OBE Solver Documentation

September 30, 2019

This Optical Bloch Equations solver is in fact a small package allowing to quickly design the quantum system and obtain the time evolution of the density matrix. It is done by numerically solving the master equation in the Lindblad form. The package consists of three classes: ***Hamiltonian***, where the quantum system's energies and couplings are encoded, ***Dissipator***, where the dissipative part of the equation is defined, and ***BlochEqns***, where the equations are created and solved.

class *Hamiltonian*(n)

This class defines system's hamiltonian using symbolic variables that's designed for a quantum system with n states. If numerical values are necessary, one can always use function ***subs(****obj.***,symbolic variables,numerical values****)*** and store the result in a new user-defined variable. To find symbolic variables that are being used, use ***symvar(****obj.******)***.

Properties

Public

The class stores the following public properties:

- `hamiltonian` - contains the quantum system's basic hamiltonian that's initially defined - an n-by-n matrix. If the hamiltonian is transformed using a unitary transformation, it will be stored in a different variable. This one always contains the original hamiltonian.
- `transformed` - contains the hamiltonian after a unitary transformation is applied
- `transMatrix` - the matrix used for a unitary transformation
- `energies` - diagonal elements of the hamiltonian (energies of all the states) stored as a vector of length n
- `couplings` - a n-by-n-by-2 matrix, where the $(i, j, 2)$ element is the Rabi rate associated with the $i \to j$ transition, and the $(i, j, 1)$ element is frequency of the light field
- `detunings` - an m-by-3 matrix, where m is number of defined detunings. $(k, 1)$ is the index of the initial state of the transition, $(k, 2)$ is the index of the final state and $(k, 3)$ is the detuning d
- `zeroEnergy` - energy level (symbolic) defined as having 0 energy. It has to be a part of one of the energy levels specified in property `energies`
- `stateGraph` - a directed graph of the system, where edges go along defined couplings (if coupling is defined as $i \to f$, the edge will follow that direction) and along the decay paths
- `brightStates` - matrix storing system's bright states as rows
- `darkStates` - matrix storing system's dark states as rows, if any exist

Private

Private properties are:

- `freqs` - vector storing light frequencies
- `cpl` - number of state couplings in the system
- `cplGr` - coupling graph between states (can be disconnected, depends on how the system is set), where edges have weights equal to coupling frequencies.

Methods

Public

In the class we have following public methods:

- ***addEnergies(****vec****)*** - this function takes vector *vec* (a list of symbolic variables) as its input and places them in the hamiltonian. It also stores the vector in the property `energies`. These represent energies of quantum states.
- ***addCoupling(****S_i, S_f, Ω, ω****)*** - couples initial state S_i with final state S_f with a light field that has Rabi rate Ω and frequency ω. It places appropriate term in the off-diagonal cells of the hamiltonian assuming rotating wave approximation. Variables S_i and S_f are indices so they have to smaller than the number of states ($0 < S_i$,$S_f < n$), while Ω and ω have to be symbolic. The terms that are added are:

  $$H(S_i, S_f) = H(S_f, S_i)^* = -\Omega\frac{e^{i\omega t}}{2}$$

  The variables used as argument of this function are then stored in the class variable `couplings`.
- ***addPolyCoupling(****S_i, S_f, Ω, ω****)*** - function very similar to ***addCoupling()***, with the exception that it adds a new term in $H(S_i, S_f)$, instead of setting it. In other words, the function acts in the following way:

  $$H(S_i, S_f) \to H(S_i, S_f) - \Omega\frac{e^{i\omega t}}{2}$$
  $$H(S_f, S_i) \to H(S_f, S_i) - \Omega^*\frac{e^{-i\omega t}}{2}.$$

- ***defineStateDetuning(****S_i, S_f, δ****)*** - changes variables, by replacing light's frequency ω coupling states S_i and S_f with detuning δ. Variables S_i and S_f are numerical indices, while δ has to be symbolic. The substitution is defined as: $\delta = E_f - E_i - \omega$, where E_f and E_i are energies of final and initial states respectively.
- ***defineEnergyDetuning(****E_i, E_f, δ, ω****)*** - this function changes variables in a more straight-forward way, which might be needed, when state energies are more complicated variables. It replaces frequency ω with energy difference minus the detuning ($w = E_f - E_i - \delta$). All variables have to be symbolic.
- ***createGraph(****L****)*** - uses an object of the `Dissipator` class to create a directed graph. Vertices are states labeled using their respective energies, while edges are either couplings between states or decay paths labeled by either Rabi rates or decay rates.
- ***plotGraph()*** - plots graph of the system that's kept in the `stateGraph` property
- ***addSidebands(****center, order, depth, spacing****)*** - adds *order* number of sidebands around *center* frequency to all the couplings defined with such frequency. They are added using expansion to Bessel functions (so it is crucial that the *order* is chosen correctly depending on *depth*; for exact solution, use ***addPhaseModulation()*** function). The modulation depth and spacing between sidebands are define by *depth* and *spacing* respectively. This function should be used only after the unitary transformation of the hamiltonian (so after ***unitaryTransformation()*** function is called).

- ***addPhaseModulation(*** ω, β, ω_m***)*** - adds phase modulation to the laser of ω frequency, depth β and modulation frequency ω_m to all the couplings defined with provided frequency. The modulating parts of the off-diagonal terms in the hamiltonian are changed in the following way:

$$e^{i\omega t} \to e^{i\omega t + i\beta \sin \omega_m t}.$$

- ***changeBasis(****U****)*** - performs basis change. The function acts on the property `transformed`:

$$H \to U^\dagger H U,$$

where H is hamiltonian in form stored in property `transformed`. The matrix U has to be provided as a list of n vectors, $U = [u_1, u_2, \ldots, u_n]$, where n is the size of the hamiltonian.

- ***findDarkStates(****GS, ES****)*** - this function finds the dark and bright by looking at the kernel of normalized of the interaction part of the hamiltonian between defined ground and excited states. Assuming that H is the hamiltonian in the form stored in the property `transformed`, Q is projection operator onto the excited states ES and P is projection operator onto the ground states GS:

$$Q = \sum_{i=ES} |i\rangle \langle i|$$
$$P = \sum_{j=GS} |j\rangle \langle j|,$$

then the bright states $B = QHP$, while the dark states $D = \mathbf{ker}\ B$. These are stored in properties `brightStates` and `darkStates` respectively. GS and ES are lists of indices.

- H_{eff}=***adiabaticElimination(****GS, ES****)*** - performs adiabatic elimination of the excited states. Physically, it means that changes occurring to quantum states provided in variable ES are happening on drastically different timescales than for the rest of the system, and therefore these states can be effectively eliminated. The mathematical procedure and physical conditions when one can use it are provided in [1]. The function returns hamiltonian after the adiabatic elimination.

- ***unitaryTransformation(******kwargs****)*** - The function finds the appropriate frame, where there would be no time dependence in the hamiltonian, by using one of two methods. Both of these methods try to find unitary transformation matrix T after which following transformation is performed on the hamiltonian H stored in property `hamiltonian`:

$$H \to T^\dagger H T - i T^\dagger \frac{dT}{dt}.$$

The resulting hamiltonian is stored in property `transformed` and transformation matrix in `transMatrix`. Options:

  - 'Method':
    - 'general' - the default option. It looks at coupling through light fields between states and the state that is defined as having zero energy. The sum of frequencies in these couplings is put into the unitary matrix and used to transform the Hamiltonian. Time dependence is usually eliminated (it is not always possible; it depends on the system) and substitutions from defined detunings are made.
    - 'equations' - the unitary transformation is found by solving a system of linear equation. This method is usable if there are at most as many couplings as there are states. It does not require a defined zero energy.

class *Dissipator*(n)

This class defines system's dissipator operator using symbolic variables that's designed for a quantum system with n states. If numerical values are necessary, one can always use function ***subs(****obj.***,symbolic variables,numerical values)* and store the result in a new user-defined variable. To find symbolic variables that are being used, use ***symvar(****obj.***)*.

Properties

Public

Public properties are:

- `densityMatrix` - the density matrix for which we are solving Bloch equations. It is also needed to create a dissipation matrix.
- `dissipator` - the dissipation matrix. Diagonal elements correspond to population transfers due to spontaneous decays. Off-diagonal elements are the decoherence terms created by the spontaneous decay only. However, additional decoherence terms can be added by hand.
- `branching` - a matrix B of branching ratios. The initial states are rows, final states are columns, so the branching ratio of the $i \to f$ decay is $B(i,f) = b_{if}$ (if state $|i\rangle$ decays with rate Γ, the decay to f is with rate $b_{if}\Gamma$)
- `decayR` - a vector storing total decay rates from all the states.

Methods

The class has following public methods:

Public

- ***addDecay(****S_i, S_f, $\Gamma_{i\to f}$)* - this function adds a decay from initial state $|i\rangle$ to final state $|f\rangle$. S_i and S_f are states' indices (numerical values). The variable $\Gamma_{i\to f}$ is the decay rate from $|i\rangle$ to $|f\rangle$ *only*, so if state $|i\rangle$ decays to several states, and its total decay rate is Γ, and the branching ratio to $|f\rangle$ is b_{if}, then $\Gamma_{i\to f} = b_{if}\Gamma$. It has to be a symbolic variable. Mathematically, a matrix associated with that decay is created first:

$$G_{i\to f} = \Gamma_{i\to f}\,|f\rangle\langle i| = \begin{array}{c} \\ f \\ \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{cccc} & & i & \end{array} \\ \begin{pmatrix} 0 & \cdots & \sqrt{\Gamma_{i\to f}} & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & & 0 \end{pmatrix} \end{array}.$$

This allows us to get a matrix that represents contribution to dissipator $\mathcal{L}$ from spontaneous decay from state i to f:

$$\mathcal{L}_{i\to f} = -\frac{1}{2}\{G^\dagger_{i\to f}G_{i\to f},\rho\} + G_{i\to f}\rho G^\dagger_{i\to f} = \begin{array}{c} \\ f \\ \\ \\ i \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} & & f & & i & & \end{array} \\ \begin{pmatrix} 0 & & & & -\frac{\Gamma_{i\to f}}{2}\rho_{1i} & & 0 \\ & \Gamma_{i\to f}\rho_{ii} & & & \vdots & & \\ & & & 0 & \vdots & & \\ -\frac{\Gamma_{i\to f}}{2}\rho_{i1} & \cdots & & \cdots & -\Gamma_{i\to f}\rho_{ii} & \cdots & -\frac{\Gamma_{i\to f}}{2}\rho_{in} \\ & & & & \vdots & & \\ 0 & & & & -\frac{\Gamma_{i\to f}}{2}\rho_{1n} & & 0 \end{pmatrix} \end{array}.$$

- ***fromBranching(****BR, DR)* - if branching ratios and total decay rates of all the states are already in a matrix, this function can be used to create the dissipator. BR is a n-by-n matrix representing branching ratios in the same form as the property `branching`. DR is a vector of all the decay rates, which has to be in the same form as property `decayR`. This function uses a simplified way of creating the dissipator. We can first create matrix $M = \sum_i M_i$, where:

$$(M_i)_{kl} = \begin{cases} \sqrt{\Gamma_i} & \text{if } k = l = i \\ 0 & \text{otherwise} \end{cases}$$

and i labels the initial state. Then, the off-diagonal decoherence terms are added via:

$$\mathcal{L}_\Gamma - \frac{1}{2}\{M^2, \rho\} + \sum_i M_i \rho M_i,$$

while diagonal population transfer terms are added one-by-one.

class *BlochEqns*$(H,\ L)$

This class builds a system of Bloch equations using Hamiltonian and the dissipation matrix. The constructor as its arguments uses object H of class ***Hamiltonian*** and object L of class ***Dissipator***. It uses the hamiltonian after the unitary transformation. It uses basic hamiltonian, if the transformation wasn't performed.

Properties

The stored public properties are:

Public

- `steadyState` - n-by-n matrix, where the (i, j) element corresponds to steady state solution for ρ_{ij} ((i, j) element of the density matrix). ρ_{ii} are populations.
- `densityMatrix` - density Matrix ρ copied from the ***Dissipator*** object.
- `evTime` - times at which the Bloch equations where evaluated when solving for the time evolution. It's a vector with elements equal to the number of steps in the solution.
- `evolution` - n-by-n-by-l matrix, where (i, j) element is a vector of length l storing values of ρ_{ij} at times saved to `evTime` property; i.e. it's the time evolution of all elements of the density matrix.
- `evolutionTr` - density matrix time evolution `evolution` after basis change.
- `eqnsRHS` - n^2-long vector storing the right-hand side of Bloch Equations in the vector form, i.e. $i[H, \rho] + \mathcal{L}$. The order is: $[(1, 1), (1, 2), ..., (1, N), (2, 1)..., (N, N - 1), (N, N)]$
- `equations` - n-by-n matrix storing full Bloch (master) equations, i.e. $\dot{\rho} == i[H, \rho] + \mathcal{L}$, in symbolic form.
- `equationsVector` - n^2-long vector storing full Bloch (master) equations
- `equationsS` - n-by-n matrix storing steady state equations, i.e. $0 == i[H, \rho] + \mathcal{L}$, in symbolic form.
- `intTime` - a 2-element vector storing the integration time for time evolution. .
- `initialConditions` - n-by-n matrix storing initial conditions used for the time evolution.
- `lastSol` - last solution obtained from numerical integration; used when extending the time evolution.
- `optParams` - optimal parameters found through optimization.
- `optVal` - optimal value found through optimization.

Methods

The public methods of this class are:

Public

- ***solveSteady(*** v ***)*** - solves Bloch equations symbolically in the steady state and saves the solution. Note, that the solution usually is feasible only for small systems.
- ***necessaryVariables()*** - finds all the symbolic variables in Bloch equations that need to be substituted before solving for the time evolution. It is always useful to use this function before attempting to solve the equations.
- ***evolve(*** t_i, t_f, IC, $vars$ ***)*** - evolves the Bloch equation using ode45 (explicit six stage fifth order Runge-Kutta method) solver. Evolution happens from t_i to t_f with initial conditions IC. $vars$ are numerical values of symbolic variables found using function ***necessaryVariables()***. They have to be provided in the same order!
- ***plotEvolution()*** - plots diagonal elements of `evolution` (populations of states) as function of time
- ***extendEvolution(*** t_f, $vars$ ***)*** - evolves Bloch equations using the previously obtained solution as the initial condition. Solution is concatenated to the already existing one.
- ***optimizeParameters(*** t_i, t_f, IC, M, S, **args* ***)*** - optimizes parameters using genetic optimization algorithm. Can be optimized to obtain high/low populations in various states. t_i and t_f are the integration time, IC - initial conditions, M - cell matrix with parameters and their domains, e.g. $M = \{G, 0, 1; w_a, -2, 2\}$, S - list of states with regards to which we're optimizing (their indices).
  There are several options:
  - 'Criterion':
    - 'minimum' - optimizer minimizes chosen parameter.
    - 'maximum' - optimizer maximizes chosen parameter.
  - 'Popsize' - an integer greater than 0. It's the number of random set of parameters at every iteration.
  - 'Iterations' - an integer greater than 0. It's the number of iterations every population is optimized for.
  - 'Popnumber' - an integer greater than 0. It's the number of different populations initialized and separately optimized.
  - 'Fraction' - a float between 0 and 1. It's part of population that is carried over to next iteration
  - 'Integration':
    - 'no' - population in chosen state(s) at time t_f is used for optimization.
    - 'yes' - time integral of population in chosen state(s) from t_i to t_f is chosen as the parameter for optimization.
  - 'Switching':
    - 'no' - time evolution with constant parameters is performed from t_i to t_f.
    - 'yes' - every $t_s = (t_f - t_i)/n_s$ the set of parameters used for optimization is turned off/on, where n_s is the 'NoSwitches' parameter.
  - 'NoSwitches' - an integer greater than 0. It's the number of times parameters are switched in the integration time.
  - 'SwitchingParameters' - a matrix of parameters showing when to switch on/off optimizable parameters. It should have at most n_s number of rows and the number of columns equal to the number of optimized parameters. In a specific row 0 indicates parameter being turned off, any other number indicates parameter turned on.

  Default settings - 'Criterion': 'maximum'; 'Popsize': 20; 'Iterations': 10; 'Popnumber': 3; 'Fraction': 0.3; 'Integration': 'no'; 'Switching': 'no'; 'NoSwitches': 0; 'SwitchingParameters': [1,1,...,1] (length equal to the number of optimized parameters)

- ***changeBasis(U)*** - performs basis change. The function acts on the property `evolution`:

$$\rho(t) \to U^\dagger \rho(t) U,$$

where ρ is the density matrix time evolution in form stored in property `evolution`. The matrix U has to be provided as a list of n vectors, $U = [u_1, u_2, \ldots, u_n]$, where n is the size of the density matrix The resulting density matrix is stored in property `evolutionTr`.

References

[1] Brion, E., Pedersen, L. H., Molmer, K. (2007). Adiabatic elimination in a lambda system. Journal of Physics A: Mathematical and Theoretical, 40(5), 1033-1043