Project description

Daniel Puzzuoli

December 5, 2018

1 Purpose

This coding project is an implementation of algorithms for searching for control sequences for quantum systems. In particular, the goal is to search for control sequences that optimize both the final gate, as well as driving terms of the form

$$U(t) \int_0^t dt_1 U^*(t_1) H U(t_1) \tag{1}$$

to desired values, which can represent various robustness criteria. I won't yet describe here what this means, as we're working on a paper (Holger Haas, myself, and David Cory) that does exactly this and I can point to it later when writing a proper description.

The code is currently in a prototype phase: It can now successfully find control sequences implementing a particular gate and satisfying various robustness criteria, though there is a lot to be done before it can potentially find useful application.

Working examples that demonstrate the aim of the project can be found in the `iPython` notebooks in the `examples` folder.

The rest of this document contains a (very) rough description of the code design and elements, as well as a to do list and set of notes for where to take the project in the future.

2 Design

(Note: as the code is currently being restructured on a regular basis, the names used in this description may not be correct.)

This project is meant to eventually handle a wide variety of control problems. Each control problem essentially requires a specification of various linear systems all built on the original quantum system, and the number and form of these linear systems will potentially vary heavily across each problem. As such, the code is designed around the idea that a user should be able to easily "construct" their own control problem out of the elements provided. I still haven't figured out exactly how I want the front end to look, and that seems like a problem for the future if the backend is ever actually useful, but it is something motivating the design.

The functional code is broken into the following parts:

- Specification/construction of control systems and auxiliary systems
- System evolution/propagation
- Objective and penalty functions

2.1 Specification/construction of control and auxiliary systems

This currently consists of the `control_system` object and the file `utb_matrices`.

- `ubt_matrices` contains various functions for the construction of upper triangular block matrices, which are used for computing derivatives of propagators, and more importantly for computing integrals of the form given in Equation (1). Nothing complicated is going on here; the point is to have very convenient functions that make constructing these auxiliary systems easy.
- `control_system` objects contain a specification of a linear system, which we call a *control system* as it has a *drift* term, as well as *control* terms. Currently these objects only store data and have a few small functions. As is described in the to do section, a potential solution to handling computational redundancies in propagation of these control systems is to have `control_system` objects be able to depend on other control system objects in a way that automates the process of sharing common computations.

2.2 System evolution

This consists entirely of the `evolve_system` file, which contains functions for calculating the final propagator of a system, as well as its derivatives. One of the first next things I want to do is add Hessian computation. There is not much to say, as this function operates in the usual way that GRAPE is done.

2.3 Objectives and penalties

- `objective_functions` contains typical objective functions for these settings, which currently compute the objective, as well as the first derivative. Currently, the objective functions are of the following form:

$$f(U) = \|\Phi(U)\|_2^2 = \langle \Phi(U), \Phi(U) \rangle = \mathrm{Tr}(\Phi(U)^* \Phi(U)), \tag{2}$$

  where $\Phi : M_n \to M_m$ is a linear map. We call functions of this form *generalized GRAPE objectives*, as they generalize the GRAPE objective $f(U) = |\langle V, U \rangle|^2$ and can be analyzed in exactly the same fashion (the GRAPE objective corresponds to the choice $\Phi(U) = \langle V, U \rangle$. This class of functions is chosen as it seems to encapsulate all of the objectives I've seen. For example, in the decoupling computations, we want to find a sequence that sets a certain block in the control system propagator to 0, and this is represented by choosing Φ to extract the block (which is a linear map).
- `constraints` contains functions that represent penalties on the control amplitudes, to be used in the optimizations overall objective function to enforce constraints on the control sequence. What's needed for this is a function that is 0 when the constraints are satisfied, but that rapidly becomes very large as the constraints are violated. For this, given an upper bound ub, a lower bound lb, an $\epsilon > 0$ and a power n, I chose the function:

$$f(x) = \begin{cases} 0 & x \in [lb + \epsilon, ub - \epsilon] \\ (\frac{x - (ub - \epsilon)}{\epsilon})^n & x > ub - \epsilon \\ (\frac{(lb + \epsilon) - x}{\epsilon})^n & x < lb - \epsilon \end{cases} \tag{3}$$

The parameter $\epsilon > 0$ is some distance from the true boundary that the constraint should activate.

Within this file, penalty functions on control amplitude, as well as control rate of change, have been implemented based on the above $f(x)$.

3 To do and notes

3.1 To do now

- Hessian computation
- Once Hessian computation is sorted, use a search algorithm that takes second derivatives into account
  - scipy has a few, including `trust-ncg` which is a newton conjugate gradient method using something called a trust region. This optimization algorithm seems to be second best according to [**?**], so assuming scipy's is actually this algorithm it is a good thing to start using once second derivatives are handled.
  - In the long run, the Newton (RFO) algorithm in [**?**] seems to clearly outperform all other algorithms, so it will make sense to implement this

3.2 To do soon

- Look into matrix multiplication algorithms
  - Whether computing derivatives of propagators, or computing decoupling terms, the majority of the matrix multiplications done are to take powers of block matrices where all diagonal blocks are the same, and the only other non-zero blocks are on the first off-diagonal (and these can potentially all be different). Any power of such a matrix will be utb with all diagonal blocks the same. Hence, computing a successive power only requires:
    - 1 matrix multiplication to compute all diagonals
    - For every non-diagonal block, 2 matrix multiplications

    Hence, for block dimension k, computing the next power only needs $1 + 2\frac{k(k-1)}{2} = k^2 - k + 1$ matrix multiplications. Multiplying a general $k \times k$ block matrix will scale as k^ω, where $\omega \geq 2.37$, so taking advantage of this structure can in principle significantly reduce the number of operations.
  - Should look into sparse matrices as well, as the generators will usually be sparse.

3.3 To do later

- A complete overhaul (or, just initially properly design) `control_system`
  - There are many natural interdependencies in these control problems
    - Decoupling systems are obviously built on the base system, so construction of the generators can be redundant

    - Sometimes the exponentiations required for a decoupling system are the same as those for derivatives of the base system (e.g. if you want to decouple one of the control generators themselves)
    - Also, at some point we want to be able to handle objectives on propagators at intermediate time steps
  - A way of addressing all of these issues is to have a more involved `control_system` object, in which there can be a hierarchy of control systems that depend on each-other **(note: I say hierarchy below, but at the moment I have no specific ideas for exactly how this will be handled)**
    - E.g. even just for a basic final gate finding problem, there is the base system, but we also need to do exponentials for a derived system for computing derivatives, so it potentially makes sense to automatically create derivatives systems, which the base system itself actually depends on
    - For a given control amplitude list, each system in the hierarchy can return the generators, single step propagators, forward backward, and final propagators. The creation of this data should be lazy, and it should always first check if it can simply be extracted in some way from another system in the hierarchy. This will enable a way to handle order of computations without having to explicitly build it into `evolve_system` (or whatever simulation setup is being used). In fact, this is nice as it will completely separate the simulation functions from data storing (i.e. it won't need to create arrays for derivatives, it will simply ask for them from the control system and combine them)

- Transfer functions