Algorithms for perturbative analysis and simulation of quantum dynamics

Daniel Puzzuoli\textsuperscript{1,*}, Sophia Fuhui Lin\textsuperscript{2}, Moein Malekakhlagh\textsuperscript{3}, Emily Pritchett\textsuperscript{3}, Benjamin Rosand\textsuperscript{3}, and Christopher J. Wood\textsuperscript{3}

\textsuperscript{1}IBM Quantum, IBM Canada, Markham, ON, L3R 9Z7, Canada
\textsuperscript{2}Department of Computer Science, University of Chicago, Chicago, IL, 60615, USA
\textsuperscript{3}IBM Quantum, IBM T.J. Watson Research Center, Yorktown Heights, NY, 10598, USA

\textsuperscript{*}daniel.puzzuoli1@ibm.com

June 7, 2023

Abstract

We develop general purpose algorithms for computing and utilizing both the Dyson series and Magnus expansion, with the goal of facilitating numerical perturbative studies of quantum dynamics. To enable broad applications to models with multiple parameters, we phrase our algorithms in terms of multivariable sensitivity analysis, for either the solution or the time-averaged generator of the evolution over a fixed time-interval. These tools simultaneously compute a collection of terms up to arbitrary order, and are general in the sense that the model can depend on the parameters in an arbitrary time-dependent way. We implement the algorithms in the open source software package Qiskit Dynamics, utilizing the JAX array library to enable just-in-time compilation, automatic differentiation, and GPU execution of all computations. Using a model of a single transmon, we demonstrate how to use these tools to approximate fidelity in a region of model parameter space, as well as construct perturbative robust control objectives.

We also derive and implement Dyson and Magnus-based variations of the recently introduced Dysolve algorithm [Shillito et al., Physical Review Research, 3(3):033266] for simulating linear matrix differential equations. We show how the pre-computation step can be phrased as a multivariable expansion computation problem with fewer terms than in the original method. When simulating a two-transmon entangling gate on a GPU, we find the Dyson and Magnus-based solvers provide a speedup over traditional ODE solvers, ranging from roughly 2\times to 4\times for a solution and 10\times to 60\times for a gradient, depending on solution accuracy.

1 Introduction

Accurate and high-performance simulation of the physics of quantum systems is a key component in device and control engineering workflows in quantum computation. Due
to the complexity of the time-dependent differential equations involved, perturbative techniques are commonly employed to simplify models of these systems. The Dyson series [1] and Magnus expansion [2, 3] are widely used time-dependent perturbation theory tools for studying quantum system dynamics. As perturbative expansions, they enable sensitivity analysis: the quantification of how the dynamics change with local changes to system parameter values. They are utilized heavily in theoretical studies of quantum systems, as they enable perturbative extension of analytically tractable problems through the construction of “effective Hamiltonians” [4, 5, 6, 7]. For example, in superconducting quantum computing, effective Hamiltonian models have been derived for microwave-activated single-qubit [8] and two-qubit gates, such as the cross-resonance gate [9, 10]. In the field of quantum control, starting with Average Hamiltonian Theory [11], these expansions have also been used to design open-loop robust control sequences that suppress the effect of perturbations [12, 13, 14, 15, 16, 17, 18], sequences for sensing by enhancing perturbations [19], dynamical decoupling [20, 21, 22], dynamically corrected gates [23, 24], the filter function formalism [25, 26], and Magnus-based methods for suppressing non-adiabatic errors [27], among others.

More recent work has focused on the numerical computation of these expansions, often with the goal of generalizing the previously mentioned applications to contexts which are not analytically tractable. For robust control, numerical methods have been developed to compute expansion terms in the interaction frame [28, 11, 15] of a control sequence, e.g. the Dyson-like terms of [29, 30] and filter function formalism terms [31, 32, 33], the latter of which have been implemented in software packages [34, 35]. In the context of simulation of quantum systems, Ref. [36] gives an algorithm for computing Dyson-series-derived expressions as part of the pre-computation step of the Dyseolve algorithm, with similar methods appearing in [37].

In this paper we develop software tools for numerically computing and utilizing the Dyson series and Magnus expansion. The general goal is to define and implement algorithms for broad versions of these computational problems, so that they may be used as primitives in many numerical research applications, including robust control and classical simulation of quantum systems. By making the construction of these terms easily accessible, their usefulness in new and existing methods can be explored to higher orders and applied to models with more parameters.

Towards this end, this paper is organized as follows:

• Section 2 defines the multivariable Dyson series and Magnus expansion, and generally introduces the power series and interaction frame notation used throughout the paper.

• Section 3 gives algorithms for computing both multivariable Dyson series and Magnus expansion terms, analyzes their scaling, and describes the implementation in the Qiskit Dynamics package.

• Section 4 demonstrates the usage of the software tools in an example of a robust control problem. In particular, it is shown how the tools can be used to approx-
imate gate fidelity in a region of model parameter space, as well as to construct robustness objectives incorporating higher order expansion terms.

- Section 5 derives Dyson and Magnus-based variants of the Dysolve algorithm \[36\] for solving linear matrix differential equations, and describes an implementation of these algorithms in Qiskit Dynamics, building on the implementation described in Section 3. We benchmark these solvers against the standard ODE solvers available in Qiskit Dynamics for the problem of simulating a two transmon CR gate. We find a speed up over traditional solvers in Qiskit Dynamics of roughly $2 \times$ to $4 \times$ for a solution and $10 \times$ to $60 \times$ for a gradient on GPU, depending on the accuracy of the solutions.

The paper ends with a discussion in Section 6.

2 Multivariable Dyson series and Magnus expansion

We begin by informally introducing the computational problems we consider in this paper. Consider a linear matrix differential equation (LMDE):

$$\dot{U}(t, c_0, \ldots, c_{r-1}) = G(t, c_0, \ldots, c_{r-1}) U(t, c_0, \ldots, c_{r-1}),$$

where $t$ is time, and $c_0, \ldots, c_{r-1}$ are some parameters of the generator $G$ and solution $U$, which are assumed to be perturbative. Note that the Schrödinger equation is an LMDE under the association $G = -iH$, for $H$ the Hamiltonian, along with other common master equations, such as the Lindblad and Bloch-Redfield equations.

Given an integration interval $[t_0, t_f]$ and a power-series decomposition of $G$ in the parameters $c_0, \ldots, c_{r-1}$ (centred at 0):

$$G(t, c_0, \ldots, c_{r-1}) = G_\emptyset(t) + \sum_{k=1}^\infty \sum_{0 \leq i_1 \leq \cdots \leq i_k \leq r-1} c_{i_1} \cdots c_{i_k} G_{(i_1, \ldots, i_k)}(t),$$

with the functions $G_\emptyset$ and $G_{(i_1, \ldots, i_k)}$ being completely arbitrary user-defined functions, the problem is to compute corresponding terms in the truncated power series for the solution $U$ itself:

$$U(t_0, t_f) = U_\emptyset + \sum_{k=1}^\infty \sum_{0 \leq i_1 \leq \cdots \leq i_k \leq r-1} c_{i_1} \cdots c_{i_k} U_{(i_1, \ldots, i_k)},$$

or for the time-averaged generator $\Omega$:

$$\Omega = \Omega_\emptyset + \sum_{k=1}^\infty \sum_{0 \leq i_1 \leq \cdots \leq i_k \leq r-1} c_{i_1} \cdots c_{i_k} \Omega_{(i_1, \ldots, i_k)},$$

\[1\]The Lindblad and Bloch-Redfield equations, which are differential equations for density matrix evolution, are not typically presented in the form of Equation (1). However, their right-hand sides are linear functions of the density matrix, and therefore they can be rewritten in the form of Equation (1) using a vectorization convention.
implicitly defined to satisfy $U(t_0, t_f) = \exp(\Omega)$. 

While we phrase the time-dependent operators $G^\emptyset(t)$ and $G(i_1, \ldots, i_k)(t)$ in Equation (2) as being arbitrary from a computational generality perspective, they are fixed by the parameterization of the generator $G(t, c_0, \ldots, c_{r-1})$. That is, it holds that $G^\emptyset(t) = G(t, 0, \ldots, 0)$ (i.e. $G^\emptyset(t)$ is the unperturbed generator), and each $G(i_1, \ldots, i_k)(t)$ is the partial derivative of $G(t, c_0, \ldots, c_{r-1})$ with respect to the variables $c_{i_1}, \ldots, c_{i_k}$ (up to the required combinatorial pre-factor for multivariable Taylor series). Hence, they are entirely determined by how $G$ is written in terms of the perturbative parameters $c_0, \ldots, c_{r-1}$.

Lastly, we note that the formal algorithms in Section 3 solve the above problem in the interaction frame of $G^\emptyset(t)$ [28, 11, 15], which is reviewed in Section 2.2.

2.1 Power series notation

Using Equation (2) as a model, we introduce notation to simplify working with multivariable power series. Each term in Equation (2) is indexed by a list of indices $0 \leq i_1 \leq \cdots \leq i_k \leq r - 1$. What uniquely identifies the above term is the number of times each index in $\{0, \ldots, r - 1\}$ appears, and hence a multi-index notation [38] is often used for multivariable power series. We use a slightly different, though functionality equivalent, notation in terms of multisets. A multiset is like a set, but in which repeated elements may appear.

Denoting multisets with round brackets, the multiset associated with the above power series term is given as: $I = (i_1, \ldots, i_k)$. Given a list of variables $c_0, \ldots, c_{r-1}$, we denote

$$ c_I = c_{i_1} \times \cdots \times c_{i_k}. $$

(5)

For example, $c_{(0,0,1)} = c_0^2 c_1$, and $c_{(0,1,1,2)} = c_0 c_1^2 c_2$. This notation enables a simple correspondence between algebraic operations and multiset operations. E.g. for two multisets $I, J$, we have that:

$$ c_{I+J} = c_I \times c_J, $$

(6)

where $I + J$ denotes the multiset summation.

With this, for $I = (i_1, \ldots, i_k)$, the summation term $c_{i_1} \cdots c_{i_k} G(i_1, \ldots, i_k)(t)$ in Equation (2) is rewritten as

$$ c_I G_I(t). $$

(7)

Letting $I_k(r)$ denote the set of multisets of size $k$ with elements in $\{0, \ldots, r - 1\}$, and letting $c$ generally denote a list of variables $(c_0, \ldots, c_{r-1})$, we may rewrite Equation (2) as

$$ G(t, c) \equiv G(t, c_0, \ldots, c_{r-1}) = G^\emptyset(t) + \sum_{k=1}^{\infty} \sum_{I \in I_k(r)} c_I G_I(t). $$

(8)

Lastly, we use $|I|$ to denote the number of elements in the multiset, including repeats. For example, $|(0, 0, 1)| = 3$. This notation is used throughout the paper to represent power series.

2 These tasks can equivalently be phrased as performing sensitivity analysis, or numerically finding a series solution, albeit specialized to the case of a linear matrix differential equation.
2.2 Interaction frame

In many quantum control applications, computations are performed in the interaction frame of the unperturbed generator \[28, 11, 15\]. The benefit of the interaction frame is that it factorizes the evolution generated by \( G(t, c) \) into two pieces: one given purely by \( G_\emptyset(t) \), and the other being trivial if \( c = 0 \). Denoting
\[
V(t) = \mathcal{T} \exp \left( \int_0^t ds G_\emptyset(s) \right),
\]
with \( \mathcal{T} \) being the time-ordering operator \[39\], the generator \( G \) in Equation (8), transformed into the interaction frame of \( G_\emptyset \), is given by:
\[
\tilde{G}(t, c) = \sum_{k=1}^\infty \sum_{I \in \mathcal{I}_k(r)} c_I \tilde{G}_I(t),
\]
where \( \tilde{G}_I(t) = V^{-1}(t) G_I(t) V(t) \). With this, it holds that:
\[
\mathcal{T} \exp \left( \int_0^t ds G(s, c) \right) = V(t) \mathcal{T} \exp \left( \int_0^t ds \tilde{G}(s, c) \right).
\]

2.3 Multivariable Dyson series

For a generator \( G(t) \), the Dyson series \[1\] expands
\[
\mathcal{T} \exp \left( \int_0^t ds G(s) \right) = 1 + \sum_{k=1}^\infty D_k(t),
\]
where \( 1 \) is the identity operator, and for the explicit formula
\[
D_k(t) = \int_0^t dt_1 \cdots \int_0^{t_{k-1}} dt_k G(t_1) \cdots G(t_k).
\]
We may view this as a power series in a single variable \( c \) by using this formula for the generator \( cG(t) \):
\[
\mathcal{T} \exp \left( c \int_0^t ds G(s) \right) = 1 + \sum_{k=1}^\infty c^k D_k(t),
\]
where the \( D_k(t) \) are still as in Equation (13).

To define the multivariable Dyson series, the goal is to generalize the above equation and explicit expression in Equation (13) to an arbitrary number of variables, with the original generator given as a power series.

**Definition 1.** Let \( \tilde{G}(t, c) \) be as in Equation (10), with \( c \) representing a list of variables. The multivariable Dyson series is the power series of the solution in \( c \):
\[
\mathcal{T} \exp \left( \int_0^t ds \tilde{G}(s, c) \right) = 1 + \sum_{k=1}^\infty \sum_{I \in \mathcal{I}_k(r)} c_I D_I(t),
\]
with the $\mathcal{D}_I(t)$, which we refer to as multivariable Dyson terms, defined implicitly by the above series decomposition. Note that we also refer to $\mathcal{D}_I(t)$ simply as Dyson terms when there is no risk of ambiguity with the typical “single-variable” Dyson series.

For a multiset $I$, let $P_k(I)$ denote the set of $k$-fold ordered partitions of $I$, which we take to be empty if $|I| < k$. I.e. $P_k(t)$ denotes the set of ordered lists of proper submultisets $I_1, \ldots, I_k \subset I$ such that $I_1 + \cdots + I_k = I$. The following Proposition gives an explicit form for the multivariable Dyson terms $\mathcal{D}_I(t)$.

**Proposition 1.** Letting $\tilde{G}(t,c)$ and the $\mathcal{D}_I(t)$ be defined as in Definition [1], it holds that

$$\mathcal{D}_I(t) = \sum_{m=1}^{|I|} \sum_{I_1, \ldots, I_m \in P_m(I)} \int_0^t dt_1 \cdots \int_0^{t_{m-1}} dt_m \tilde{G}_{I_1}(t_1) \cdots \tilde{G}_{I_m}(t_m)$$

(16)

*Proof.* For the generator $\tilde{G}(s,c)$, the standard Dyson series gives

$$\mathcal{T} \exp \left( \int_0^t ds \tilde{G}(s,c) \right) = 1 + \sum_{k=1}^{\infty} D_k(t)$$

(17)

with

$$D_k(t) = \int_0^t dt_1 \cdots \int_0^{t_{k-1}} dt_k \tilde{G}(t_1,c) \cdots \tilde{G}(t_k,c).$$

(18)

First, we find an explicit power series decomposition of $D_k(t)$ in the variables $c$. We start by implicitly writing $D_k(t)$ as a power series in the variables $c$:

$$D_k(t) = \sum_{m=k}^{\infty} \sum_{I \in I_m(r)} c_I A_{k,I}(t),$$

(19)

where we start the sum at $m = k$, as all terms below this order will be 0. To determine the form of the $A_{k,I}(t)$, we expand each instance of $\tilde{G}(t,c)$ in Equation (18) via the power series in Equation (10), and then collect terms corresponding to each monomial $c_I$. The initial expansion results in a sum of terms of the form:

$$c_{I_1+\cdots+I_k} \int_0^t dt_1 \cdots \int_0^{t_{k-1}} dt_k \tilde{G}_{I_1}(t_1) \cdots \tilde{G}_{I_k}(t_k),$$

(20)

for some list of index multisets $I_j$. Determining the form of $A_{k,I}(t)$ requires identifying all such terms for which $I = I_1 + \cdots + I_k$, which corresponds to identifying ordered partitions of $I$. Collecting such terms results in the equality

$$A_{k,I}(t) = \sum_{(I_1, \ldots, I_k) \in P_k(I)} \int_0^t dt_1 \cdots \int_0^{t_{k-1}} dt_k \tilde{G}_{I_1}(t_1) \cdots \tilde{G}_{I_k}(t_k).$$

(21)

Finally, Equation (16) is obtained by collecting the $A_{k,I}(t)$ for $k \leq |I|$.

\[\square\]
2.4 Multivariable Magnus expansion

For a generator $G(t)$, the Magnus expansion \cite{2,3} alternatively gives an expansion for an operator:

$$\Omega(t) = \sum_{k=1}^{\infty} \Omega_k(t)$$

for which, under suitable convergence conditions, gives:

$$\mathcal{T} \exp \left( \int_0^t ds G(s) \right) = \exp(\Omega(t)).$$

Explicit expressions for the $\Omega_k(t)$ can also be given \cite{2,3}; however we do not work with them here. Similarly to the Dyson series, we can consider the Magnus expansion for the generator $cG(t)$ as a single-variable power series expansion. The operator $\Omega(t)$ is often referred to as the time-averaged generator.

**Definition 2.** Let $\tilde{G}(t,c)$ be as in Equation (10), with $c$ representing a list of variables. The multivariable Magnus expansion for the generator $\tilde{G}(t,c)$ is the power series of the time-averaged generator of the solution in $c$:

$$\Omega(t,c) = \sum_{k=1}^{\infty} \sum_{I \in I_k} c_I \mathcal{O}_I(t)$$

satisfying

$$\mathcal{T} \exp \left( \int_0^t ds \tilde{G}(s,c) \right) = \exp(\Omega(t,c)).$$

The $\mathcal{O}_I(t)$ are defined implicitly according to the above relations, and we refer to them as multivariable Magnus terms, or simply as Magnus terms when there is no risk of ambiguity with the standard Magnus expansion.

We do not derive an explicit expression for $\mathcal{O}_I(t)$ — we only define them implicitly by the above relation. The algorithm we develop for computing them is based on recursion relations rather than explicit expressions.

3 Algorithms, scaling, and implementation

We now introduce the two algorithms that are the main results of this work: one for computing a collection of multivariable Dyson terms $\mathcal{D}_I(t)$, and one for computing a collection of multivariable Magnus terms $\mathcal{O}_I(t)$ recursively from already-computed $\mathcal{D}_I(t)$.

3.1 Computing multivariable Dyson terms

For a desired collection of $\mathcal{D}_I(T)$, the algorithm presented here computes all terms simultaneously, along with the solution of the interaction frame propagator $V(T)$, by
Phrasing them as the solution to a single differential equation, which is then solved via numerical integration. This approach is similar to block-matrix methods used in both the numerical methods [40, 41] and quantum literature [29, 42, 43]. It may also be viewed as a fully time-dependent generalization of the computation performed in the pre-computation step of Dysolve [36], with similar methods appearing in [37]. Furthermore, due to the correspondence between power series and derivatives, the algorithm we present can alternatively be viewed as a forward-mode sensitivity analysis method [44], specialized to linear matrix differential equations, that can compute arbitrary order derivatives in a multivariable setting.\(^3\)

A technical detail of our algorithm is that, for each index multiset \(I\), it computes \(E_I(T) = V(T)D_I(T)\), rather than \(D_I(T)\), where \(V(t)\) is the interaction frame propagator. Depending on application, one may want either \(E_I(T)\) or \(D_I(T)\). As \(V(T)\) is also computed by the algorithm, this factor can be numerically removed if desired by solving the linear equation

\[
V(T)X = E_I(T)
\]

for \(X\) (\(V(T)\) is always in-principle invertible). When there is no risk of confusion, we will refer to both \(D_I(T)\), and \(E_I(T)\) as multivariable Dyson terms.

The algorithm is based on the following proposition, which shows that the derivatives of the \(E_I(t)\) satisfy a recursion relation. The proof is given in Appendix A.1.

**Proposition 2.** Let \(\tilde{G}(t,c)\) and the \(D_I(t)\) be as in Definition 1. For \(E_I(t) = V(t)D_I(t)\), with \(V(t)\) being the interaction frame propagator, it holds that

\[
\dot{E}_I(t) = G_\emptyset(t)E_I(t) + G_I(t)V(t) + \sum_{J \subseteq I} G_J(t)E_{I \setminus J}(t),
\]

where subsets are understood in terms of multisets. For the case \(|I| = 1\), the sum over \(J \subseteq I\) is interpreted as being empty (as there are no proper submultisets).

Hence, to compute a collection of such terms for a desired list of index multisets \(L = \{I_1, \ldots, I_j\}\), we need only solve the above differential equation. The formalized algorithm is given in Algorithm 1. Beyond organizing and structuring the right-hand side function expressed in Equation (27), the algorithm must first complete the list of index multisets \(L = \{I_1, \ldots, I_j\}\) in the following sense: Equation (27) shows that \(\dot{E}_I(t)\) depends on \(E_J(t)\) for every \(J \subseteq I\). Hence, given a particular set of desired terms \(L = \{I_1, \ldots, I_j\}\) to compute, constructing a single coupled differential equation to compute them requires first filling out the list \(L\) until it is closed under taking subsets. More formally, we say that \(L\) is complete if for every \(I \in L\) and \(J \subseteq I\), it holds that \(J \in L\). Hence, the first step of Algorithm 1 is finding the completion of the desired terms \(L\).

\(^3\)The sensitivity-analysis phrasing implies that these terms can be computed using automatic differentiation applied to differential equation solvers. We have found however that, in practice, recursively calling general automatic differentiation routines to compute higher order derivatives is slow compared to the specialized method presented here. See [45] for a discussion of automatic differentiation tools applied to the sensitivity analysis problem for differential equations.
**Algorithm 1:** Compute multivariable Dyson terms

**Input:**
- Callable matrix-valued function $G_\emptyset(t)$ giving the interaction frame generator.
- A list of pairs $(I, G_I(t))$, with $I$ being an index multiset, and $G_I(t)$ a callable matrix-valued function, describing the power series decomposition of the generator. All other power series terms for the generator are assumed by the algorithm to be 0.
- List of index multisets $L = (I_1, \ldots, I_j)$ describing the desired Dyson terms to compute.
- Integration time $T$.
- Boolean flag `remove_V` for whether to return $E_I(T)$ or $D_I(T)$ for each $I \in L$.

**Output:**
- Interaction frame propagator solution $V(T) = \mathcal{T} \exp \left( \int_0^T dt_1 G_\emptyset(t_1) \right)$.
- The completion of $L$, $L'$.
- Multivariable Dyson terms (in the frame of $V(t)$), either $D_I(T)$ if `remove_V` == `True` or $E_I(T)$ otherwise, for each $I \in L'$.

1 Compute the completion $L'$ of $L$ by recursively looping through each $I \in L$ in order of non-increasing length, adding $I \setminus i$ to $L$ for each $I \in L$

2.1 Canonically order the completed set $L' = I_1, \ldots, I_m$

2.2 Represent the state of the DE $y(t) = (V(t), E_{I_1}(t), \ldots, E_{I_m}(t))$ with initial condition $y_0 = (I, 0, \ldots, 0)$, with $I$ the appropriately-sized identity matrix.

2.3 Define RHS function $f(t, y) = (G_\emptyset(t)V(t), G_\emptyset(t)E_{I_1}(t) + \sum_{J \subseteq I_1} G_J(t)E_{I_1 \setminus J}(t), \ldots)$, treating unspecified $G_J(t)$ as 0.

3 Solve the differential equation.

4 if `remove_V` == `True` then

5 For each $I \in L'$, compute $D_I(T)$ by solving $V(T)X = E_I(T)$.

6 Return all the $D_I(T)$, and $V(T)$.

7 else

8 Return all the $E_I(T)$, and $V(T)$. 


3.2 Computing multivariable Magnus terms from multivariable Dyson terms

Here we develop multivariable generalizations of recursive methods for computing Magnus terms from Dyson terms given in [46, 47], though following the notation in [3, Section 2.4]. Recursive methods provide a compact representation of the computational steps while avoiding the complexity of explicit expressions for Magnus expansion terms [3, 48].

To review, the recursive methods of [46, 47] begin by expanding both sides of Equation (23): the left hand is expanded using the Dyson series, and the right-hand side is expanded assuming a series for $\Omega$ and the Taylor series for the exponential. After grouping terms by order and rearranging, [46] gives the formula:

$$
\Omega_k = D_k - \sum_{m=2}^{k} \frac{1}{m!} Q_k^{(m)},
$$

where

$$
Q_k^{(m)} = \sum_{i_1 + \cdots + i_m = k} \Omega_{i_1} \cdots \Omega_{i_m}.
$$

It is then shown in [46] that the $Q_k^{(m)}$ matrices satisfy the recursion relation

$$
Q_k^{(m)} = \sum_{j=1}^{k-m+1} Q_j^{(1)} Q_{k-j}^{(m-1)},
$$

with base case $Q_j^{(1)} = \Omega_j$.

The following proposition, proven in Appendix A.2, shows that direct analogues of these formulas hold in the multivariable case.

**Proposition 3.** Let $D_I(t)$ and $O_I(t)$ be as in Definitions 7 and 3. For all index multisets $I$, it holds that

$$
O_I = D_I - \sum_{m=2}^{\left|I\right|} \frac{1}{m!} Q_I^{(m)}
$$

where the matrices $Q_I^{(m)}$ are defined as

$$
Q_I^{(m)} = \sum_{(I_1, \ldots, I_m) \in P_m(I)} O_{I_1} \cdots O_{I_m},
$$

and satisfy the recursion relation

$$
Q_I^{(m)} = \sum_{J \subseteq I, |J| \leq |I|-(m-1)} Q_J^{(1)} Q_{I \setminus J}^{(m-1)}
$$

with base case $Q_J^{(1)} = O_J$.

Hence, our algorithm for computing multivariable Magnus terms is to first compute the corresponding multivariable Dyson terms, then compute the Magnus terms utilizing the above recursion relation. A formal statement of the algorithm, with full bookkeeping details for implementing the recursion relation, is given in Algorithm 2.
Algorithm 2: Compute multivariable Magnus terms from Dyson terms

Input:
- Complete list of index multisets \( L = (I_1, \ldots, I_m) \).
- Computed Dyson terms \( D_{I_1}(T), \ldots, D_{I_m}(T) \).

Output:
- Magnus terms \( O_I(T) \) for each \( I \in L \).

1 Sort \( L \) in order of non-decreasing length.
2 Generate list of ordered pairs \((I, m)\) that index each \( Q^{(m)}_I \) to compute:
   2.1 For each \( I \in L \), append \((I, |I|), (I, |I| - 1), \ldots, (I, 1)\) to the list of pairs.
   
   /* This list is in an order for which the recursion relation for \( Q^{(m)}_I \) depends only on \( Q^{(n)}_J \) for which \((J, n)\) appears earlier in the list. */
3 Increment through each pair \((I, m)\) in the list, computing \( Q^{(m)}_I \):
4 if \(|I| = 1\) and \( m = 1 \) then
5      Set \( Q^{(1)}_I(T) = D_I(T) \)
6 else if \(|I| > 1\) and \( m = 1 \) then
7      Set \( Q^{(1)}_I = D_I(T) - \sum_{m=2}^{|I|} \frac{1}{m!} Q^{(m)}_I \)
8    else
9      Set \( Q^{(m)}_I = \sum_{J \subset I, |J| \leq |I| - (m-1)} Q^{(1)}_J Q^{(m-1)}_{I\setminus J} \)
10 return \( O_I(T) = Q^{(1)}_I \) for each \( I \in L \).

3.3 Scaling
Here we consider the scaling of the algorithms for computing all terms in the multivariable Dyson series and Magnus expansion in \( r \) variables up to truncation order \( n \), assuming the user supplies non-zero \( G_I(t) \) for all \(|I| \leq n\).

First, the inherent scaling of the problem is determined by the total number of terms at a given truncation order for a given number of variables:

**Fact 1.** The number of terms in a homogeneous multivariate polynomial of order \( n \) in \( r \) variables is [49]:

\[
\binom{r + n}{n} - 1.
\] (34)

If either \( r \) or \( n \) is fixed, the number of terms in the other parameter grows polynomially, which can be seen via the following bound (proof given in Appendix A.3).
Fact 2. It holds that
\[
\binom{r+n}{n} \leq \min(nr^n, rn^r).
\] (35)

On the other hand, for large \( r \) and \( n \), Stirling’s approximation gives
\[
\binom{r+n}{n} \approx \frac{(r+n)^{r+n}}{r^n n^r},
\] (36)
and thus if both \( r \) and \( n \) vary, e.g. setting \( r = n \), the right hand side yields \( 2^{2n} \), which is exponential. Hence, asymptotically there is an inherent exponentiality to the problem along the line \( r = n \), however for either fixed \( r \) or \( n \), the scaling in the other parameter is polynomial.

Finally, for scaling of the algorithms themselves, we have the following bounds on the number of operations:

Fact 3. When computing all terms up to order \( n \) in \( r \) variables:

- For computing Dyson terms, evaluating the RHS in Equation (27) requires
\[
O\left(\left[\binom{r+n}{n} - 1\right]^2\right)
\] operations.

- Carrying out the recursive procedure outlined in Section 3.2 requires
\[
O\left(\left[\binom{r+n}{n} - 1\right]^3\right)
\] operations.

The proofs of these facts are given in Appendix A.3. Hence, the algorithms scale polynomially in the number of terms being computed.

We note again this analysis applies to the case of computing all terms up to a given order for a given number of variables. In practical applications it may be possible to a priori eliminate the need to compute certain subsets of terms, e.g. if they can be shown to be negligible, or, in the case of the Magnus expansion, if terms are known to commute.

3.4 Implementation

Algorithms 1 and 2 have been implemented in the function solve_lmed_perturbation in the perturbation module of the open-source software package Qiskit Dynamics (50). We describe the API here; however note that it may change over time, and hence the package documentation is the best source for up-to-date information.

\[\text{4This function can also be used to compute the Dyson-type terms of [29].}\]
The goal of the `solve_lmde_perturbation` interface is to closely represent the mathematical problems of computing the multivariable Dyson or Magnus terms from the generator power series. Roughly, the inputs are:

- A description of the power series for the generator. This description is given by supplying a description of the non-zero $G_I(t)$, in terms of a list of indices given as `Multiset` objects [51], and a corresponding list of python-callable functions implementing the $G_I(t)$. Any $G_I(t)$ not explicitly given in this description are assumed to be 0.
- A python callable function implementing the interaction frame operator $G_\emptyset(t)$.
- A choice of either Dyson or Magnus expansion.
- A description of which terms in the expansion to compute.
- Arguments describing how to perform the integration for computing Dyson terms, in the form of the differential equation solver to use, and any optional arguments for the solver.

A data storage object is returned providing access to the computed terms.

Additionally, the `perturbation` module contains the class `ArrayPolynomial` for representing an array-valued polynomial of scalar variables. This class has functionality for both evaluating and manipulating the polynomial. As we will show in the demo in Section 4, this enables direct evaluation of truncated Dyson or Magnus expansions for specific values of the power-series variables, as well as other non-trivial computations involving them.

Finally, as will be described in more detail in Section 5, numerical integration schemes for linear matrix differential equations based on the Dyson series and Magnus expansion have also been built into Qiskit Dynamics, using the above perturbation functionality. All functionality can be executed using the JAX [52] array backend. Code examples used to generate the plots in the following section are given in Appendix [D].

4 Demonstration in robust control problem

The goal of robust quantum control is to design control sequences that perform their function in a region of model parameter space. A common approach in numerical robust control is to utilize the Dyson series or Magnus expansion to quantify the sensitivity of a control sequence to variations in model parameters. Control sequences are then designed to reduce the impact of terms in these expansions on the overall evolution. In this section we show how the software tools developed here can be used in these workflows. We emphasize that this section is a software demo; it shows how the tools can be used in application, and is not itself meant to be new research.

Using a model of a closed-system transmon with a variety of model parameters, we demonstrate how to:
1. Compute and use the Magnus expansion to approximate the fidelity of a control sequence, relative to a target gate, in a region of model parameter space. The accuracy of various truncation orders of the Magnus expansion is demonstrated for each model parameter.

2. Construct a robustness function that utilizes Magnus expansion terms to arbitrary order. Note that we are not advocating for the practical usefulness of this particular construction, but are demonstrating how the tools enable such constructions. The usefulness of any such construction must be determined by further research.

The full code for generating the plots can be found in the `control_example.ipynb` Jupyter Notebook in the supplemental repository [53]. In what follows we describe the code as it pertains to the algorithms presented here, but for a full detail see the Notebook.

### 4.1 Transmon model and control parameterization

Here we consider a model of a closed-system transmon, which we model using the Schrodinger equation for unitary evolution:

\[ \dot{U}(t) = -iH(t)U(t), \]  

(39)

where the Hamiltonian \( H(t) \) is Hermitian for all \( t \). We will phrase things in terms of \( H(t) \), rather than \( G(t) = -iH(t) \), as in the preceding sections; however note that all previous discussions and formulae can be translated under the symbolic association \( G = -iH \).

Let \( \mathcal{N} \) be the number operator, and \( a \) be the annihilation operator, which are infinite dimensional matrices defined as

\[
\mathcal{N} = \begin{pmatrix}
0 & 0 & 0 & 0 & \ldots \\
0 & 1 & 0 & 0 & \ldots \\
0 & 0 & 2 & 0 & \ldots \\
0 & 0 & 0 & 3 & \ldots \\
\vdots & \vdots & \vdots & \vdots & \ddots
\end{pmatrix}, \quad \text{and} \quad a = \begin{pmatrix}
0 & 1 & 0 & 0 & \ldots \\
0 & 0 & \sqrt{2} & 0 & \ldots \\
0 & 0 & 0 & \sqrt{3} & \ldots \\
0 & 0 & 0 & 0 & \ddots \\
\vdots & \vdots & \vdots & \vdots & \ddots
\end{pmatrix}.
\]  

(40)

The model of a single transmon that we use is as follows:

\[
H(t) = 2\pi\nu \mathcal{N} + \pi\alpha \mathcal{N}(\mathcal{N} - 1) + \frac{\pi}{3} \beta \mathcal{N}(\mathcal{N} - 1)(\mathcal{N} - 2) + s(b,t)2\pi r(a + a^\dagger),
\]  

(41)

where \( I \) is the identity, \( \nu \) is the qubit frequency, \( \alpha \) is the anharmonicity, \( \beta \) sets the spacing to the 4th energy level, \( r \) is the drive strength, and finally \( s(b,t) \) is the control field, with \( b \) representing a vector of parameters describing the control (which we will explicitly parameterize later). For simulation we truncate the transmon to be 5-dimensional, and choose parameters typical of IBM transmons [10]: \( \nu = 5.0, \alpha = -0.33, \beta = -0.015, \) and \( r = 0.02 \), where all times are in ns and frequencies are in GHz.

We expand the model by adding a list of uncertain parameters: (1) perturbations in the frequency \( \nu \), (2) perturbations in the anharmonicity \( \alpha \), (3) perturbations in the
drive strength \( r \), (4) addition of a non-linear control term proportional to \( s(b, t)^2 \), representing non-linearities in the control electronics, (5) perturbations to the higher level spacing given by \( \beta \), and (6) perturbations to higher level drive operator elements. These parameters represent common uncertainties in a single transmon model, and we select an assortment to demonstrate the numerical behaviour of our algorithms. Denoting the perturbation parameters according to the above ordering as \( c = (c_1, \ldots, c_6) \), this modifies the model to:

\[
H(t, c) = 2\pi \nu (1 + c_1)N + \pi \alpha (1 + c_2)N(N - 1) + s(b, t)2\pi r (1 + c_3)(a + a^\dagger) \\
+ c_4s(b, t)^22\pi r (a + a^\dagger) + (1 + c_5)\frac{\pi}{3} \beta N(N - 1)(N - 21) \\
+ c_6s(b, t)2\pi r P(a + a^\dagger)P,
\]

where \( P \) is the orthogonal projection onto the levels above the first two. By collecting terms according to the coefficients in \( c \), the relevant structure of the Hamiltonian that we will apply perturbation theory to is:

\[
H(t, c) = H_\emptyset(t) + \sum_{j=1}^{6} c_j H(j)(t),
\]

where \( H_\emptyset(t) = H(t, 0) \) is the unperturbed Hamiltonian.

Lastly, we choose a parameterization of the control signal \( s(b, t) \). While the exact details aren’t particularly important for the demonstration, we choose a parameterization that has desirable properties for an optimization application: it produces smooth and bounded signals that start and end at 0, and is automatically differentiable with respect to the control parameters. We use the standard representation utilized in Qiskit Dynamics:

\[
s(b, t) = \text{Re}[f(b, t)e^{i2\pi \mu t}],
\]

where \( f(b, t) \) is the parameterized complex-valued envelope, and the signals carrier frequency is \( \mu \), the expected frequency of the transmon. Here, we set \( \mu = \nu \), the true frequency of the modelled qubit. We construct the envelope \( f(b, t) \) as a piecewise constant complex-valued function via the following process. Starting with the input parameters \( b \), which are a 2-dimensional real array of shape \((2, k)\), we build and iteratively refine the samples for \( f(b, t) \) as follows:

a) First, given a choice of basis vectors \( \{d_j\}_{j=1}^{k} \), construct a pair of unbounded sample arrays \( x_R = \sum_{j=1}^{k} b_{0j}d_j \) and \( x_I = \sum_{j=1}^{k} b_{1j}d_j \). That is, \( x_R \) and \( x_I \) are the starting point for the real and imaginary samples for \( f(b, t) \), and the control parameters \( b \) are the coefficients in the two linear combinations. Here, we choose \( \{d_j\}_{j=1}^{k} \) to be the discretized Chebyshev basis over the interval \([0, 50]\) ns with sample width 1 ns, and use \( k = 8 \). Note, these samples are truly unbounded, as we are allowing the entries of \( b \) to take any real value.

b) Next, given a diffeomorphism \( h : \mathbb{R} \mapsto [-1, 1] \), construct a pair of bounded sample arrays as \( y_R = h(x_R) \) and \( y_I = h(x_I) \) (where we apply \( h \) to a vector by applying
it independently to all entries). By requiring $h$ to be a diffeomorphism, we ensure that all of the values of $y_R$ and $y_I$ lie in the interval $[-1, 1]$, and that this stage of the construction is automatically differentiable. For this diffeomorphism, we choose $h(x) = \frac{\arctan(x)}{\pi/2}.

c) Finally, construct the smoothened complex-valued samples for $f(b, t)$ by convolving $y_R$ and $y_I$ with a smooth kernel. Here, we resample $y_R$ and $y_I$ to be sampled at a rate of 0.125 ns, then convolve each with a discretized Gaussian function (with an amplitude of 1 and standard deviation of 0.5 ns) containing 24 samples. We normalize the samples of the Gaussian convolution kernel so they sum to 1, which ensures that the entries of the convolution output lie within, and can achieve all values in, the interval $[-1, 1]$. Denoting $C$ as the resampling and convolution mapping, the final samples for $f(b, t)$ with sample width 0.125 ns are given by $z = C(y_R) + iC(y_I)$.

The above process is a useful recipe for creating smooth and bounded piecewise constant functions, where the samples are differentiable functions of the unbounded input parameters. See Figure [1] for a visualization of the samples constructed at each step of this process. For all of the demonstrations we use random input parameters to the envelope parameterization.

### 4.2 Fidelity approximation via Magnus expansion

Here we use the Magnus expansion to generate approximations of gate infidelity in a region of model parameter space around $c = 0$, for a random choice of control parameters $b$.\[^5\] Infidelity of a unitary $U$ is defined relative to an $X$ gate on the first two levels of the transmon:

$$E(U) = 1 - \frac{|Tr(X^\dagger U|_2)|^2}{4},$$

where $U|_2$ is the top left $2 \times 2$ block of the unitary $U$. The truncated Magnus expansion, which we denote $\Omega(c)$, as a function of the model parameters $c$ for a fixed $b$, is computed in the frame of $H_{\emptyset}(t)$. The unitary $U(T, c)$ is then approximated as:

$$U(T, c) \approx T \exp \left(-i \int_0^T ds H_{\emptyset}(s) \right) \exp(\Omega(c)),$$

and the approximate infidelity is computed by inputting Equation (46) into the fidelity function in Equation (45). See Appendix D.1 for an outline of the code used to compute

\[^5\]Note that we choose to demonstrate Magnus expansion approximation of the infidelity – rather than the unitary itself – as the infidelity is typically the most important single metric in numerical control optimization schemes. Furthermore, we consider a random choice of control parameters $b$ (as opposed to a choice close to a implementation of a particular gate), as we are generally interested in how the Magnus expansion captures fidelity in a region of model parameter space, regardless of gate quality. Gate optimization procedures need to navigate through both high and low fidelity regions, and as such it is important to test the approximation quality for randomly chosen control parameters, which are a common starting point for optimization.
(a) The unbounded real and imaginary sample arrays $x_R = \sum_{j=1}^{k} b_{0j}d_j$ and $x_I = \sum_{j=1}^{k} b_{1j}d_j$.

(b) The bounded real and imaginary sample arrays $y_R = h(x_R)$ and $y_I = h(x_I)$, with $h(x) = \frac{\arctan(x)}{\pi/2}$.

(c) The final convolved real and imaginary sample arrays $C(y_R)$ and $C(y_I)$.

Figure 1: Sample creation pipeline for the piecewise constant envelope $f(b, t)$, for a random choice of input parameters $b$. The sequential transformations (a), (b), and (c), are described in the main text after Equation (44).

Note that we use the `expm` function in JAX to compute the matrix exponential, which utilizes the combination of Padé approximation and scaling and squaring given in [54]. Once the Magnus expansion is computed to a desired order, the cost of evaluating Equation (46) for a given $c$ is a function of the number of terms in the expansion, and the matrix dimension. Computing $\Omega(c)$ consists primarily of taking a linear combination of the $d \times d$ matrices in the expansion, which depends linearly on the number of terms in the expansion, and quadratically on the dimension $d$. Computing the matrix exponential doesn’t depend on the number of terms, but typically involves many matrix multiplications, which roughly scale as $d^3$, and which we expect to dominate the cost of the computation.

The `odeint` solver is based on the Dormand-Prince method outlined in [55], which is a mixed 4th and 5th order Runge-Kutta variable step-size method.
trend is that, within a neighbourhood of 0, the higher order Magnus truncations provide
a better approximation. However, as the perturbation parameters become larger, the
approximations begin to break down, including the ordering of which order of the Magnus
expansion provides a better approximation. Furthermore, some of the infidelity error
curves exhibit spurious kinks, as in the Order 1 curve for Perturbation 2. As with any
application of perturbation theory, these effects demonstrate that care must always be
taken in choosing a truncation order, and in limiting the region of approximation. The \(c_5\)
perturbation provides an interesting example of numerical breakdown: the perturbation
appears to have very little impact on the infidelity, as evidenced by the extremely flat
infidelity curve in Figure 2. Over this region, however, we still observe breakdown of the
approximations, with higher orders breaking down more quickly. This is plausibly due
to the flatness of the infidelity curve: the Magnus expansion terms are near zero, and
therefore the computed matrices are dominated by numerical error, which are enhanced
at higher order due to the number of matrices involved.

For perturbations \(c_1\) and \(c_3\), Figure 4 demonstrates the quality of infidelity approx-
imation given by the Magnus expansion in a 2d plane. For different expansion orders,
this plot uses a colormap to distinguish between regions of over and underestimation
of the infidelity. With increasing expansion order, we see a growing 2d region of high
quality approximation.

4.3 Magnus robustness objective construction

We now demonstrate the construction of a robustness objective function that aggregates
arbitrary order perturbation data. Following the notation in the previous section, but
introducing explicit functional dependence on the control parameters \(b\), let \(\Omega(c,b)\) denote
a truncated Magnus expansion in the model parameters \(c\), in the interaction frame of
\(-i\tilde{H}(t,c,b)\). Equation (16) shows that the impact of the perturbations on the evolution
is to apply \(\exp(\Omega(c,b))\) before the evolution given by \(-i\tilde{H}(t,0,b)\). The robust control
literature referenced in the introduction builds robustness objectives based on this fact:
if \(\Omega(c,b)\) acts trivially on the system, then the perturbations have no effect. Typical
approaches consider low order terms and attempt to set them all to 0. In numerical
applications, the goal is to construct an objective function that is minimized if all are
0. Building these computations is typically done by hand at low orders. We show here
how the software tools presented in this paper can be used to extend these approaches to
higher orders, for an arbitrary number of perturbations. We emphasize that the practical
utility of this approach needs to be investigated: here we simply wish to show how these
tools can enable such an investigation.

To build a robustness objective, we need to define a function which is minimized if
\(\Omega(c,b)\) has no effect on the subspace of interest. In this case, it is the computational
subspace given by the first two levels of the transmon model. Furthermore, we need this
objective to quantify this over some region of \(c\) values. For this, we choose the following:

\[
g(b) = \int_{c \in \mathcal{D}} \| \Omega(c,b) P - \frac{Tr(\Omega(c,b) P)}{2} P \|_2^2 \, dc,
\] (47)
Figure 2: Infidelity curves along individual perturbation parameters, holding the others as 0. For each value of the perturbation parameters \( c \) represented by the x-axis in the above plots, the “true” unitary \( U(T,c) \) is computed using the odeint solver in JAX with absolute and relative tolerances set to 1e-14, and the infidelity is computed as \( E(U(T,c)) \). The ranges displayed have been chosen by trial and error based on the impact of the perturbation on the fidelity.

where in the above:

- \( D \) is the region of parameter space of interest, and \( p \) is a probability distribution over \( D \),
- \( P \) is the orthogonal projection onto the first two levels, and
- \( \| \cdot \|_2 \) is the Frobenius norm, defined as \( \| A \|_2 = \sqrt{\text{Tr}(A^\dagger A)} \).
Figure 3: Absolute error of the Magnus approximations to infidelity in Figure 2 for each perturbation. For a given Magnus order and perturbation value $c$, the approximation $U$ to $U(T, c)$ is computed as in Equation (46), and the absolute infidelity error is computed $|E(U(T, c)) - E(U)|$. The $y$-axis of each plot is on a logarithmic scale. The $x$-axis for each plot is a percentage of the maximum value displayed for each perturbation in Figure 2 and shown on a logarithmic scale, with linear scaling within $[-10^{-2}, 10^2]$. Negative $x$-axis values correspond to deviations with a minus sign.

To understand the integrand, note that, for a matrix $X$, it holds that:

$$\left\|XP - \frac{Tr(XP)}{2}P\right\|_2^2 = 0$$

(48)

if and only if $XP = aP$ for some $a \in \mathbb{C}$, i.e. if $XP$ is proportional to $P$. For quantum
Figure 4: For perturbations $c_1$ (y-axis) and $c_3$ (x-axis), the 2d infidelity error, on a symmetric base-10 log scale. Note that in contrast to Figure 4 for a given perturbation value $c$ and approximation $U$ to the “true” unitary $U(T, c)$, these figures plot $E(U) - E(U(T, c))$, without the absolute value. Hence, the plot differentiates regions of both over and underestimation of the infidelity. The scale of the $x$- and $y$-axes are linear, and the colour scale is a symmetric log scale, which is linear over $[-10^{-6}, 10^{-6}]$. The white region roughly indicates where the approximations are achieving an error of magnitude below $10^{-6}$.

Systems, this is equivalent to $X$ “acting trivially” on the system: such an operator can only impact global phase of the first two levels, and therefore has no impact on the physical evolution restricted to that subspace. Hence, the objective $g$ is 0 if and only if $\Omega(c, b)$ acts trivially on the computational subspace over a whole region of parameter space.

Effectively choosing the region $D$ and distribution $p$ is itself a challenging problem in multiobjective optimization. This choice effectively sets a linearization strategy for
aggregating all Magnus terms of different orders. The distribution impacts which perturbations are favoured in the optimization, and deciding which to weigh more depends on what impact they have on relevant quantities, such as fidelity. Here, we choose \( p \) to be a product distribution of Gaussians, i.e.:

\[
p(c) = \prod_{i=1}^{r} p_i(c_i),
\]

with \( p_i(c_i) \) given by a Gaussian distribution, with the integration region \( D \) being rectangular.

To compute \( g(b) \), we first identify parts of it that can be pre-computed and reused in each evaluation. Note the following: if \( \Omega(c) \) is a multivariable array-valued polynomial in the variables \( c \), then so is:

\[
h(c, b) = \left\| \Omega(c, b) P - \frac{Tr(\Omega(c, b)P)}{2} P \right\|_2^2.
\]

That is, \( h(c, b) \) necessarily has a decomposition of the form:

\[
h(c, b) = \sum_k \sum_{I \in \mathcal{I}_k(r)} c_I h_I(b),
\]

where we have explicitly included the dependence of the \( h_I \) on the control parameters \( b \). Assuming such a decomposition (which we will need to compute), we have:

\[
g(b) = \int_{c \in D} dc p(c) h(c, b) = \sum_k \sum_{I \in \mathcal{I}_k(r)} \left( \int_{c \in D} dc p(c) c_I \right) h_I(b).
\]

Observe that pre-factors \( \int_{c \in D} dc p(c) c_I \), which are moments of the distribution \( p \), are independent of the control parameters \( b \), and hence can be precomputed once and reused in every evaluation of the objective. Hence, given these pre-computed coefficients, we need only compute the \( h_I(b) \), and then compute the dot product indicated by the above equation. Note that, under the assumption that both \( D \) and \( p \) are symmetric under flipping parameter axes, any moments for which the \( c_I \) contain odd powers will be 0, eliminating many terms that need to be computed.

In Appendix D.2 we walk through how Qiskit Dynamics is used to compute the \( h_I(b) \), highlighting the correspondence between the mathematical procedure described above and the code. Appendix B shows the scaling of the computation time of both \( g(b) \) and its gradient. The scaling is demonstrated with respect to the number of perturbations, for different orders of the Magnus expansion. The CPU plots are generated in the control_example.ipynb notebook of the supplemental repository [53], and the GPU plots in the notebook control_example_gpu.ipynb.
5 Numerical integrators for systems with fast carrier frequencies

The recently introduced Dysolve algorithm [36] utilizes the Dyson series to construct a fixed-step numerical integration method specialized to linear matrix differential equations with fast carrier frequencies. By fixing a particular structure of the generator, Dysolve pre-computes elements of a truncation of the Dyson series, which can then be utilized to repeatedly solve the system for different time-varying envelopes.

In Section 5.1, we review the computational problem of Dysolve, and show how it, and a Magnus-based equivalent, can be phrased in terms of multivariable power series decompositions, and hence the pre-computation step can be performed using the algorithms in Section 3. This leads to a reduction in the number of terms required as reported in [36], in some cases from exponential to polynomial in the truncation order.

In Section 5.2 we describe our implementation of these integrators in the Qiskit Dynamics package, and in Section 5.3 we demonstrate the performance of the implementation, both for solving, as well as for computing gradients with respect to control parameters via automatic differentiation. Finally, in Section 5.4 we discuss the memory requirements of these solvers.

5.1 Perturbative solvers

The computational problem addressed in [36] is to simulate a linear matrix differential equation whose generator is decomposed as

\[ G(t) = F + \sum_{j=1}^{N} \text{Re}[f_j(t)e^{i\omega_j t}]A_j, \]  

where \( f_j(t) \) are complex-valued envelope functions, \( \omega_j \) are carrier frequencies, and \( F \) and the \( A_j \) are constant matrices. More specifically, the goal is to simulate the above system for different instances of the envelopes \( f_j(t) \), while keeping \( F, A_j, \) and the carrier frequencies \( \omega_j \) fixed. With the assumption that \( F, A_j, \) and \( \omega_j \) are fixed, Dysolve [36] utilizes the Dyson series to \text{pre-compute} aspects of the evolution. Given a fixed time-step \( \Delta t \), the approach is to integrate the system over an interval \([t_0, t_0 + \Delta t]\) by computing a truncated Dyson series. While this is generally expensive compared to calling a standard ODE solver, the key observation of [36], phrased in terms of the terminology of this paper, is that the multivariable Dyson series terms associated with a carefully chosen approximate decomposition of the generator are actually independent of both the envelopes and interval start time \( t_0 \) (up to a frame rotation). Hence, the multivariable expansion terms need only be computed once, and can be reused for arbitrary \( t_0 \) and envelopes \( f_j(t) \).

We now explicitly outline the approach in our notation and terminology, describe a Magnus-based version, and discuss how this framing leads to a reduction in the number of terms required to compute and store. We deviate from [36] and consider the generator
in the frame of $F$:

$$
\tilde{G}(t) = \sum_{j=1}^{s} \text{Re}[f_{j}(t)e^{i\omega_{j}t}] \tilde{A}_{j}(t),
$$

(54)

where $\tilde{A}_{j}(t) = e^{-tF} A_{j} e^{tF}$. Fixing the model details, the method is parameterized in terms of a step size $\Delta t$, and a linear approximation scheme for the envelopes. I.e. for $t \in [t_{0}, t_{0} + \Delta t]$, the envelopes are approximated as:

$$
f_{j}(t) \approx d_{j} \sum_{m=1}^{d_{j}} f_{j,m} T_{m}(t - t_{0}),
$$

(55)

where the $f_{j,m}$ are the linear approximation coefficients, and the $T_{m}$ are some chosen basis of functions. The presentation in this section leaves the choice of $T_{m}$ free, though we assume they are real-valued, and implicitly assume some method of computing the coefficients $f_{j,m}$.

Using the envelope approximations in Equation (55) for $t \in [t_{0}, t_{0} + \Delta t]$, the generator $\tilde{G}(t)$ is approximated as:

$$
\tilde{G}(t) \approx \sum_{j=1}^{s} \sum_{m=0}^{d_{j}} \text{Re} \left[ f_{j,m} e^{i\omega_{j}t_{0}} \right] \cos(\omega_{j}(t - t_{0})) T_{m}(t - t_{0}) \tilde{A}_{j}(t)
$$

$$
+ \sum_{j=1}^{s} \sum_{m=0}^{d_{j}} \text{Im} \left[ f_{j,m} e^{i\omega_{j}t_{0}} \right] \sin(-\omega_{j}(t - t_{0})) T_{m}(t - t_{0}) \tilde{A}_{j}(t),
$$

(56)

where we have used that

$$
\text{Re} \left[ f_{j,m} T_{m}(t - t_{0}) e^{i\omega_{j}t_{0}} \right] = \text{Re} \left[ f_{j,m} e^{i\omega_{j}t_{0}} \right] \cos(\omega_{j}(t - t_{0})) T_{m}(t - t_{0})
$$

$$
+ \text{Im} \left[ f_{j,m} e^{i\omega_{j}t_{0}} \right] \sin(-\omega_{j}(t - t_{0})) T_{m}(t - t_{0}).
$$

(57)

The decomposition in Equation (56) gives the generator as approximately equal to a linear combination of time-dependent terms, with the property that dependence of the generator on the envelope functions is entirely compartmentalized in the constant coefficients. It is with respect to this decomposition that we compute either the multivariable Dyson series or Magnus expansion over the interval $[t_{0}, t_{0} + \Delta t]$:

- The coefficients of the expansion $c_{(0)}, \ldots, c_{(r-1)}$ are the

  $$
  \text{Re} \left[ a_{j,m} e^{i\omega_{j}t_{0}} \right] \quad \text{and} \quad \text{Im} \left[ a_{j,m} e^{i\omega_{j}t_{0}} \right]
  $$

  (58)

  in some prescribed order, and

---

8Note “linear” here refers to the $f_{j}(t)$ being written as a linear combination of basis functions, not that the function is necessarily approximated as linear over each interval.

9As described in Section 5.2, our implementation takes the $T_{m}(t - t_{0})$ to be the Chebyshev polynomials defined on the interval $[t_{0}, t_{0} + \Delta t]$, and the $f_{j,m}$ are computed via Discrete Chebyshev Transformation.
The corresponding time-dependent operators $G_j(t_0, t)$ are given by
\[ \cos(\omega_j(t - t_0))T_m(t - t_0)A_j \] (59)
and
\[ \sin(-\omega_j(t - t_0))T_m(t - t_0)A_j, \] (60)
where we have explicitly included the start time $t_0$ in the signature of the $G_j(t_0, t)$. In the above, $r = 2\sum_{s=1}^s d_j$, corresponding to the number of terms appearing in the approximate generator in Equation (56).

Denote $D_I(t_0, t_0 + \Delta t)$ and $O_I(t_0, t_0 + \Delta t)$ as the multivariable Dyson and Magnus terms for a given index multiset $I$ for the above operators, in the frame $F$, with integration time $[t_0, t_0 + \Delta t]$. Critically, these operators satisfy the following time-translation identities:
\[ D_I(t_0, t_0 + \Delta t) = e^{-t_0F}D_I(0, \Delta t)e^{t_0F}, \] (61)
and:
\[ O_I(t_0, t_0 + \Delta t) = e^{-t_0F}O_I(0, \Delta t)e^{t_0F}. \] (62)
That is, the power series terms over an interval of length $\Delta t$ can be translated to different start times via a frame transformation. This was shown in [36] for the Dyson series, and we provide a proof for both the Dyson series and Magnus expansion for our particular setup in Appendix C.1.

Thus, after pre-computing the relevant collection of $D_I(0, \Delta t)$ or $O_I(0, \Delta t)$, the solution $U(t_0, t_0 + \Delta t)$ over the interval $[t_0, t_0 + \Delta t]$ for the generator $\tilde{G}(t)$ is approximated via the following steps:

- Compute the series variables $c_i$ by computing the envelope approximation coefficients $f_{j,m}$, and computing $\text{Re} \left[ f_{j,m}e^{i\omega_j t_0} \right]$ and $\text{Im} \left[ f_{j,m}e^{i\omega_j t_0} \right]$ in suitable order.
- Evaluate the truncated series $\sum_I c_I D_I(0, \Delta t)$ or $\sum_I c_I O_I(0, \Delta t)$.
- In the case of the Magnus expansion, exponentiate the above results.
- Conjugate the results from the previous step by $e^{-t_0F}(\cdot)e^{t_0F}$ to translate the truncated series to the right starting time $t_0$.

In Appendix C.2 we describe how to save on the frame translation steps when simulating over a contiguous series of intervals.

We end by noting the differences between our presentation and that of [36] in the case of the Dyson series. First, consider the number of perturbation terms required to compute, store, and take linear combinations of. According to Section 3.3, utilizing an $n^{th}$ order in the series, given the approximate generator decomposition of Equation (56), requires $\left(\begin{array}{c} n + r \\ n \end{array}\right) - 1$, with $r = 2\sum_{s=1}^s d_j$. Ref. [36] states that when approximating to the $n^{th}$ order in the Dyson series, for a single term in the sum in Equation (54) (corresponding to $s = 1$), and using only a single term in the envelope decomposition in Equation (55) (corresponding to $d_1 = 1$), the Dyson algorithm requires computing
$2^{n+1} - 1$ perturbation terms. However, in this case, we have $r = 2$, and hence our version only requires
\[
\binom{n + 2}{n} - 1 = \frac{(n + 2)(n + 1)}{2} - 1
\] (63)
terms for expansion order $n$, which is polynomial rather than exponential in the order.

Second, in this paper we set up the problem in the frame of the constant operator $F$, then truncate the Dyson series at a given order, whereas [36] expands the propagator in the “lab frame”. In terms of the resulting operators, the two are related by left-multiplication by $e^{-\Delta tF}$, so there is no fundamental difference between the two choices.

5.2 Implementation in Qiskit Dynamics

The classes DysonSolver and MagnusSolver implement the perturbative solvers described above. For the envelope approximation over each interval, we choose to use Chebyshev polynomials, with the approximation coefficients in Equation [55] computed via Discrete Chebyshev Transform. The matrix exponentials in MagnusSolver are computed using the expm function in JAX, which uses the method in [54]. These solvers have been implemented to evaluate the approximate solution over all sub-intervals simultaneously in a vectorized way, and the overall solution is then computed by multiplying these together.

The API and behaviour of these classes, as of version 0.3.0, are described in Appendix D.3. The main goal of this implementation is to enable easy configuration of the various solver parameters, to enable performance investigations as in the next section.

5.3 Performance demonstration

The potential performance gains of these methods are demonstrated in [36], in which their implementation of the method is compared against the solver in QuTiP [56]. Here, we compare the performance of our implementation against the traditional ODE solvers available in Qiskit Dynamics.

We consider a model of two interacting transmons in the Duffing approximation described by Hamiltonian:
\[
H(t) = 2\pi\nu_0 N_0 + \pi\alpha_0 N_0(N_0 - I_0) \\
+ 2\pi\nu_1 N_1 + \pi\alpha_1 N_0(N_1 - I_1) \\
+ 2\pi J(a_0 a_1^\dagger + a_1 a_0^\dagger) \\
+ s_0(t) \times 2\pi(a_0 + a_0^\dagger) \\
+ s_1(t) \times 2\pi(a_1 + a_1^\dagger),
\] (64)
where for transmon $j \in \{0, 1\}$:

\[^{10}\text{These comparisons were performed using the Qiskit Dynamics main branch on commit 948809 (to be included in version 0.4.1), JAX version 0.4.8, and CUDA 12.}\]
\( a_j, N_j, \) and \( 1_j \) are the raising, number, and identity operators,

- \( \nu_j \) is the frequency and \( \alpha_j \) is the anharmonicity,

- \( J \) is the coupling strength, and

- \( s_j(t) = Re[f_j(t)e^{i\omega_j t}] \) is the drive signal on the transmon.

The parameters are chosen to model qubit pair [Q3, Q5] of \texttt{ibmq_montreal} as reported on 04/19/2022:

- \( \nu_0 = 5.105, \nu_1 = 5.033, \alpha_0 = -0.33516, \alpha_1 = -0.33721, \) and \( J = 0.002, \)

where Q3 is indexed by 0, and Q5 by 1.

For the drive signals \( s_0(t) \) and \( s_1(t) \), we choose the Direct CX pulse described in Fig. 7 of [57], with the control being qubit 0 (modeling Montreal Q3), and the target being qubit 1 (modeling Montreal Q5). The Direct CX is a maximally entangling \( ZX \) rotation between the two qubits facilitated by three drive tones at the target qubit’s transition frequency \( \nu_1 \). One drive tone is applied to the control qubit, and the other two pulses are symmetric and asymmetric pulses of variable amplitude and phase applied to the target qubit. The shapes are shown in Figure 5 and the explicit mathematical forms, as functions of the amplitude \( A \), gate time \( T \), risetime \( r \) (the length of time over which the pulse goes from 0 to max amplitude), and \( \sigma \) (width of shape during risetime), are given in Appendix C.3. In the simulations below, we treat the amplitudes and phases of the three pulses as variable parameters, while \( T = 200, r = 7, \) and \( \sigma = 7, \) are fixed. All times and frequencies are in ns and GHz.

The benchmarking folder in the supplementary code repository [53] contains scripts for comparing the Dyson and Magnus perturbative solvers against standard ODE solvers for this problem[11]. We consider both the time to generate the solution, as well as the time to compute the gradient of some real-valued function of the solution with respect to the pulse parameters. For setting up the perturbative solvers, we set the static part and frame operator to:

\[
F = -i(2\pi \nu_0 N_0 + \pi \alpha_0 N_0 (N_0 - 1) + 2\pi \nu_1 N_1 + \pi \alpha_1 N_0 (N_1 - 1) + 2\pi J (a_0 a_1^\dagger + a_0^\dagger a_1))
\]

(65)

and the control operators to:

\[
A_0 = -i2\pi (a_0 + a_0^\dagger) \quad \text{and} \quad A_1 = -i2\pi (a_1 + a_1^\dagger).
\]

(66)

Note that \( A_0 \) corresponds to the operator acting on the control qubit, and hence is modulated by \( s_0(t) \) whose envelope is given by the control drive in Figure 5 and \( A_1 \) is the operator acting on the target qubit, which is modulated by \( s_1(t) \) whose envelope is

[11] The exact data used in the plots presented in this section is stored in the benchmarking/data folder.
The sum of the target drives in Figure 5. For setting up standard ODE solvers, we use the Solver class in Qiskit Dynamics, specifying the Hamiltonian decomposition required by it, and also specifying the simulation to be in the rotating frame of the static part (we have found that this generally improves performance). The odeint ODE solver available in JAX [52] is used (see footnote 7 for a description of odeint). For all solvers, we use a truncation dimension of 5 for each transmon, for a total system dimension of 25.

To gain a picture of the trade-off between speed and accuracy of both odeint and the perturbative solvers, we choose many different configurations of each solver, then compare, for randomly chosen input control parameters, the total time to solve vs the average accuracy of the generated solutions relative to benchmark solutions. This approach is chosen as:

- It allows for comparison of solvers with very different forms of configuration (e.g. tolerances vs fixed step size), and
- We do not know a priori what is the best way to choose all of the parameters of the perturbative solvers, and want to get an empirical sense of what speed vs accuracy ratios are possible.

Additionally, to fairly compare the perturbative solvers to odeint, it is necessary to

12The same random input parameters are used for all configurations, to ensure a fair comparison.
account for the fact that the perturbative solvers naturally utilize parallelization.\textsuperscript{13} As such, on GPU we use the JAX \texttt{vmap} transformation to vectorize calls to \texttt{odeint}, effectively parallelizing calls to this solver. As shown in Figure 11 in Appendix C.4, the benefits of this parallelization for \texttt{odeint} on GPU are saturated around 7000 inputs, so we choose this number of inputs for speed comparisons to the perturbative solvers. By contrast there appear to be no benefits to vectorizing our implementation of the perturbative solvers on GPU (which already consist of parallel operations), and hence for these solvers we run the 7000 simulations using a serial loop. For CPU comparisons, which are shown in Appendix C.4 we use a single core for both \texttt{odeint} and the perturbative solvers, as we have found that the speed of both types of solvers scales sub-linearly with the number of cores utilized. The CPU comparisons are performed with only 100 inputs, which is enough to average out performance variations and just-in-time compilation costs for a single core.

For measuring solution accuracy, we use the distance metric between solutions $U, V$:

$$
\|U - V\|_2 / \sqrt{d},
$$

where $d$ is the dimension of the whole space. This metric is chosen as it is used for the error estimation in the \texttt{odeint} solver. Using this metric, the accuracy of a given solution at the final time is measured via the distance to a benchmark solution, computed using \texttt{odeint} with tolerances set to $1e^{-14}$ (using 64-bit precision).

In terms of configurations, for \texttt{odeint}, we consider various choices of tolerances, setting both \texttt{rtol} and \texttt{atol} to all powers of 10 within the range of $1e^{-6}$ to $1e^{-14}$. For both the Dyson and Magnus perturbative solvers, we consider all possible combinations of the following parameter choices:

- The Chebyshev approximation order of each signal, either 0, 1, or 2,
- The order of the expansion, from 2 to 5,
- The number of time steps used to simulate over $T = 200$. We divide the interval into $M = 10^4, 2 \times 10^4, 3 \times 10^4, 4 \times 10^4$, or $5 \times 10^4$, and set $dt = T/M$.

Figure 6 shows the speed v.s. accuracy trade-off for the best performing solver configurations when run on the Nvidia A100 GPU with 80GB of memory. The Dyson and Magnus-based solvers outperform the traditional ODE solver to various degrees depending on the accuracy level. As shown in Figure 7 the speedups from using the perturbative solvers range from $2 \times$ to $4 \times$, and for gradient computations, from about $10 \times$ to $60 \times$. The full data, showing performance on GPU for all tested solver configurations, is shown in Figure 12 in Appendix C.4.

\textsuperscript{13}The perturbative solvers consist of operations that can be naturally parallelized, whereas the ODE solvers are fundamentally serial computations. As such, comparisons running a single simulation at a time would be misrepresentative of performance; a single run of a traditional ODE solver may take more time due to the number of serial steps, but actually use far less of a device at any given time (either a GPU, or a multicore CPU). This is especially true for the system dimension considered here, which is small.
Figure 6: Total Runtime v.s. Average Distance for (A) computing the final unitary for 7000 randomized control parameters and (B) computing the gradient of the fidelity of the final unitary with respect to the same pulse parameters, computed on an Nvidia A100 GPU with 80GB of memory. For both Dyson and Magnus solvers, the label \((m, n)\) denotes the configuration parameters used: a \(m\)-order Chebyshev approximation of the signals, and an \(n\)th order perturbative expansion. For a given solution \(U\), distance is computed via the metric \(\|U - V\|_2 / \sqrt{d}\), where \(V\) is a benchmark solution computed using \texttt{odeint} at absolute and relative tolerances \(1e^{-14}\). Average distance is the arithmetic mean of these values. For each configuration, the data points correspond to various numbers of time-steps \(M = 10^4, 2 \times 10^4, 3 \times 10^4, 4 \times 10^4,\) or \(5 \times 10^4\). This plot only contains select data of the best performing perturbative solver configurations. The \texttt{odeint} solutions were computed by vectorizing the computation over all inputs at once, and the perturbative solvers computed all solutions in a serial loop. Full data for both CPU and GPU-based simulations can be found in Appendix C.4. Note the peculiar feature that some points in plot (B) lie below the corresponding points in plot (A), indicating that the gradient computation is actually faster in some instances. While the fidelity computation of plot (B) seemingly involves more computation than the unitary computation in plot (A), we believe this is due to optimizations in the JAX compilation taking advantage of the fidelity only depending on a sub-block of the full unitary.

The same speed comparisons are shown for CPU in Figures 14 and 15 in Appendix C.4. For the CPU case, we conclude that this implementation of the perturbative solvers is unlikely to be practically useful. At best, modest speedups are observed, and due to the memory requirements of the perturbative solvers, they will not easily scale up to be run in parallel on many CPU cores. See Appendix C.4 for details. As the speedups for these solvers are likely very problem-dependent, more real-world usage is necessary to draw more conclusions. They may get better or worse as the dimension of the system scales, and will certainly be limited by the number of time-dependent signals appearing in the generator decomposition in Equation (54).

Lastly, the full data plots for GPU in Appendix C.4 show that the Magnus solver tends to be slower than the Dyson solver, however it becomes comparable, and even
Figure 7: Speedup vs average distance for data in Figure 6. For each ODE solver point, the speedup is computing the solution time ratio of the fastest perturbative solver (drawing from both Dyson and Magnus solvers) achieving an average distance no worse than the ODE solver point. Plot (A) shows the speedup for computing the final unitary $U$, in which we observe a speedup of roughly $2 \times$ at very low average distances, and up to $4 \times$ for higher average distances. Plot (B) demonstrates the corresponding data for computing solution gradients. For gradients, the speedup is more dramatic, potentially owing to the perturbative solvers being an “easier” computation to automatically differentiate as compared to traditional ODE solvers.

slightly faster than the Dyson solver for the highest accuracy solutions. Furthermore, the Magnus solver generally seems to produce higher accuracy solutions: on average, the Magnus solver points appear to have lower average distance. We suspect that these observations are the result of a trade-off with the Magnus-based approach: it requires a matrix exponential at every step, which is costly, but requires fewer expansion terms to achieve the same level of accuracy as the Dyson-based approach. This latter point is evidenced by Figure 8, which plots the average distance attained by the Dyson and Magnus solvers vs the number of expansion terms used in a given configuration. Generally, for the same number of terms and time-steps, the Magnus solver achieves better accuracy, and hence may be of more use in memory-limited scenarios.

5.4 Memory requirements and potential improvements

Lastly, we end this section with a discussion on memory requirements of the perturbative solvers. To simplify the discussion, assume that all signal envelopes are approximately decomposed as a linear combination of $d_c$ functions in Equation (55). As described in Section 5.1, the number of terms appearing in the approximate generator at each time step are $r = 2sd_c$, where $s$ is the number of signals. Hence, for a Dyson or Magnus expansion of order $n$, the pre-computation step for the perturbative solvers involves
Figure 8: The average distance achieved vs the number of terms used in the expansion for both Dyson and Magnus for (A) $M = 10^4$ time-steps and (B) $M = 5 \times 10^4$ time-steps. The number of terms is determined as follows: Section 3.3 shows that, for $r$ variables and expansion order $n$, there are $\binom{r+n}{n} - 1$ terms to compute and store in the expansion. In this case, the number of variables is $r = 2\text{ (number of signals)} \times 2\text{ (real/imaginary parts)} \times \text{ chebyshev_order}$. For a given number of terms, the Magnus solver always achieves a lower average distance per number of terms than the Dyson solver, in some cases significantly so. This data supports the notion that the Magnus solvers may be of more use in memory-limited scenarios, in which fewer terms can be stored and utilized.
solving a differential equation whose solution is an array of
\[
\binom{n + 2sd_c}{n} - 1
\] (68)

complex matrices of dimension \(d\). For the example in the preceding section, where the matrix dimension is 25 and we use 64 bit floating point numbers, each matrix requires 10 kilobytes of memory. For the case \(d_c = 3\) (corresponds to Chebyshev order 2) and \(n = 5\), Equation (68) evaluates to 6187, and therefore the storage of the perturbative expansion terms requires about 59 megabytes of memory. The pre-computation step itself therefore should require some small constant multiple of 59 megabytes of memory, depending on the number of copies of the state the ODE solver stores at any given time. For these parameters, we observe roughly 700 megabytes of memory usage on a personal laptop during the pre-computation step, which is about 10 to 12 \(\times\) the memory required to simply store the matrices.

After the pre-computation step is completed, to solve over \(M\) time steps, we have implemented the solvers to evaluate the approximate solution over each time step simultaneously using vectorization. The solution at the final time is then computed by multiplying these together using the associative scan looping construct in JAX, which utilizes vectorized operations to simultaneously carry out independent computations. For this step it is difficult to exactly estimate the memory requirements, as the underlying compilation can merge operations in non-trivial ways, and would require a detailed understanding of the algorithm underlying the associative scan function. For example, for \(M = 40000\), storing the \(M\) matrices (with \(d = 25\)) requires about 381 megabytes of memory. However, when running the Dyson solver for \(M = 40000\) and the parameters in the preceding paragraph, it uses about 10 gigabytes of memory.

As we have observed, even with the heavy memory usage of this implementation, the solvers provide an advantage over traditional solvers on GPU. Tradeoffs between memory usage and speed of the solvers could be further explored by changing how parallelization is utilized. For example, the memory usage could be decreased using a serial loop to construct and propagate the solution over each time step, rather than using associative scan. While this would slow down an individual instance of the solver, it would become more feasible to run multiple instances in parallel. Alternatively, a custom version of associative scan could be written to only compute the solution at the final time (or a list of desired times), eliminating some redundancy in its output (the definition of the problem associative scan solves, for our problem, amounts to explicitly computing the solution at \(t_0 + k\Delta t\) for all \(k\) up to \(M\)).

Lastly, the heavy memory requirements of these solvers impact the system size they can feasibly be applied to. Even at dimension 25 they use significant memory, and as such, applying them to larger systems will require tweaking the methods to reduce memory footprint. Whether or not they provide an advantage in larger systems is an open question warranting further investigation.
6 Discussion

We have developed algorithms and software tools for numerically working with the Dyson series and Magnus expansion in a multivariable setting, with the goal of facilitating numerical research applications in quantum control and device engineering. We have demonstrated the tools in the context of robust control, and built higher-level numerical tools in the form of solvers based on the Dyson series and Magnus expansion, demonstrating speed ups on GPU in a simulation of a two transmon gate.

In terms of the Dyson and Magnus-based solvers, given their speed on GPU, it would be helpful to develop more automated ways of choosing the parameters, or to devise a variable-step version of them. Given the phrasing of the pre-computation step in this paper as an ODE, the step size could be dynamically adjusted by solving the ODE from the current step size to a new one. The comparison of low order vs higher order formula typically utilized in variable-step solvers could be done between different expansion orders and signal envelope approximation orders.

Finally, an important observation is that the ODE derived to compute Dyson terms in Algorithm 1 for linear matrix differential equations is itself a linear matrix differential equation. That is, the right hand side is a linear function of the state, and as it only consists left-multiplication of the state, it can be rewritten as a large matrix multiplying the state (e.g. as is done in [29]). As such, the Dyson and Magnus-solvers could be applied to this ODE, which could speed up the computations performed in the robust control demo when executed on GPU.

Acknowledgements

We thank Holger Haas and Haggai Landa for helpful discussions and feedback. We are also grateful to the reviewers for improving the clarity of the paper with their careful reading and helpful comments.

A Proofs

A.1 Multivariable Dyson term recursion relation

Proof of Proposition 2. For an index multiset I with \(|I| \geq 2\), we can relate \(D_I(t)\) recursively to lower order terms utilizing the explicit expression given in Proposition 1:

\[
D_I(t) = \sum_{m=1}^{\binom{|I|}{2}} \sum_{(I_1,\ldots,I_m) \in \mathcal{P}_m(I)} \int_0^t dt_1 \cdots dt_{m-1} dt_m \tilde{G}_{I_1}(t_1) \cdots \tilde{G}_{I_m}(t_m) \\
= \int_0^t dt_1 \tilde{G}_I(t) + \sum_{m=2}^{\binom{|I|}{2}} \sum_{(I_1,\ldots,I_m) \in \mathcal{P}_m(I)} \int_0^t dt_1 \cdots dt_{m-1} dt_m \tilde{G}_{I_1}(t_1) \cdots \tilde{G}_{I_m}(t_m).
\]
We can reorganize the sum: sum first over \( J \subset I \) for the first partition in the integral, and then sum over all ordered partitions of \( I \setminus J \):

\[
\mathcal{D}_I(t) = \int_0^t dt_1 \tilde{G}_I(t)
+ \sum_{J \subset I} \sum_{m=1}^{[I \setminus J]} \sum_{(I_1, \ldots, I_m) \in P_m(I \setminus J)} \int_0^t ds \tilde{G}_J(s) \int_0^s dt_1 \cdots \int_0^{t_{m-1}} dt_m \tilde{G}_{I_1}(t_1) \cdots \tilde{G}_{I_m}(t_m)
= \int_0^t dt_1 \tilde{G}_I(t) + \sum_{J \subset I} \int_0^t dt_1 \tilde{G}_J(t_1) \mathcal{D}_{I \setminus J}(t_1).
\]

Left-multiplying the above equation by \( V(t) \) on both sides and differentiating yields Equation (27).

\section*{A.2 Multivariable Magnus terms recursion relation}

\textbf{Proof of Proposition 3.} Here we prove Equations (31) and (33) by walking through the multivariable analogue of the derivations in [46, 47]. Starting from the definitions of the multivariable Dyson series and Magnus expansion, we have:

\[
1 + \sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \mathcal{D}_I(t) = \exp \left( \sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \mathcal{O}_I(t) \right) \quad (71)
\]

Expanding the right hand side using the Taylor series for the exponential gives:

\[
1 + \sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \mathcal{D}_I(t) = 1 + \sum_{m=1}^{\infty} \frac{1}{m!} \left( \sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \mathcal{O}_I(t) \right)^m.
\]

By expanding out the power, and recollecting terms in the coefficients \( c_I \) we may observe that

\[
\left( \sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \mathcal{O}_I(t) \right)^m = \sum_{k=m}^{\infty} \sum_{(I_1, \ldots, I_m) \in P_m(I)} c_I \mathcal{O}_{I_1}(t) \cdots \mathcal{O}_{I_m}(t),
\]

where \( k \) starts at \( m \) as the \( m^{th} \) power only contains elements \( c_I \) for which \( |I| \geq m \). This leads to the power series equality

\[
\sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \mathcal{D}_I(t) = \sum_{m=1}^{\infty} \frac{1}{m!} \sum_{k=m}^{\infty} \sum_{(I_1, \ldots, I_m) \in P_m(I)} c_I \mathcal{O}_{I_1}(t) \cdots \mathcal{O}_{I_m}(t).
\]

Swapping the order of summation for \( k \) and \( m \) on the right hand side yields:

\[
\sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \mathcal{D}_I(t) = \sum_{k=1}^{\infty} \sum_{I \in \mathcal{I}_k(r)} c_I \left( \mathcal{O}_I(t) + \sum_{m=2}^{\infty} \frac{1}{m!} \sum_{(I_1, \ldots, I_m) \in P_m(I)} \mathcal{O}_{I_1}(t) \cdots \mathcal{O}_{I_m}(t) \right).
\]

(75)
Equating terms with the same coefficients leads to the recursion relation:

\[ O_I(t) = D_I(t) - \sum_{m=2}^{\lfloor I \rfloor} \frac{1}{m!} \sum_{(I_1, \ldots, I_m) \in \mathcal{P}_m(I)} O_{I_1}(t) \ldots O_{I_m}(t). \]  

(76)

Defining \( Q^{(m)}_I \) as in Equation (32):

\[ Q^{(m)}_I = \sum_{(I_1, \ldots, I_{m-1}) \in \mathcal{P}_{m-1}(I \setminus J)} O_{I_1} \ldots O_{I_{m-1}} \] 

(77)

and subbing into Equation (76) yields Equation (31).

To derive the recursion relation in Equation (33), we use the same trick as in the Dyson case: break the sum over the first multiset \( I_1 \) and a sum over the remaining members of the ordered partition:

\[ Q^{(m)}_I = \sum_{J \subset I, |J| \leq |I|-(m-1)} O_J \sum_{(I_1, \ldots, I_{m-1}) \in \mathcal{P}_{m-1}(I \setminus J)} O_{I_1} \ldots O_{I_{m-1}} \]

\[ = \sum_{J \subset I, |J| \leq |I|-(m-1)} O_J Q^{(m-1)}_{I \setminus J} \]

\[ = \sum_{J \subset I, |J| \leq |I|-(m-1)} Q^{(1)}_{J} Q^{(m-1)}_{I \setminus J}. \]  

(78)

\[ \Box \]

A.3 Algorithmic scaling bounds

**Proof of Fact 3.** It holds that

\[ \binom{r+n}{n} - 1 = \sum_{m=1}^{n} \sum_{1 \leq i_1 \leq \cdots \leq i_m \leq r} 1 \leq \sum_{m=1}^{n} \sum_{1 \leq i_1, \ldots, i_m \leq r} 1 = 1 + r + \cdots + r^n \leq 1 + nr^n, \]  

(79)

where in the inequality we have removed the ordering requirements on \( i_1, \ldots, i_m \). This establishes the required upper bound \( nr^n \), and the upper bound \( rn^r \) follows from \( \binom{r+n}{n} \) being symmetric under the exchange \( r \leftrightarrow n \).

**Proof of Fact 3.** For computing the RHS for all terms in Equation (27), observe that computing the derivative of a single \( E_I(t) \) requires computing as many products as there are submultisets of \( I \). The number of submultisets is trivially bounded by the total number of terms being computed. Summing this bound over all terms being computed results in the computation of the RHS requiring the square of the number of terms of matrix multiplications and matrix additions, yielding the desired bound.

The recursive algorithm for computing Magnus terms from Dyson terms consists of computing all of the \( Q^{(m)}_I \) matrices. For a fixed index multiset \( I \), there are \(|I|\) values
of $m$. The worst case cost of computing a given $Q_I^{(m)}$ occurs when $m > 1$. Computing a single such term involves performing as many matrix multiplications and additions as there are submultisets $J \subset I$ satisfying $|J| \leq |I| - (m - 1)$. For a coarse bound, we can simply bound the number of such sets with the total number of terms being computed, $\binom{r+n}{n} - 1$. Hence, computing a single term $Q_I^{(m)}$ requires

$$O\left(\binom{r+n}{n} - 1\right)$$  \hfill (80)

operations. Finally, note that there are $\binom{r+n}{n} - 1$ values of $I$, and for each $I$, $|I|$ values of $m$. We may coarsely bound $|I| \leq n \leq \binom{r+n}{n} - 1$, and hence the cost of performing the recursion relation for all terms requires

$$O\left(\left[\binom{r+n}{n} - 1\right]^3\right)$$  \hfill (81)

operations.

\section*{B Robustness objective computation time}

The robustness objective computation time of Section 4.3 is given in Figure 9 for CPU, and Figure 10 for GPU.
Figure 9: CPU timing plots for computing the robustness $g(b)$ in Section 4.3. The top left plot shows the cost of computing $g(b)$, including the JAX compilation time, and the top right shows the cost once compiled. Similarly, the bottom left plot shows the cost of computing the gradient of $g$, including compilation, and the bottom right shows the cost of the gradient once compiled. All plots are generated on the CPU of a personal laptop.
Figure 10: The GPU equivalent of Figure 9. Computations were done using an Nvidia A100 GPU with 80GB of memory.
C Numerical integrators for systems with fast carrier frequencies

C.1 Time translation formula derivation

Starting from Equation (56), we can rewrite:

\[ \tilde{A}_j(t) = e^{-tF} A_j e^{tF} = e^{-t_0 F} A_j e^{t_0 F} \]

Subbing this into Equations (59) and (60), we find that the \( G_I(t_0, t) \) operators satisfy

\[ \tilde{G}(t_0, t) = e^{-tF} G_I(t_0, t) e^{tF} = e^{-t_0 F} \tilde{G}(0, t - t_0) e^{t_0 F}. \]

Finally, note that the \( D_I(t_0, t_0 + \Delta t) \) and \( O_I(t_0, t_0 + \Delta t) \) are both linear combinations of integrals of the form (or, of products of integrals of the following form):

\[ \int_{t_0}^{t_0 + \Delta t} dt_1 \cdots \int_{t_0}^{t_k - 1} dt_k \tilde{G}(t_0, t_{\sigma_1} - t_0) \cdots \tilde{G}(0, t_{\sigma_k} - t_0) \]

for some permutation \( \sigma \). Equation (83) gives that the above integral is equal to

\[ e^{-t_0 F} \left( \int_{t_0}^{t_0 + \Delta t} dt_1 \cdots \int_{t_0}^{t_k - 1} dt_k \tilde{G}(0, t_{\sigma_1} - t_0) \cdots \tilde{G}(0, t_{\sigma_k} - t_0) \right) e^{t_0 F}, \]

and performing a change of variables \( s_j = t_j - t_0 \) for all integration variables yields

\[ e^{-t_0 F} \left( \int_{0}^{\Delta t} ds_1 \cdots \int_{0}^{s_{k-1}} ds_k \tilde{G}(0, s_{\sigma_1}) \cdots \tilde{G}(0, s_{\sigma_k}) \right) e^{t_0 F}. \]

Equations (61) and (62) immediately follow.

C.2 Simulation frame simplification

When simulating over a contiguous series of intervals with endpoints \( t_k = t_0 + k \Delta t \), the frame handling steps can be simplified. Let \( W_k \) denote the approximation to the propagator at the \( k \)th interval, before performing the final frame translation step. Over \( M \) contiguous intervals starting at \( t_0 \), the approximate evolution in the frame \( F \) is then given as:

\[ e^{-t_{M-1} F} W_{M-1} e^{t_{M-1} F} \cdots e^{-t_0 F} W_0 e^{t_0 F}. \]

Each intermediate frame product is of the form: \( e^{t_{k+1} F} e^{-t_k F} = e^{\Delta t F} \). Hence, rather than computing \( e^{t_k F} \) for every \( t_k \), we only need to compute 3 exponentials \( e^{-t_{M-1} F}, e^{t_0 F}, \) and \( e^{\Delta t F} \), and each time-step requires only one extra multiplication as opposed to two.

In the case of the Magnus expansion no further simplifications are possible, but for the Dyson series, the pre-factor \( e^{\Delta t F} \) can be included directly in the decomposition as part of the pre-computation step. This corresponds to using the \( E_I(t) \) version of the symmetric Dyson operators, and fully eliminates the additional multiplications for frame handling at each step.
C.3 Functional form of Direct CX pulses

Denoting \( A \) the pulse amplitude, \( T \) the total gate time, \( r \) the risetime, and \( \sigma \) the width of the shape during the risetime, the envelope functions for the Direct CX gate are as follows. First, denote the functions:

\[
C(r) = e^{-\frac{r^2}{2}} 
\]

\[
D(r, \sigma) = \sigma \left( \sqrt{2\pi} \text{erf} \left( \frac{r}{\sqrt{2}} \right) - 2rC(r) \right). 
\]

The symmetric target and control drive envelopes are parameterized by

\[
f(t, A, \sigma, r, T) = \begin{cases} 
A \left( e^{-\frac{1}{2} \left( \frac{t-r\sigma}{\sigma} \right)^2} - C(r) \right) / D(r, \sigma) & t < r\sigma \\
A(1 - C(r))/D(r, \sigma) & r\sigma \leq t \leq T - r\sigma \\
A \left( e^{-\frac{1}{2} \left( \frac{T-t-r\sigma}{\sigma} \right)^2} - C(r) \right) / D(r, \sigma) & T - r\sigma \leq t 
\end{cases}.
\]

and the anti-symmetric target drive is parameterized by

\[
g(t, A, \sigma, r, T) = \begin{cases} 
f(t, A, \sigma, r, T/2) & t < T/2 \\
-f(t - T/2, A, \sigma, r, T/2) & T/2 \leq t 
\end{cases}.
\]

C.4 Speed comparison supplement

The plot showing the benefits of parallelization of \texttt{odeint} on GPU via vectorization is shown in Figure 11. The ratio of the total simulation time over the number of simulations, when all simulations are run simultaneously via vectorization, is plotted as a function of the number of simulations. The curve decreases steadily until it becomes nearly flat over 6000 simulations. This indicates that the time to simulate scales sub-linearly until about 6000 simulations, and then scales roughly linearly beyond this point. We interpret this as a sign that parallelization on GPU using \texttt{odeint} for our problem is beneficial until 6000, and beyond this there is no additional speed gain provided by vectorization. This motivates the choice of 7000 inputs to use for speed comparisons on GPU.

Full perturbative solver GPU performance data is given in Figures 12 and 13. The corresponding data run on CPU are given in Figures 14 and 15. For the CPU data we reduce the number of random inputs to 100. As we are only using a single core, there are no concerns about ensuring we have enough inputs to “saturate” parallelization of the device. Figure 16 presents the CPU speedup data. In the CPU case, we observe that for high accuracies, the perturbative solvers are actually slower than \texttt{odeint}. For medium range accuracies, we observe a more modest speedup than in the GPU case, with up to 7\times speedup for gradient computation. Despite the presence of some speedup, realistically this implementation of the perturbative solvers will not be useful on CPU, as the number of instances that can be run in parallel on multiple cores will be limited.
Figure 11: For `odeint` with relative and absolute tolerances set to $10^{-10}$, the ratio of simulation time over number of parallel simulations is plotted, as a function of number of parallel simulations. For a given number of simulations, the input parameters are randomly generated (using the same random generator seed), and are simulated simultaneously via vectorization on an Nvidia A100 GPU with 80GB of memory. Above 6000 simulations, the curve becomes nearly constant, indicating that the total simulation time is becoming a linear function of the number of simulations, which indicates that the parallelization capabilities of the GPU has become saturated.

Due to memory requirements. By contrast, standard ODE solvers have minimal memory requirements, and therefore can scale to be run in parallel on many more CPU cores, eliminating any practical advantage of the perturbative solvers. As described in the main text, the memory requirements could be alleviated by implementing the core loop of the perturbative solvers in a serial way. Future versions of these solvers including this feature could lead to better comparisons on CPU.
Figure 12: Total Runtime v.s. Average Distance comparison on GPU for (A) Dyson-based perturbative solvers and (B) Magnus-based perturbative solvers. For both Dyson and Magnus solvers, the label \((d, n)\) denotes the configuration parameters used: a \(d\)-order Chebyshev approximation of the signals, and an \(n\)th order perturbative expansion. For a given solution \(U\), distance is computed via the metric \(\|U - V\|_2/\sqrt{d}\), where \(V\) is a benchmark solution computed using \texttt{odeint}\ at the smallest possible tolerance values. Average distance is the arithmetic mean of these values. For each configuration, the data points correspond to various numbers of time-steps \(M = 10^3, 2 \times 10^4, 3 \times 10^4, 4 \times 10^4, \text{ or } 5 \times 10^4\). The ODE solver was vectorized over all 7000 inputs.
Figure 13: Total Runtime v.s. Average Distance comparison for computing gradients on GPU for (A) Dyson-based perturbative solvers and (B) Magnus-based perturbative solvers. See the caption for Figure 12 for further details on reading this plot.
Figure 14: Total Runtime v.s. Average Distance comparison on CPU for (A) Dyson-based perturbative solvers and (B) Magnus-based perturbative solvers. This data was generated using 100 inputs run serially on a single-core CPU for all solvers. See the caption for Figure 12 for a description of the legend.
Figure 15: Total Runtime v.s. Average Distance comparison for computing gradients on CPU for (A) Dyson-based perturbative solvers and (B) Magnus-based perturbative solvers. This data was generated using 100 inputs run serially on a single-core CPU for all solvers. See the caption for Figure 12 for a description of the legend.
Figure 16: Speedup vs average distance for the data shown in Figures 14 and 15. For each ODE solver point with the fastest time for a given average distance, we find the fastest perturbative solver point whose average distance is no larger than that of the ODE solver. The “speedup” is the ratio of the ODE solver speed over the perturbative solver speed.

D Code examples

This appendix contains code blocks implemented with Qiskit Dynamics version 0.3.0, demonstrating how to perform computations described in the main text. See the API documentation for the latest usage details.

D.1 Constructing unitary approximations with the Magnus expansion

The following code block computes the unitary approximation of Section 4.2 using the `solve_lmde_perturbation` function. The pre-defined variables are:

- `perturb_list`: A list of callable functions for evaluating each perturbation.
- `T`: The time to integrate.
- `magnus_order`: The order of the Magnus expansion to compute.
- `generator`: A callable function computing the unperturbed interaction frame generator, in this case the function evaluating $-iH_0(t)$.
- `c`: The perturbation coefficient values.

Additional package-specific information is supplied in the comments.
# compute unperturbed unitary and Magnus expansion
# 'jax_odeint' is an integration method available in Qiskit Dynamics
results = solve_lmde_perturbation(
    perturbations=perturb_list,
    t_span=[0, T],
    expansion_method='magnus',
    expansion_order=magnus_order,
    generator=generator,
    integration_method='jax_odeint',
    rtol=1e-12,
    atol=1e-12
)

# Extract unperturbed unitary from the results object
U = results.y[-1]

# construct ArrayPolynomial for Magnus expansion
term_labels = results.perturbation_results.expansion_labels
perturbation_terms = results.perturbation_results.expansion_terms
magnus_expansion = ArrayPolynomial(
    array_coefficients=perturbation_terms,
    monomial_labels=term_labels,
)

# compute approximate unitary for a given value of c
# jexpm is the JAX matrix exponentiation routine
U_approx = U @ jexpm(magnus_expansion(c).data)

The final line in the above code can be called repeatedly for different values of c.

D.2 Robustness objective construction

We walk through the computation of the $h_I(b)$ matrices described in Section 4.3. Let $\text{mp}$ denote the `ArrayPolynomial` instance representing computed Magnus expansion truncation $\Omega(c, b)$, as constructed in the preceding section.

First, from $\Omega(c, b)$, we construct $\Omega(c, b)P - \frac{\text{Tr}(\Omega(c, b)P)}{2} P$. This can be done by interacting with an `ArrayPolynomial` using the same syntax as a standard `numpy` array:

# project onto subspace by restricting to first two columns
mp = mp[:, 0:2]

# remove part of mp proportional to projection
mp_nontrivial = (
mp = mp.trace() * np.eye(5, 2, dtype=complex) / 2

In the above mp_nontrivial is a ArrayPolynomial instance representing the array-valued polynomial given by $\Omega(c,b)P - \frac{Tr(\Omega(c,b)P)}{2}P$.

Finally, we construct an ArrayPolynomial representing $h(c,b)$ (for fixed $b$) by applying the definition of the Frobenius norm to mp_nontrivial, but when doing so, we only keep the terms $h_I(b)$ for which $\int_{c \in D} dc \rho(c) c_I \neq 0$. Let multiset_list denote the list of Multiset instances representing the $I$ corresponding to the non-zero moments, and let nonzero_moments denote the computed non-zero moments with the same ordering as multiset_list. We compute $g(b)$ as follows (with the variable robustness_integral corresponding to $g(b)$):

```python
# compute inner product of mp_nontrivial with itself
# keeping only terms corresponding to nonzero moments
mp_norm = mp_nontrivial.conj().mul(mp_nontrivial, monomial_filter=lambda x: x in nonzero_multisets).real.sum()

# compute integral by taking inner product of
# nonzero moments and the $h_I$ coefficients
robustness_integral = np.dot(mp_norm.array_coefficients, nonzero_moments[1:].data)
```

D.3 Perturbative solver API

The classes DysonSolver and MagnusSolver implement the perturbative solvers described in Section 5. Both classes are instantiated with the following signature:

```python
solver = PerturbativeSolver(
    operators: List[Operator],
    rotating_frame: Union[Array, Operator, RotatingFrame, None],
    dt: float,
    carrier_freqs: Array,
    chebyshev_orders: List[int],
    expansion_order: Optional[int] = None,
    expansion_labels: Optional[List[Multiset]] = None,
    integration_method: Optional[str] = None,
    include_imag: Optional[List[bool]] = None,
    **kwargs
)
```

where:
• **operators** is a list of the operators $A_j$.

• **rotating frame** is the frame operator $F$.

• **$\Delta t$** is the step size $\Delta t$.

• **carrier_freqs** is the list of analog carrier frequencies relative to which the expansion is computed. Note that these must be specified in frequency units, rather than angular frequency, as above.  

• **chebyshev_orders** is a list of integers indicating the number of Chebyshev terms to include when approximating each envelope, as in Equation (55). A different value must be specified for each signal, with 0 indicating a constant approximation.

• **expansion_order** and **expansion_labels** indicate the terms in the expansion to use when approximating the solution over each interval, with all terms up to **expansion_order** being used, and any additional terms in **expansion_labels** being included.

• **integration_method** is the ODE method used to compute the perturbation terms; any method available via qiskit.dynamics.solve_ode can be used.

• **include_imag** is a list of booleans indicating whether to keep, on a signal by signal basis, the second term in the decomposition in Equation (57).

• **kwargs** are any additional keyword arguments to pass to qiskit.dynamics.solve_ode when computing the expansion.

The pre-computation of the expansion occurs at object instantiation. Once instantiated, the solver can be run on different envelopes by calling the **solve** method:

\[
yf = \text{solver.solve}(t0, n\text{steps}, y0, signals)
\]

where

• **t0** and **n_steps** specify the interval to solve over: \([t0, t0 + n\text{steps} \times \Delta t]\).

• **y0** is the initial state of the linear matrix differential equation.

• **signals** is a list of qiskit.dynamics.Signal objects representing the time-dependent signals appearing in the decomposition of the generator. The carrier frequencies of these signals will be shifted to the reference carrier frequencies used at instantiation, with any leftover frequency absorbed into the envelope.

---

14. I.e. a frequency $\nu_j$ specified in this list corresponds to $\omega_j = 2\pi\nu_j$ in the mathematical presentation of the solver.

15. This term may be dropped if $\omega_j = 0$, or if there is some guarantee that $\text{Im}[f_{j,m}e^{i\omega_j t_0}] = 0$ over each interval startpoint. We note that, even if in the model there is a non-trivial carrier frequency for a given signal, it is possible to set $\omega_j = 0$ and absorb the complex oscillator into the definition of the envelope. This will come at a potential cost of requiring smaller $\Delta t$ for accuracy, but at the benefit of fewer terms in the expansion.
The method returns the final state $y_f$.

While their usage is technical, `DysonSolver` and `MagnusSolver` enable usage of these methods while only requiring a mathematical description of the problem. Internally, they make use of the `solve_lmde_perturbation` function and `ArrayPolynomial` class.

References

[1] F. J. Dyson. The Radiation Theories of Tomonaga, Schwinger, and Feynman. *Physical Review*, 75(3):486–502, February 1949.

[2] W. Magnus. On the exponential solution of differential equations for a linear operator. *Communications on Pure and Applied Mathematics*, 7(4):649–673, November 1954.

[3] S. Blanes, F. Casas, J. A. Oteo, and J. Ros. The Magnus expansion and some of its applications. *Physics Reports*, 470(5–6):151–238, January 2009.

[4] C. E. Soliverez. General theory of effective Hamiltonians. *Physical Review A*, 24(1):4–9, 1981. Publisher: American Physical Society.

[5] D. F. James and J. Jerke. Effective Hamiltonian theory and its applications in quantum information. *Canadian Journal of Physics*, 2011. Publisher: NRC Research Press Ottawa, Canada.

[6] S. Bravyi, D. P. DiVincenzo, and D. Loss. Schrieffer–Wolff transformation for quantum many-body systems. *Annals of Physics*, 326(10):2793–2826, 2011.

[7] D. Zeuch, F. Hassler, J. J. Slim, and D. P. DiVincenzo. Exact rotating wave approximation. *Annals of Physics*, 423:168327, December 2020.

[8] J. M. Gambetta, F. Motzoi, S. T. Merkel, and F. K. Wilhelm. Analytic control methods for high-fidelity unitary operations in a weakly nonlinear oscillator. *Physical Review A*, 83(1):012308, 2011. Publisher: American Physical Society.

[9] E. Magesan and J. M. Gambetta. Effective Hamiltonian models of the cross-resonance gate. *Physical Review A*, 101(5):052308, 2020. Publisher: American Physical Society.

[10] M. Malekakhlagh, E. Magesan, and D. C. McKay. First-principles analysis of cross-resonance gate operation. *Physical Review A*, 102(4):042605, 2020. Publisher: American Physical Society.

[11] U. Haeberlen and J. S. Waugh. Coherent Averaging Effects in Magnetic Resonance. *Physical Review*, 175:453–467, 1968.

[12] J. S. Waugh, L. M. Huber, and U. Haeberlen. Approach to High-Resolution nmr in solids. *Physical Review Letters*, 20:180–182, 1968.
[13] P. Mansfield. Symmetrized pulse sequences in high resolution NMR in solids. *Journal of Physics C: Solid State Physics*, 4(11):1444, 1971.

[14] W.-K. Rhim, D. D. Elleman, and R. W. Vaughan. Analysis of multiple pulse NMR in solids. *The Journal of Chemical Physics*, 59(7):3740–3749, 1973.

[15] M. Mehring. *Principles of High Resolution NMR in Solids*. Springer-Verlag Berlin Heidelberg, 1983.

[16] K. Takegoshi and C. A. McDowell. A “magic echo” pulse sequence for the high-resolution NMR spectra of abundant spins in solids. *Chemical Physics Letters*, 116(2):100 – 104, 1985.

[17] M. H. Levitt. Composite pulses. *Progress in Nuclear Magnetic Resonance Spectroscopy*, 18(2):61 – 122, 1986.

[18] D. G. Cory. A new multiple-pulse cycle for homonuclear dipolar decoupling. *Journal of Magnetic Resonance (1969)*, 94(3):526 – 534, 1991.

[19] D. G. Cory, J. B. Miller, R. Turner, and A. N. Garroway. Multiple-pulse methods of 1H N.M.R. imaging of solids: second-averaging. *Molecular Physics*, 70(2):331–345, 1990.

[20] L. Viola and S. Lloyd. Dynamical suppression of decoherence in two-state quantum systems. *Physical Review A*, 58:2733–2744, 1998.

[21] L. Viola, S. Lloyd, and E. Knill. Universal Control of Decoupled Quantum Systems. *Physical Review Letters*, 83:4888–4891, 1999.

[22] K. Khodjasteh and D. A. Lidar. Fault-Tolerant Quantum Dynamical Decoupling. *Physical Review Letters*, 95:180501, 2005.

[23] K. Khodjasteh and L. Viola. Dynamically Error-Corrected Gates for Universal Quantum Computation. *Physical Review Letters*, 102:080501, 2009.

[24] K. Khodjasteh, D. A. Lidar, and L. Viola. Arbitrarily Accurate Dynamical Control in Open Quantum Systems. *Physical Review Letters*, 104:090501, 2010.

[25] T. J. Green, J. Sastrawan, H. Uys, and M. J. Biercuk. Arbitrary quantum control of qubits in the presence of universal noise. *New Journal of Physics*, 15(9):095004, 2013.

[26] G. A. Paz-Silva and L. Viola. General Transfer-Function Approach to Noise Filtering in Open-Loop Quantum Control. *Physical Review Letters*, 113:250501, 2014.

[27] H. Ribeiro, A. Baksic, and A. A. Clerk. Systematic Magnus-Based Approach for Suppressing Leakage and Nonadiabatic Errors in Quantum Dynamics. *Physical Review X*, 7(1):011021, February 2017. Publisher: American Physical Society.

52
[28] W. A. B. Evans and J. G. Powles. A time-dependent Dyson expansion - the nuclear resonance signal in a rotating single crystal. *Proceedings of the Physical Society*, 92(4):1046–1054, 1967.

[29] H. Haas, D. Puzzuoli, F. Zhang, and D. G Cory. Engineering effective Hamiltonians. *New Journal of Physics*, 21(10):103011, October 2019.

[30] S. Tabatabaei, H. Haas, W. Rose, B. Yager, M. Piscitelli, P. Sahafi, A. Jordan, P. J. Poole, D. Dalacu, and R. Budakian. Numerical Engineering of Robust Adiabatic Operations. *Physical Review Applied*, 15(4):044043, April 2021. Publisher: American Physical Society.

[31] L. Cywiński, R. M. Lutchyn, C. P. Nave, and S. Das Sarma. How to enhance dephasing time in superconducting qubits. *Physical Review B*, 77(17):174509, 2008. Publisher: American Physical Society.

[32] T. J. Green, J. Sastrawan, H. Uys, and M. J. Biercuk. Arbitrary quantum control of qubits in the presence of universal noise. *New Journal of Physics*, 15(9):095004, 2013.

[33] T. Hangleiter, P. Cerfontaine, and H. Bluhm. Filter-function formalism and software package to compute quantum processes of gate sequences for classical non-Markovian noise. *Physical Review Research*, 3(4):043047, 2021. Publisher: American Physical Society.

[34] H. Ball, M. Biercuk, A. Carvalho, J. Chen, M. R. Hush, L. A. de Castro, L. Li, P. J. Liebermann, H. Slatyer, C. Edmunds, V. Frey, C. Hempel, and A. Milne. Software tools for quantum control: Improving quantum computer performance through noise and error suppression. *Quantum Science and Technology*, 2021.

[35] T. Hangleiter, J. D. Teske, and H. Bluhm. qutech/filter_functions: Release 1.1.2, May 2022.

[36] R. Shillito, J. A. Gross, A. Di Paolo, É. Genois, and A. Blais. Fast and differentiable simulation of driven quantum systems. *Physical Review Research*, 3(3):033266, September 2021. Publisher: American Physical Society.

[37] A. Kalev and I. Hen. An integral-free representation of the Dyson series using divided differences. *arXiv:2010.09888*, October 2020.

[38] Multi-index notation, September 2021. URL:https://en.wikipedia.org/wiki/Multi-index_notation, Page Version ID: 1046377330.

[39] Path-ordering, October 2022. URL:https://en.wikipedia.org/wiki/Path-ordering, Page Version ID: 1116280769.

[40] C. Van Loan. Computing integrals involving the matrix exponential. *IEEE Transactions on Automatic Control*, 23(3):395–404, 1978.
[41] F. Carbonell, J. C. Jimenez, and L. M. Pedroso. Computing multiple integrals involving matrix exponentials. *Journal of Computational and Applied Mathematics*, 213(1):300–305, 2008.

[42] D. L. Goodwin and I. Kuprov. Auxiliary matrix formalism for interaction representation transformations, optimal control, and spin relaxation theories. *The Journal of Chemical Physics*, 143(8):084113, 2015.

[43] S. Machnes, E. Assémat, D. Tannor, and F. K. Wilhelm. Tunable, Flexible, and Efficient Optimization of Control Pulses for Practical Qubits. *Physical Review Letters*, 120:150401, 2018.

[44] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005.

[45] C. Rackauckas, Y. Ma, V. Dixit, X. Guo, M. Innes, J. Revels, J. Nyberg, and V. Ivaturi. A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions. *arXiv:1812.01892 [cs]*, 2018.

[46] D. P. Burum. Magnus expansion generator. *Physical Review B*, 24(7):3684–3692, October 1981. Publisher: American Physical Society.

[47] W. R. Salzman. An alternative to the magnus expansion in time-dependent perturbation theory. *The Journal of Chemical Physics*, 82(2):822–826, January 1985. Publisher: American Institute of Physics.

[48] A. Arnal, F. Casas, and C. Chiralt. A general formula for the Magnus expansion in terms of iterated integrals of right-nested commutators. *Journal of Physics Communications*, 2(3):035024, March 2018. Publisher: IOP Publishing.

[49] J. Silverman. Number of polynomial terms for certain degree and certain number of variables. MathOverflow. URL:https://mathoverflow.net/q/225963 (version: 2015-12-12).

[50] D. J. Egger, H. Landa, A. Parr, D. Puzzuoli, B. Rosand, R. K. Rupesh, M. Treinish, and C. J. Wood. Qiskit Dynamics, 2021.

[51] multiset. URL:https://pypi.org/project/multiset/.

[52] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[53] D. Puzzuoli, S. Lin, M. Malekakhlagh, E. Pritchett, B. Rosand, and C. J. Wood. Supplemental code repository, 2022. URL:https://github.com/DanPuzzuoli/multivariable_dyson_magnus.
[54] A. Al-Mohy and N. Higham. A New Scaling and Squaring Algorithm for the Matrix Exponential. *SIAM Journal on Matrix Analysis and Applications*, 31(3):970–989, August 2009.

[55] Lawrence F. Shampine. Some practical Runge-Kutta formulas. *Mathematics of Computation*, 46(173):135–150, 1986.

[56] J. R. Johansson, P. D. Nation, and Franco Nori. QuTiP 2: A Python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 184(4):1234–1240, 2013.

[57] P. Jurcevic, A. Javadi-Abhari, L. S. Bishop, I. Lauer, D. F. Bogorin, M. Brink, L. Capelluto, O. Günlük, T. Itoko, N. Kanazawa, A. Kandala, G. A. Keefe, K. Krsulich, W. Landers, E. P. Lewandowski, D. T. McClure, G. Nannicini, A. Narasgond, H. M. Nayfeh, E. Pritchett, M. B. Rothwell, S. Srinivasan, N. Sundaresan, C. Wang, K. X. Wei, C. J. Wood, J.-B. Yau, E. J. Zhang, O. E. Dial, J. M. Chow, and J. M. Gambetta. Demonstration of quantum volume 64 on a superconducting quantum computing system. *Quantum Science and Technology*, 6(2):025020, 2021. Publisher: IOP Publishing.