \ldots$ is complex, and their real and imaginary parts are composed of real-valued (transfer) functions $u^l(t), v^l(t)$ multiplying constant real and imaginary parts of the Hamiltonian matrix $\mathrm{Re}(H_d^l)$, $\mathrm{Im}(H_d^l)$:

$$H_d^l(t) = u^l(t)\mathrm{Re}\left(H_d^l\right) + iv^l(t)\mathrm{Im}\left(H_d^l\right) \tag{22}$$

  for some arbitrary number of terms $l = 0, 1, \ldots, L-1$. Providing time-dependent Hamiltonians is optional.

  The real-valued and constant matrices $\mathrm{Re}(H_d^l), \mathrm{Im}(H_d^l) \in \mathbb{R}^{N\times N}$, are passed to Quandary throught the interface functions `getHdt_real()` and `getHdt_imag()`, respectively, which both return a list (of length L) of matrices. Similar to H_d, the matrices each are passed as lists of floats (column-wise vectorization of the matrix). (So the functions return a list of lists of floats).

  python functions

```
def getHdt_real():
    [...]
    return [ vec(Re(H_d^{l=0})), vec(Re(H_d^{l=1})), ...]
def getHdt_imag():
    [...]
    return [ vec(Im(H_d^{l=0})), vec(Im(H_d^{l=1})), ...]
```

  The transfer functions are real-valued, $u^l, v^l \colon \mathbb{R} \to \mathbb{R}$. These are provided in terms of a spline representation. Quandary receives them through the interface functions `getHdtTransfer_real()` for $u^l(t)$ and `getHdtTransfer_imag()` for $v^l(t)$, which return a triple [knots, coefficients, order], for lists of spline knots and coefficients and an integer order. These triples should be provided for each of the defined matrices H_d^l. Providing transfer functions for H_d^l is optional. If an interface function `getHdtTransfer_real/imag()` is not defined, even though the corresponding H_d^l has been specified, then the transfer function will be constant 1.0. (Note that the corresponding Hamiltonian term will be constant then, and could have gone into the time-independent system Hamiltonian H_d...)

  python functions

```
def getHdtTransfer_real():
    [...]
    return [ [knots, coeffs, order]^{l=0},[knots, coeffs, order]^{l=1} , ...]
def getHdtTransfer_imag():
    [...]
    return [ [knots, coeffs, order]^{l=0},[knots, coeffs, order]^{l=1} , ...]
```

- **Time-dependent control Hamiltonians** $H_c(t)$**:** (optional)
  For each modeled subsystem $k = 0, \ldots, Q-1$, a number of control Hamiltonians can be applied. Each control Hamiltonian is driven with a separate set of control pulses $p^k(t)$, $q^k(t)$, which Quandary optimizes for. The control pulses can be applied to *multiple* control Hamiltonian matrices per subsystem k: $H_c^{k,l} \in \mathbb{C}^{N\times N}$ for $l = 0, \ldots, C^k - 1$, where the number of control Hamiltonians C^k can be different amongst the oscillators k. Real-valued transfer

functions can optionally be applied to the control pulses before multiplying the constant control matrices:

$$H_c^{k,l}(t) = u^{k,l}\left(p^k(t)\right)\mathrm{Re}\left(H_c^{k,l}\right) + iv^{k,l}\left(q^k(t)\right)\mathrm{Im}\left(H_c^{k,l}\right) \tag{23}$$

for $l = 0, \ldots, C^k - 1$. Note that *one* set of control pulses $p^k(t), q^k(t)$ is applied to *all* terms $H_c^{k,l}, l = 0, \ldots, C^k - 1$ for that subsystem k, however the corresponding transfer functions $u^{k,l}, v^{k,l}$ can be different amongst each control term l (and each oscillator k).

The interface functions `getHc_real()/imag()` return a list over oscillators, containing a list of control Hamiltonian matrices for each oscillator. Again, each matrix is passed as a list of floats containing the vectorized matrix elements (column-wise vectorization), so the return is a list (iterating over oscillators) of lists (iterating over control terms) of vectorized Hamiltonians

python functions

```
def getHc_real/imag():
    [...]
    return [  vec(Re/Im(H_c^{k=0,l=0})), vec(Re/Im(H_d^{k=0,l=1})), ...,
                    ⋮
              vec(Re/Im(H_c^{k=Q-1,l=0})), vec(Re/Im(H_d^{k=Q-1,l=1})), ...  ]
```

The corresponding transfer functions are provided in terms of their spline representation, i.e. a triple of spline knots, spline coefficients and order, for each given $H_c^{k,l}$. Transfer functions for the control terms are optional. If the transfer function interface is not defined, the identity will used instead (i.e. the controls will be applied 'raw': $p^k(t)\mathrm{Re}(H_c^{k,l}) + iq^k(t)\mathrm{Im}(H_c^{k,l})$).

python functions

```
def getHcTransfer_real/imag():
    [...]
    return [  [knots, coeffs, order]^{k=0,l=0}, [knots, coeffs, order]^{k=0,l=1} , ...,
                    ⋮
              [knots, coeffs, order]^{k=Q-1,l=0}, [knots, coeffs, order]^{k=Q-1,l=1} , ...  ]
```

When defining Hamiltonian matrices in the python script, the units should be radian (multiply them 2π). Again, note that all matrices will be passed to Quandary as lists of floats containing the column-wise vectorized matrix. Transfer functions are passed in terms of their spline representation (knots, coefficients and order). To generate spline representations within the python script, you can use for example SciPy's `interpolate` module. Quandary uses FitpackPP to recreate the spline representation for the transfer functions from provided knots, coefficients and order. Therefore, in order to use transfer functions, one needs to have fitpackpp installed and linked to Quandary (compare the Readme.md file).

After setting up a python script that provides the interface functions, the user has to set up the configuration file for Quandary, and provide the name of the python script with the config option `python_file`. Further, the number of levels per subsystem (defining the system dimension) should match the definitions in the python script. Make sure to choose an appropriate time-step size in the configuration file, in order to resolve the resulting dynamics. Some of the configuration options will be overwritten (the parameters that define (default) the system and control Hamiltonians, such as transition frequencies, self-kerr, cross-ker, coupling coefficients), since those are included in the python interface Hamiltonians. Configuration options concerning the runtype, control pulse

definitions, initial conditions, and optimization target and parameters, output etc. will still be valid and used to define the current simulation / optimization.

Similar to using the default Hamiltonian model, when using user-defined Hamiltonians one can still switch between the Schroedinger solver and the Lindblad solver by changing the configuration option `collapse_type` (`=none` for Schroedinger, any other choise for Lindblad solver). For the Lindblad solver, the same collapse terms as defined in (6) will be added to the dynamical equation.

Note: the Python interface can only be used in conjuction with the sparse-matrix solver (not the matrix-free solver).

3 The Optimal Control Problem

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

$$\min_{\boldsymbol{\alpha}} J(\{\rho_i^{target}, \rho_i(T)\}) + \gamma_1 \int_0^T P\left(\{\rho_i(t)\}\right) \, \mathrm{d}t + \gamma_2 \|\boldsymbol{\alpha}\|_2^2 \tag{24}$$

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, $i = 1, \dots, n_{init}$. The first term in (24) 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 serves as a penalty that can be added with $\gamma_1 \geq 0$ in oder to achieve a desired behavior of the quantum system over the entire time-domain $0 \leq t \leq T$, see Section 3.5. The third term is a Tikhonov regularization that can be added with parameter $\gamma_2 \geq 0$ in order to regularize the optimization problem (stabilize optimization convergence) by favoring solutions with small norm.

3.1 Fidelity

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

$$F = \begin{cases} \frac{1}{n_{init}} \sum_{i=1}^{n_{init}} \mathrm{Tr}\left(\left(\rho_i^{target}\right)^\dagger \rho_i(T) \right) & \text{if Lindblad} \\ \left| \frac{1}{n_{init}} \sum_{i=1}^{n_{init}} (\psi_i^{target})^\dagger \psi_i(T) \right|^2 & \text{if Schroedinger} \end{cases} \tag{25}$$

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`):

$$J_{frobenius} = \sum_{i=1}^{n_{init}} \frac{\beta_i}{2} \left\| \rho_i^{target} - \rho_i(T) \right\|_F^2 \tag{26}$$

$$J_{trace} = \begin{cases} 1 - \sum_{i=1}^{n_{init}} \frac{\beta_i}{w_i} \mathrm{Tr}\left((\rho_i^{target})^\dagger \rho_i(T) \right) & \text{if Lindblad} \\ 1 - \left| \sum_{i=1}^{n_{init}} \beta_i (\psi_i^{target})^\dagger \psi_i(T) \right|^2 & \text{if Schroedinger} \end{cases} \tag{27}$$

$$J_{measure} = \sum_{i=1}^{n_{init}} \beta_i \mathrm{Tr}\left(N_m \rho(T) \right) \tag{28}$$

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. (25). 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. 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. The type of the optimization target can be specified in the configuration option `optim_target`.

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

$$\psi^{target} = \boldsymbol{e}_m \quad \text{or} \quad \rho^{target} = \boldsymbol{e}_m \boldsymbol{e}_m^\dagger, \quad \text{for} \quad m \in \mathbb{N}_0 \quad \text{with} \quad 0 \le m < N \tag{29}$$

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

$$\psi^{target} = |m_0 m_1 m_2 \dots m_{Q-1}\rangle \quad \text{aka} \quad \psi^{target} = \boldsymbol{e}_{m_0} \otimes \boldsymbol{e}_{m_1} \otimes \cdots \otimes \boldsymbol{e}_{m_{Q-1}} \tag{30}$$

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

$$m = m_0 \frac{N}{n_0} + m_1 \frac{N}{n_0 n_1} + m_2 \frac{N}{n_0 n_1 n_2} + \cdots + m_{Q-1} \tag{31}$$

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 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 [3], that if one chooses the objective function $J_{measure}$ with corresponding measurement operator N_m (see eq. (28)), 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 [3] for details, and Section 3.4.4.

3.3.2 Read a target state from file

A specific target state ρ^{target} can also be read from file. This can be useful when considering non-pure target states. 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).

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

[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 [3] for a more detailed description).

initial state $\psi(0)$ to the unitary transformation $\psi(T) = V\psi(0)$. Target gates that are currently implemented are

$$V_X := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad V_Y := \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad V_Z := \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad V_{Hadamard} := \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{32}$$

$$V_{CNOT} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad V_{SWAP} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{33}$$

Further, a generalization of the SWAP and the CNOT gate for general Q subsystems is available: 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 easily added to the code by extending the `Gate` class in the corresponding `.cpp` and `.hpp` files. Lastly, 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). Compare the template configuration file for more details.

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 $\sum_i \beta_i = 1$.

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 $V_{SWAP} \in \mathbb{C}^{4\times4}$, 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 $V_{SWAP} \in \mathbb{C}^{4\times4}$, 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. (24) 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

$$\psi(0) = |m_1\rangle \otimes |m_2\rangle \quad \text{for } m_1 \in \{0,\dots,n_1-1\}, m_2 \in \{0,\dots,n_2-1\} \tag{34}$$

$$\text{and} \quad \rho(0) = \psi(0)\psi(0)^\dagger \tag{35}$$

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 [3] are implemented:

$$B^{kj} := \frac{1}{2}\left(\boldsymbol{e}_k\boldsymbol{e}_k^\dagger + \boldsymbol{e}_j\boldsymbol{e}_j^\dagger\right) + \begin{cases} 0 & \text{if } k = j \\ \frac{1}{2}\left(\boldsymbol{e}_k\boldsymbol{e}_j^\dagger + \boldsymbol{e}_j\boldsymbol{e}_k^\dagger\right) & \text{if } k < j \\ \frac{i}{2}\left(\boldsymbol{e}_j\boldsymbol{e}_k^\dagger - \boldsymbol{e}_k\boldsymbol{e}_j^\dagger\right) & \text{if } k > j \end{cases} \tag{36}$$

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 (36) 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 $J_{measure}$ (28), one can use the ensemble state

$$\rho_s(0) = \frac{1}{N^2} \sum_{i,j=0}^{N-1} B^{kj} \tag{37}$$

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 [3]). 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 $< k_0 >, \dots, < k_m >$ 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

$$\rho(0) = \boldsymbol{e}_0 \boldsymbol{e}_0^\dagger \otimes \underbrace{\rho_s(0)}_{\in \mathbb{C}^{N_s \times N_s}} \otimes \boldsymbol{e}_0 \boldsymbol{e}_0^\dagger \tag{38}$$

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

$$\rho(0)_1 = \sum_{i=0}^{N-1} \frac{2(N-i+1)}{N(N+1)} \boldsymbol{e}_i \boldsymbol{e}_i^\dagger \tag{39}$$

$$\rho(0)_2 = \sum_{ij=0}^{N-1} \frac{1}{N} \boldsymbol{e}_i \boldsymbol{e}_j^\dagger \tag{40}$$

$$\rho(0)_3 = \frac{1}{N} I_N \tag{41}$$

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 Integral penalty term and leakage

In order to achieve a desired behavior at earlier times other than the final time T, an integral penalty term can be added to the objective with parameter $\gamma_1 \geq 0$:

$$\gamma_1 \int_0^T P(\rho(t))\mathrm{d}t \tag{42}$$

In the standard setting, the following penalty term is currently implemented (can be extended in the `TimeStepper` class.):

$$P(\rho(t)) = w(t)J\left(\rho(t)\right) \quad \text{where} \quad w(t) = \frac{1}{a}e^{-\left(\frac{t-T}{a}\right)^2}, \tag{43}$$

for a penalty parameter $0 \leq a \leq 1$. Note, that as $a \to 0$, the weighting function $w(t)$ converges to the Dirac delta distribution with peak at final time T, hence reducing a leads to more emphasis on the final time T while larger a penalize non-zero energy states at earlier times $t \leq T$. Both γ_1 as well as a can be tuned in the configuration options, and choosing $\gamma_1 = 0$ and/or $a = 0$ will skip the extra computation of this term.

Be aware that adding the above integral penalty term requires to evaluate the objective J at *every* time-step within the time-domain $[0, T]$, instead of only the last one. This can be quite expensive, in particular when considering gate optimization, and should hence be turned off if possible (choosing $\gamma_1 = 0$ and/or $a = 0$ does that).

Leakage prevention: In addition to the above penalty, another intergral term can be added to the objective function in order to prevent leakage into higher levels that are not modelled by the finite-dimensional approximation to the infinite dimensional Hamiltonian. The leakage preventing term is enabled whenever the the number of essential levels is smaller than the number of modelled levels for at least one oscillator ($n_k^e < n_k$ for at least one subsystem k, compare Section 3.3.4), and in that case, the occupation of all *guard levels* (i.e. the last modelled energy levels of each of the subsystems) is penalized by adding

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

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

Note that if you want to include this leakage-preventing term, but do *not* want to include the above penalty term (43) (e.g. for efficiency reasons), you can choose $\gamma_1 > 0$ while $a = 0$, which will skip the computation of (43) but will include the term (44).

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 $q(t) := \mathrm{vec}(\rho(t)) \in \mathbb{C}^{N^2}$ (columnwise vectorization). Using the relations

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

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

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 $q^n \approx q(t_n)$. It is a second-order accurate, symplectic 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, we first solve a linear equation to get the stage variable k_1, then use it to update q^{n+1}.

While the implicit midpoint rule is the default (and preferred) setting in Quandary, there is the possibility to apply any time-stepping integrators provided by PETSc (class `TS` within PETSc). However, implementation needs to be verified, and adjoint time-stepping with Petsc is currently not available.

4.3.1 Choice of the time-step size

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 $Y^\dagger H_d Y = \Lambda$ where $Y^\dagger = Y$ 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 a first order scheme we would use something like $p = 20$, second order may use $p = 10$. 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 himself 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 to varify if the time-step is small enough (testing for second order convergence of the error when reducing Δt).

If one wants to include the time-varying Hamiltonian part $H = H_d + H_c(t)$ in the analysis, one could use the constraints on the control parameter amplitudes to remove the time-dependency using their large value instead.

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

$$\bar{q}^n = \bar{q}^{n+1} + \delta t \left(M^{n+1/2}\right)^T \bar{k}_1 \quad \text{where } \bar{k}_1 \text{ solves} \quad \left(I - \frac{\delta t}{2} M^{n+1/2}\right)^T \bar{k}_1 = \bar{q}^{n+1} \tag{52}$$

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

$$\nabla J+ = \delta t \left(\frac{\partial M^{n+1/2}}{\partial z}\left(q^n + \frac{\delta t}{2} k_1\right)\right)^T \bar{k}_1 \tag{53}$$

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 (53) 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 three 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.

3. Parallel optimization schemes. Currently under development.

Strong and weak scaling studies are presented in [3].

In the main code, the global communicator (MPI_COMM_WORLD) is split into three sub-communicator, one for each of the above. The total number of executing MPI processes (np_{total}) is split into three subgroups in such a way that

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

The user specifies the size of the communicator for distributing the initial conditions (np_{init}) as well as parallelization for optimization (np_{optim}) in the configuration file. The number of cores for parallel linear algebra (np_{petsc}) is then computed from the above equation. The following requirements for parallel distribution must be considered when setting up parallel runs:

- $\frac{n_{init}}{np_{init}} \in \mathbb{N}$, where n_{init} is the number of initial conditions that are considered. This requirement ensures that each processor group owns the same number of initial conditions.
- $\frac{np_{total}}{np_{init} * np_{optim}} \in \mathbb{N}$, so that each processor group has the same number of cores for Petsc.
- $\frac{M}{np_{petsc}} \in \mathbb{N}$, for $M = N^2$ in the Lindblad case and $M = N$ in the Schroedinger case. Hence the system dimension must be an integer multiple of the number of cores used for distributed linear algebra from Petsc. This constraint 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 $m = 1, \dots, n_{init}$ 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

$$\langle N^{(n_k)} \rangle = \mathrm{Tr}\left(N^{(n_k)} \rho^k(t)\right) \tag{54}$$

  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 $N \times N$.

- `population`: This option prints the time evolution of the population of subsystem k into files named `population<k>.iinit<m>.dat` for each initial condition $m = 1, \dots, n_{init}$. 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 $l = 0, \dots, n_k - 1$ 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 $(\rho^k_{ll}(t), l = 0, \dots n_k - 1)$, for Schroedinger's solver it's the absolute values of the reduced state vector elements $|\psi^k_l(t)|^2, l = 0, \dots n_k - 1$. Note that the reduction to the subsystem k induces a sum over all oscillators to collect contributions to the reduced state.

- `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 $f^k(t)$.
- `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:

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

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, 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.

[3] 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.

[4] 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

$$iH = iH_d(t) + iH_c(t) \tag{55}$$

$$= i\left(\sum_k(\omega_k - \omega_k^{\text{rot}})a_k^\dagger a_k - \frac{\xi}{2}a_k^\dagger a_k^\dagger a_k a_k - \sum_{l>k}\xi_{kl}a_k^\dagger a_k a_l^\dagger a_l + \sum_{l>k}J_{kl}\cos(\eta_{kl}t)\left(a_k^\dagger a_l + a_k a_l^\dagger\right)\right. \tag{56}$$

$$\left. + \sum_k p^k(\vec{\alpha}^k, t)\left(a_k + a_k^\dagger\right)\right) \tag{57}$$

$$+ \left(\sum_k\sum_{kl} -J_{kl}\sin(\eta_{kl}t)\left(a_k^\dagger a_l - a_k a_l^\dagger\right) - \sum_k q^k(\vec{\alpha}^k, t)\left(a_k - a_k^\dagger\right)\right) \tag{58}$$

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

$$A(t) = A_d + \sum_k q^k(\vec{\alpha}^k, t) A_c^k + \sum_{l>k} J_{kl} \sin(\eta_{kl} t) A_d^{kl} \tag{59}$$

$$\text{with} \quad A_d := \sum_k \sum_{j=1,2} \gamma_{jk} \left(\mathcal{L}_{jk} \otimes \mathcal{L}_{jk} - \frac{1}{2} \left(I_N \otimes \mathcal{L}_{jk}^T \mathcal{L}_{jk} + \mathcal{L}_{jk}^T \mathcal{L}_{jk} \otimes I_N \right) \right) \tag{60}$$

$$A_c^k := I_N \otimes \left(a_k - a_k^\dagger \right) - \left(a_k - a_k^\dagger \right)^T \otimes I_N \tag{61}$$

$$A_d^{kl} := I_N \otimes \left(a_k^\dagger a_l - a_k a_l^\dagger \right) - \left(a_k^\dagger a_l - a_k a_l^\dagger \right)^T \otimes I_N \tag{62}$$

and

$$B(t) = B_d + \sum_k p^k(\vec{\alpha}^k, t) B_c^k + \sum_{kl} J_{kl} \cos(\eta_{kl} t) B_d^{kl} \tag{63}$$

$$\text{with} \quad B_d := \sum_k (\omega_k - \omega_k^{\text{rot}}) \left(-I_N \otimes a_k^\dagger a_k + (a_k^\dagger a_k)^T \otimes I_N \right) - \frac{\xi_k}{2} \left(-I_N \otimes a_k^\dagger a_k^\dagger a_k a_k + (a_k^\dagger a_k^\dagger a_k a_k)^T \otimes I_N \right) \tag{64}$$

$$- \sum_{l>k} \xi_{kl} \left(-I_N \otimes a_k^\dagger a_k a_l^\dagger a_l + (a_k^\dagger a_k a_l^\dagger a_l)^T \otimes I_N \right) \tag{65}$$

$$B_c^k := -I_N \otimes \left(a_k + a_k^\dagger \right) + \left(a_k + a_k^\dagger \right)^T \otimes I_N \tag{66}$$

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

$$\tag{68}$$

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

$$A(t) = Re(M(t)) = A_d + \sum_k q^k(\alpha^k, t) A_c^k + \sum_{l>k} J_{kl} \sin(\eta_{kl} t) A_d^{kl} \tag{69}$$

$$B(t) = Im(M(t)) = B_d + \sum_k p^k(\alpha^k, t) B_c^k + \sum_{kl} J_{kl} \cos(\eta_{kl} t) B_d^{kl} \tag{70}$$

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.