Notes

June 1, 2019

1 Design as of May 28 2019

After learning about symbolic stuff, here is roughly how I imagine the code. When I say 'symbol' in what follows, I am thinking of actually Symbol objects in SymPy. At this point I'm only thinking at the level of differential equations, i.e. I'm completely assuming that objectives and transfer functions can be built on top of this and don't need to fundamentally interact with the processes I'm outlining.

(Note: Need to add derivatives into the bottom flow. I.e. To compute derivatives at each time step, we will need to construct even larger generators, when we exponentiate it, we will get the exponentiation of the base generator, but also of derivatives of these with respect to each control amplitude. Where we keep track of this needs to be figured out.)

- User specification of symbols/matrices involved in problem
  - Concrete symbol specification: Associations between symbols and concrete matrices, e.g. the symbol 'X' is associated with the Pauli matrix. Could be implemented as a dictionary. Note, this could also include concrete scalars, e.g. constants (not totally sure how this would come up but may be useful to include).
  - Reserved symbols for control amplitudes, e.g. $(a[0], a[1], \dots)$.[1]
  - Symbolic symbol specification: Symbolic specification of new symbols that are related to concrete symbols. The one specific thing I have in mind is how the generator relates to control amplitudes, e.g.

$$G = D + a[0]X + \cdots + a[k]Z. \tag{1}$$

- User specification of DE terms of interest (e.g. final propagator, various Dyson terms, stochastic terms...)
  - This is the one bit of the interface that will take a lot of thought. At the most general, I'd like to have a function that enables specification of any integral that we know can be computed using these methods. Then, we can write more custom functions to specify, e.g. Dyson terms, Magnus terms, terms with exponentials, etc... .

[1] Here we are only thinking of the differential equations, so these are the control amplitudes as they appear in the DE, after being fed through a transfer function.

  - In the meantime though, we can write special functions for specifying Dyson terms, and expand it as things come up. Also, even before this, we can skip this step and manually perform the next bullet.

- Symbolic creation of generators from base DE specification and AHT specification
  - As with the previous point, this is something that could potentially be developed a lot in the long run. Given an integral from the previous point, returns a symbolic specification for the new DE, i.e. symbolic rep of the drift and terms depending on control amplitudes.
  - In the short term, we can just do Dyson terms and put in terms for exponentials by hand. In the longer run, for an arbitrary integral computable with these methods (along with certain scalar functions), it could actually find the generator via the procedure outlined in the paper.
  - The main point is that we need to generate a block structure, along with:
    * Rules for how the blocks relate to the user-specified symbols
    * Some description at the control level; decomposition of generators into the drift and the parts that depend on the control amplitudes.
    * Rules for where to find specified AHT terms in the exponentiated generators.

- Symbolically find algebra, generate custom symbolic rules for algebra members (multiplication, norms, linear system solving)
  - Given the symbolic structure of generators, find a containing algebra (I think the procedure I have now finds the smallest one)
  - Return:
    * Algebra description (list of unique symbols, along with locations they appear)
    * Symbolic multiplication rule
    * Symbolic norm rule (if we're using, e.g., column norms, this will be some specification of what columns appear)
    * System solving rule (not 100% sure how this will look. This I think will be completely contained in the multiplication rule so is probably redundant.)

- Generate code to carry out custom rules
  - Create 3 custom functions for use in matrix exp: mult(A,B), norm(A), systemsolve(A,B) (may need more), with A,B being some lists or something that contain all of the unique blocks in the algebra. (By system solve I mean solving the linear system $AC = B$, which is used in the inversion step of matrix exp.)
  - One uncertainty I have is exactly where we will store these functions. Should we have a folder that the user can specify as where to save all of the custom stuff as well as store optimization results?
  - This will obviously take as input a string or something specifying the implementation we want to use (e.g. numpy, tensorflow, ...)

- Feed into a DE solver, also giving derivatives (whether GRAPE, Runge-Kutta, ...)

2 Some rough resource analysis

Consider the example in the Jupyter notebook of computing the first order Dyson terms for X,Y, and Z for a qubit, along with their derivatives with respect to X and Y. The generators for computing these things via matrix exponentiation are 6 4×4 block matrices, with each block having dimension 2. For now lets actually take the dimension of the blocks to be n (i.e. perhaps this is a higher dimensional system with the above type control problem, e.g. a higher dimensional spin, or a truncated an-harmonic oscillator).

In principle, without taking advantage of any structure (beyond that it is matrix multiplication) multiplying two systems of matrices of this type requires $6 \times (4n)^\omega$ operations. Taking $\omega \approx 3$, we have about $216n^3$ operations. If we take advantage of the upper-triangular structure (but nothing else), we have $6(4n^\omega + 6n^\omega + 6n^\omega + 4n^\omega) = 120n^\omega$ operations (ignoring additions). That is, there are 120 multiplications of $n \times n$ matrices required. Finally, for the reduced rules, we have $25n^\omega$, as there are only 25 matrix multiplications of $n \times n$ required (again, ignoring additions or scalar multiplications, which there will be fewer of). So, at least in principle (and roughly), we see the number of operations reducing by a factor of ≈ 4.8, and this is actually one of the simplest examples.