The UniDyn *Mathematica* package: Self-derived unitary operator rotations in quantum mechanics

Aritro Sinha Roy[1], Garett M. Leskowitz[2], and John A. Marohn*[1]

[1]Dept. of Chemistry and Chemical Biology
Cornell University
Baker Laboratory
Ithaca, NY, USA 14851-1301
[2]Application Chemistry
Nanalysis Corp.
4-4500 5th Street NE
Calgary, AB, Canada T2E 7C3

January 6, 2026

Abstract

We have developed *Mathematica* algorithms for symbolically calculating unitary transformations of quantum-mechanical operators. These algorithms obtain closed-form analytical results, do not rely on a matrix representation of operators, and are applicable to both bounded systems like coupled spins and unbounded systems like harmonic oscillators. The unitary transformations are *self derived* from the operators' underlying commutation relations and can be carried out in a *basis free* way. Example calculations are presented involving magnetic resonance and quantum optics.

* jam99@cornell.edu

Contents

1 Introduction

Given a time-independent Hamiltonian $\mathcal{H}$ and an initial density operator $\rho(0)$ evolving in the Schrödinger represenation, the `Evolver` algorithms described below computes the density operator at time t by implementing the following unitary rotation

$$\rho(t) = e^{-i\mathcal{H}t}\ \rho(0)\ e^{+i\mathcal{H}t} \equiv \texttt{Evolver}[\mathcal{H}, t, \rho(0)] \tag{1}$$

The same algorithm can be used to compute the time dependence of an operator evolving in the Heinsenberg representation by replacing t with $-t$ and $\rho(0)$ with the operator of interest.

Consider two representative examples. In the Schrödinger representation, the time evolution of transverse spin magnetization in a longitudinal magnetic field is described by the rotation

$$e^{-i\,\omega t\,I_z}\ I_x\ e^{+i\,\omega t\,I_z} = I_x\ \sin(\omega t) + I_y\ \cos(\omega t) \tag{2}$$

where ω is the Larmor frequency and I_x, I_y, and I_z are the spin angular-momentum operators. In the Heisenberg representation, the harmonic oscillator's creation operator $a^\dagger$ evolves in time under the harmonic oscillator's Hamiltonian as follows:

$$e^{+i\omega t\frac{1}{2}(a^\dagger a + a\,a^\dagger)} a^\dagger e^{-i\omega t\frac{1}{2}(a^\dagger a + a\,a^\dagger)} = a^\dagger e^{+i\omega t} \tag{3}$$

where a is the annihilation operator and ω is the harmonic-oscillator's resonance frequency. Knowing the commutation relations among the operators comprising $\mathcal{H}$ and $\rho(0)$, it is often possible to obtain a closed-form algebraic solution to Eq. 1. The `Evolver` algorithms obtain closed-form solutions to Eq. 1 by extending an approach introduced by Slichter in Section 2.3 of his *Principles of Magnetic Resonance* text [1]. In this approach, one infers the differential equation that $\rho(t)$ satisfies. The only input to the algorithm is the commutation relations for the relevant operators.

2 Prior work

Numerous software programs exist for computing eq. 1 dynamics numerically. In order to gain physical understanding, researchers want to compute the dynamics symbolically, using computer algebra programs. This effort has been motivated by applications in magnetic resonance [2–10], quantum computing [11–13], electronic structure [14], vibrational dynamics [15], and quantum optics [16, 17]. Unitary rotations like eq. 1 occur in exact effective Hamiltonian theory [18, 19], the Baker-Campbell-Hausdorff formula [15, 20], the Zassenhaus product formula [15, 20, 21], the Hadamard lemma [15], and

simplifications of exponential operators involving raising and lower operators that are possible if the underlying potential has certain symmetries [15].

Most eq. 1 work to date, motivated by applications in magnetic resonance and quantum computing, has been semi-symbolic. Spins and few-level electron systems have a finite Hilbert space and can be represented by finite-dimensional matrices. In the semi-symbolic approaches, spin-angular momentum operators are expressed as matrices [5, 6, 11–13, 22] or irreducible spherical tensors [8, 9, 22, 23], while Hamiltonian and irradiation parameters remain symbolic. This semi-symbolic approach has led to powerful *Mathematica* packages like *MathNMR* [23] and *SpinDynamica* [22].

Vibrations and photons have an infinite (or near-infinite) Hilbert space and consequently the associated creation and annihilation operators do not have a matrix representation. Evolving creation and annihilation operators requires a purely symbolic approach.

There are fewer examples of purely symbolic approaches to computing eq. 1 dynamics. We know of only one algorithm that attempted to calculate commutators of spin- and harmonic-oscillator operators using a purely symbolic approach [14], and this algorithm stopped short of considering unitary evolution. In magnetic resonance, spin angular momenta have been described using non-commuting product operators, with evolution under pulsed irradiation and free evolution implemented either approximately, using a Baker-Campbell-Hausdorff (BCH) expansion [10], or exactly using operator substitution rules [2–4, 7]. Computer algebra programs have been used to carry out the multiplication and commutation of harmonic-oscillator (*i.e.* boson) operators symbolically [14, 16, 17], but eq. 1 dynamics was only computed approximately using a BCH expansion, with the unitary transformation represented by a truncated, finite series of nested commutators. Nguyen and workers described a promising procedure for resumming the resulting finite series to give a closed-form expression [17]. We find that this approach fails when the arguments of the associated sines, cosines, and exponentials involve non-trivial expressions such as fractions.

Slichter was interested in calculating the unitary evolution of angular momentum operators arising in spin-physics problems [1], like the rotation in eq. 2. In the following section we introduce the Slichter procedure by considering example cases. We show that his procedure works well for harmonic oscillator problems too, including problems involving the coupling of a harmonic oscillator with a two-level system, a minimum model for describing electron transfer in chemical reactions [24] and radiation-matter interactions in the strong-coupling limit [25–28].

It will become apparent that automating Slichter's procedure in a computer algebra program like *Mathematica* is possible but challenging. There are two challenges. The first is implementing non-commutative algebra in *Mathematica* in a useful way. The second is defining an operator inverse. We address these problems below. We then introduce two generalizations of the Slichter algorithm that are well suited for automation by a computer algebra program. We have implemented these algorithm as a *Mathematica* function, `Evolver`. This function evaluates Eq. 1 given an $\mathcal{H}$, an initial operator $\rho(0)$, and the commutation relations between the operators comprising $\mathcal{H}$ and $\rho(0)$.

3 Performing unitary rotations

To understand the `Evolver` algorithm it is essential to understand Slichter's approach to calculating the unitary evolution of an operator. To help the reader understand and appreciate Slichter's approach, let us briefly review two methods commonly used for solving Eq. 1. As an example, consider the case where $\mathcal{H} = \omega I_z$ and $\rho(0) = I_+ = I_x + i\, I_y$. In this case we want to compute

$$\rho(t) = e^{-i\,\omega t\, I_z}\, I_+\, e^{+i\,\omega t\, I_z}. \tag{4}$$

3.1 Method 1

One approach to computing $\rho(t)$ is to expand Eq. 4 in a Taylor series,

$$\rho(t) = \rho(0) + \rho^{(1)}(0)\, t + \frac{1}{2!}\rho^{(2)}(0)\, t^2 + \frac{1}{3!}\rho^{(3)}(0)\, t^3 + \cdots \tag{5}$$

The coefficients are obtained by differentiating ρ and setting $t \rightarrow 0$. Taking the first derivative,

$$\rho^{(1)}(t) = e^{-i\,\omega t\, I_z}\, (-i\,\omega\, I_z)\, I_+ e^{+i\,\omega t\, I_z} + e^{-i\,\omega t\, I_z}\, I_+ (+i\,\omega\, I_z)\, e^{+i\,\omega t\, I_z} \tag{6a}$$

$$= e^{-i\,\omega t\, I_z}\, (-i\,\omega [I_z, I_+])\, e^{+i\,\omega t\, I_z} \tag{6b}$$

$$\rho^{(1)}(0) = -i\,\omega\, I_+ \tag{6c}$$

where $\rho^{(n)}$ represents the n^{th} derivative with respect to time and where we have used $[I_z, I_+] = I_+$ to simplify the commutator. Taking the second derivative,

$$\rho^{(2)}(t) = e^{-i\,\omega t\, I_z}\, ((-i\,\omega)^2 [I_z, [I_z, I_+]])\, e^{+i\,\omega t\, I_z} \tag{7a}$$

$$\rho^{(2)}(0) = (-i\,\omega)^2\, I_+ \tag{7b}$$

By induction, we see that

$$\rho^{(n)}(0) = (-i\,\omega)^n\, I_+ \tag{8}$$

Substituting this finding into the Taylor expansion gives

$$\rho(t) = I_+ \left(1 + (-i\,\omega\,t) + \frac{1}{2!}(-i\,\omega t)^2 + \frac{1}{3!}(-i\,\omega t)^3 + \cdots\right) \tag{9}$$

We are now supposed to recognize the term in parenthesis as the Taylor series of $e^{-i\,\omega\,t}$. This insight enables us to resum the infinite series in the Taylor expansion to obtain the closed-form result

$$\rho(t) = I_+\, e^{-i\,\omega\,t}. \tag{10}$$

3.2 Method 2

A second approach to evaluating Eq. 4 is to expand the exponential using the Löwdin projection-operator theorem [29], equivalent to the Cayley-Hamilton theorem [30]. This theorem allows us to expand a function of an operator — I_z here — in terms involving the function evaluated at the operator's eigenvalues times an operator that project's onto the eigenvalue's subspace. Taking the total spin angular momentum to be $I = 1/2$ for simplicity, the relevant eigenvalues are $+1/2$ and $-1/2$ and the relevant projection operators are $\mathcal{P}_{1/2} = |\alpha\rangle\langle\alpha|$ and $\mathcal{P}_{-1/2} = |\beta\rangle\langle\beta|$. Applying Löwdin's theorem,

$$e^{-i\,\omega t\,I_z} = e^{-i\,\omega t/2}\,|\alpha\rangle\langle\alpha| + e^{+i\,\omega t/2}\,|\beta\rangle\langle\beta|\,. \tag{11}$$

Substituting this result into Eq. 4 yields

$$\rho(t) = \left(e^{-i\,\omega t/2}\,|\alpha\rangle\langle\alpha| + e^{+i\,\omega t/2}\,|\beta\rangle\langle\beta|\right) I_+ \left(e^{+i\,\omega t/2}\,|\alpha\rangle\langle\alpha| + e^{-i\,\omega t/2}\,|\beta\rangle\langle\beta|\right)$$

Applying the relations $I_+\,|\alpha\rangle = 0$, $I_+\,|\beta\rangle = |\alpha\rangle$, $\langle\alpha|\alpha\rangle = 1$, $\langle\beta|\alpha\rangle = 0$, and $|\alpha\rangle\langle\beta| = I_+$, this expression simplifies to

$$\rho(t) = I_+\, e^{-i\,\omega\,t}. \tag{12}$$

While both Methods 1 and 2 yield closed-form solutions, each is hardly extensible. Method 1 requires resumming a Taylor series; this step would be difficult or impossible to automate for any evolution more complicated than the one above. Method 2 requires obtaining the eigenvalues of the Hamiltonian, by reducing it to matrix form and diagonalizing it. It is hard to see how to apply this procedure in an infinite-level system like the idealized harmonic oscillator. For spin problems where the number of levels is finite, Method 2 still has serious limitations. Diagonalizing $\mathcal{H}$ forces us to write down the matrix representation of $\mathcal{H}$ which in turn commits us to specifying the total angular momentum I of the spin we are interested in. An exponentiated matrix A of the form e^{iA} can be written, using the

Cayley-Hamilton theorem [30], as a polynomial function of A [31, 32]. If A is a spin angular-momentum operator, then e^{iA} can be written as a sum over four Pauli matrices for $I = 1/2$ and as a sum over eight Gell-Mann matrices for $I = 1$ [33–35]. In many problems, we would like to obtain a solution valid for a spin of *any* I, and most interesting spin problems involve Hamiltonians more complicated than a spin angular-momentum operator.

3.3 Method 3

Now consider Slichter's procedure [1].

Example 1 — Let us take another look at the derivative of ρ:

$$\dot{\rho}(t) = e^{-i\,\omega t\,I_z}\,(-i\,\omega[I_z, I_+])\,e^{+i\,\omega t\,I_z} \tag{13a}$$

$$= -i\,\omega\left(e^{-i\,\omega t\,I_z}\,I_+\,e^{+i\,\omega t\,I_z}\right) \tag{13b}$$

As before we have used $[I_z, I_+] = I_+$ to reduce the commutator in Eq. 13a. The key insight in the Slichter procedure is that the term in parenthesis in Eq. 13b is nothing more than the original time-dependent density operator, $\rho(t)$. This insight allows us to write Equation 13b as

$$\dot{\rho}(t) = -i\,\omega\,\rho(t) \tag{14}$$

In this differential equation, ω is a *number*, while $\rho(t)$ is an *operator*. The solution to this differential equation is

$$\rho(t) = \rho(0)\,e^{-i\,\omega t} = I_+\,e^{-i\,\omega t}, \tag{15}$$

which is easily verified by back substitution.

Example 2 — An analogous calculation arises in a harmonic oscillator problem where the Hamiltonian is $\mathcal{H} = \omega(a^\dagger a + 1/2)$ and the initial operator is $\rho(0) = a^\dagger$:

$$\rho(t) = e^{-i\omega t(a^\dagger a+1/2)}\,a^\dagger\,e^{+i\omega t(a^\dagger a+1/2)} \tag{16}$$

Using the same procedure and the commutation relation $[a^\dagger a, a^\dagger] = a^\dagger[a, a^\dagger] + [a^\dagger, a^\dagger]a = a^\dagger$, this equation reduces to

$$\rho(t) = a^\dagger e^{-i\,\omega t}. \tag{17}$$

These first two Slichter-procedure example cases have in common that the commutator of the Hamiltonian with the operator of interest is simply proportional to the operator. As a result of this underlying commutation

relation, the problem of calculating Eq. 1 in both cases has been reduced to the problem of solving a first-order differential equation, Eq. 14. In light of the two previous methods, the Slichter procedure is rather remarkable. It allows us to obtain a closed-form solution for Eq. 4 without resorting to Taylor series and without even requiring knowledge of the Hamiltonian's eigenvalues.

Example 3 — The Slichter procedure is readily applied to more complicated unitary-evolution problems. Consider the case where $\mathcal{H} = \omega I_z$ and $\rho(0) = I_x$. Then

$$\rho(t) = e^{-i\,\omega t\, I_z}\, I_x\, e^{+i\,\omega t\, I_z}. \tag{18}$$

Taking the time derivative we obtain

$$\dot{\rho}(t) = e^{-i\,\omega t\, I_z}\, (-i\,\omega[I_z, I_x])\, e^{+i\,\omega t\, I_z} \tag{19a}$$

$$= \omega \left(e^{-i\,\omega t\, I_z}\, I_y\, e^{+i\,\omega t\, I_z} \right) \tag{19b}$$

where we have used $[I_z, I_x] = i\, I_y$ to reduce the commutator in Eq. 19a. In contrast to the previous case, $\dot{\rho}(t)$ is not proportional to $\dot{\rho}$. Taking another time derivative we obtain

$$\ddot{\rho}(t) = \omega\, e^{-i\,\omega t\, I_z}\, (-i\,\omega[I_z, I_y])\, e^{+i\,\omega t\, I_z} \tag{20a}$$

$$= -\omega^2 \left(e^{-i\,\omega t\, I_z}\, I_x\, e^{+i\,\omega t\, I_z} \right) \tag{20b}$$

where we have used $[I_z, I_y] = -i\, I_x$ to reduce the commutator in Eq. 20a. The term in parenthesis in Eq. 20b is proportional to $\rho(t)$ and, consequently,

$$\ddot{\rho}(t) = -\omega^2\, \rho(t). \tag{21}$$

The solution to this second-order differential equation is

$$\rho(t) = \rho(0)\, \cos(\omega t) + \frac{\dot{\rho}(0)}{\omega}\, \sin(\omega t). \tag{22}$$

We are given that $\rho(0) = I_x$ and we see from Eq. 19b that $\dot{\rho}(0) = \omega\, I_y$. Plugging these initial conditions into the above equation we obtain

$$\rho(t) = I_x\, \cos(\omega t) + I_y\, \sin(\omega t) \tag{23}$$

as the solution to Eq. 18.

Based on our experience so far, developing an algorithm capable of self-deriving Eqs. 15, 16, and 23 would now seem to be a matter of calculating

$$\rho^{(1)} = U^\dagger\, [-i\mathcal{H}, \rho(0)]\, U \tag{24}$$

$$\rho^{(2)} = U^\dagger\, [-i\mathcal{H}, [-i\mathcal{H}, \rho(0)]]\, U = U^\dagger\, [-i\mathcal{H}, \rho^{(1)}]\, U \tag{25}$$

and continuing until we obtain an expression that is proportional to $\rho = U^\dagger \rho(0) U$. If n iterations are required, then $\rho(t)$ must satisfy an n^{th} order differential equation. If the equation's coefficients and initial conditions can be extracted from the available non-commutative expressions correctly, then the differential equation can be fed to *Mathematica* to solve. This is roughly the procedure we will use. Because it would have to handle solving a large number of possible differential equations, however, significant effort would be required to implement the procedure just outlined in an automated way. The next example highlights why this is so. This example will serve as our launching point for introducing a revised, simple, and general algorithm for implementing the Slichter procedure for evaluating unitary evolution in an automated way.

Example 4 — Consider a unitary evolution with $\mathcal{H} = \Delta\omega I_z + \omega_1 I_x$ and $\rho(0) = I_z$. This rotation is involved in calculating the evolution of the difference in populations of a two level system when near-resonant irradiation is applied. The two-level system could be, for example, a spin $I = 1/2$ particle like a proton spin in a magnetic resonance experiment or it could be the lowest two electronic energy levels of an atom in a quantum optics experiment. The unitary evolution we are interested in computing is

$$\rho(t) = e^{-i\,(\Delta\omega I_z + \omega_1 I_x)\,t}\, I_z\, e^{+i\,(\Delta\omega I_z + \omega_1 I_x)\,t} \equiv U^\dagger\, I_x\, U \tag{26}$$

Computing the first few derivatives, we find

$$\dot{\rho} = U^\dagger \left(-i[\Delta\omega I_z + \omega_1 I_x, I_z]\right) U \tag{27a}$$
$$= U^\dagger \big(-\omega_1 I_y \big) U \tag{27b}$$
$$\ddot{\rho} = U^\dagger \left(-i[\Delta\omega I_z + \omega_1 I_x, -\omega_1, I_y]\right) U \tag{27c}$$
$$= U^\dagger \big(\Delta\omega\, \omega_1 I_x - \omega_1^2 I_z \big) U \tag{27d}$$
$$\dddot{\rho} = U^\dagger \left(-i[\Delta\omega I_z + \omega_1 I_x, -\omega_1, \Delta\omega\, \omega_1 I_x - \omega_1^2 I_z]\right) U \tag{27e}$$
$$= U^\dagger \big(\omega_1 \left(\omega_1^2 + \Delta\omega^2 \right) I_y \big) U \tag{27f}$$

Neither $\dot{\rho}$ nor $\ddot{\rho}$ is proportional to ρ. We see, however, that $\dddot{\rho}$ is proportional to $\dot{\rho}$. Defining $\sigma \equiv \dot{\rho}$, we see that σ satisfies the following differential equation

$$\ddot{\sigma} = -(\Delta\omega^2 + \omega_1^2)\, \sigma \tag{28}$$

with initial conditions

$$\sigma(0) = \dot{\rho}(0) = -\omega_1 I_y \tag{29a}$$
$$\dot{\sigma}(0) = \ddot{\rho}(0) = \Delta\omega\, \omega_1 I_x - \omega_1^2 I_z \tag{29b}$$

The solution to Eq. 28 is

$$\sigma(t) = \sigma(0)\cos(\omega_{\text{eff}}t) + \frac{\dot{\sigma}(0)}{\omega_{\text{eff}}}\sin(\omega_{\text{eff}}t) \tag{30}$$

with

$$\omega_{\text{eff}} \equiv \sqrt{\Delta\omega^2 + \omega_1^2} \tag{31}$$

an effective evolution frequency. Plugging in initial conditions,

$$\sigma(t) = \frac{\Delta\omega\,\omega_1}{\omega_{\text{eff}}}I_x\sin(\omega_{\text{eff}}\,t) - \omega_1 I_y\cos(\omega_{\text{eff}}\,t) - \frac{\omega_1^2}{\omega_{\text{eff}}}I_z\sin(\omega_{\text{eff}}\,t) \tag{32}$$

To obtain an equation for $\rho(t)$ we need to integrate this equation for $\sigma(t)$, taking care to include a constant of integration. Noting that

$$\int \cos(\omega_{\text{eff}}\,t)\,dt = \frac{\sin(\omega_{\text{eff}}\,t)}{\omega_{\text{eff}}} \text{ and } \int \sin(\omega_{\text{eff}}\,t)\,dt = -\frac{\cos(\omega_{\text{eff}}\,t)}{\omega_{\text{eff}}},$$

the solution for $\rho(t)$ becomes

$$\rho(t) = -\frac{\Delta\omega\,\omega_1}{\omega_{\text{eff}}^2}I_x\sin(\omega_{\text{eff}}\,t) - \frac{\omega_1}{\omega_{\text{eff}}}I_y\sin(\omega_{\text{eff}}\,t) + \frac{\omega_1^2}{\omega_{\text{eff}}^2}I_z\cos(\omega_{\text{eff}}\,t) + c \tag{33}$$

The integration constant c is determined by requiring the above equation to satisfy $\rho(0) = I_z$, the initial condition. Solving for the integration constant,

$$c = \frac{\Delta\omega\,\omega_1}{\Delta\omega^2 + \omega_1^2}I_x + \frac{\Delta\omega^2}{\Delta\omega^2 + \omega_1^2}I_z \tag{34}$$

Plugging this integration constant into Eq. 33 we obtain the following solution for the density operator in Eq. 26:

$$\begin{aligned}\rho(t) = \left(\frac{\Delta\omega^2}{\Delta\omega^2 + \omega_1^2}I_z + \frac{\Delta\omega\,\omega_1}{\Delta\omega^2 + \omega_1^2}I_x\right) + \frac{\omega_1}{\sqrt{\Delta\omega^2 + \omega_1^2}}\sin\left(\sqrt{\Delta\omega^2 + \omega_1^2}\,t\right) \\ + \left(\frac{\omega_1^2}{\Delta\omega^2 + \omega_1^2}I_z - \frac{\Delta\omega\,\omega_1}{\Delta\omega^2 + \omega_1^2}I_x\right)\cos\left(\sqrt{\Delta\omega^2 + \omega_1^2}\,t\right).\end{aligned} \tag{35}$$

We see in this example an answer for $\rho(t)$ that is markedly more complicated that in the previous examples, the solution to a third order differential equation. To uncover this differential equation, is was necessary to compare subsequent derivatives of $\rho(t)$ not to $\rho(0)$ but to $\dot{\rho}(0)$ instead.

Considering in total the four examples of unitary evolution enumerated so far, it would seem that any algorithm we develop needs to consider the possibility that the density operator satisfies a first, second, or even third-order differential equation. Before continuing, there is another case to consider.

Example 5 — Consider the unitary rotation

$$x' = e^{-i\hbar^{-1}\,\delta x\,p}\;x\;e^{+i\hbar^{-1}\,\delta x\,p} \tag{36}$$

which we can obtain from

$$\rho(\tau) = e^{-i\hbar^{-1}\,\tau\,\delta x\,p}\;x\;e^{+i\hbar^{-1}\,\tau\,\delta x\,p} \tag{37}$$

by setting the unitless parameter τ equal to 1. Computing the first two derivatives of ρ with respect to τ we find

$$\dot{\rho} = e^{-i\hbar^{-1}\,\tau\,\delta x\,p}\left(-i\hbar^{-1}\,\delta x\,[p,x]\right)e^{+i\hbar^{-1}\,\tau\,\delta x\,p} \tag{38a}$$

$$= -\delta x \tag{38b}$$

$$\ddot{\rho} = 0 \tag{38c}$$

where we have used that $[p,x] = -i\,\hbar$. As in Examples 1 and 2, we have a first-order differential equation. Here the relevant differential equation is simply $\dot{\rho}(\tau) = -\delta x$, with initial condition $\rho(0) = x$. It follows that $\rho = x + \tau\,\delta x$ and

$$x' = x - \delta x. \tag{39}$$

In a similar manner, we can obtain

$$p' = e^{-i\hbar^{-1}\,\delta p\,x}\;p\;e^{+i\hbar^{-1}\,\delta p\,x} = p + \delta p \tag{40}$$

From these examples, we recognize $\exp\left(-i\hbar^{-1}\,\delta x\,p\right)$ as the unitary transformation inducing a translation and $\exp\left(-i\hbar^{-1}\,\delta p\,x\right)$ as the unitary transformation inducing a momentum kick.

3.4 Method 4

We now show by example that the single differential equation in each of these five cases can be reduced to a set of *coupled first-order* differential equations — a significant step towards making a simple automated, unitary evolution algorithm. We would expect the five cases to each still require a decision on the number of coupled of equations required to solve for ρ in each case. Surprisingly, no decision is needed. We will show that all the cases we have discussed so far can be handled by a set of four coupled differential equations. This is the essential new insight implemented here in the `Evolver1` algorithm.

To understand the new method, consider transforming the second-order differential equation in Eq. 21 into two coupled first-order equations. Let the two new variables be

$$x_1 = \rho \text{ and } x_2 = \dot{\rho}. \tag{41}$$

Taking the time derivative of each of these variables we obtain

$$\dot{x}_1 = \dot{\rho} = x_2, \text{ and} \tag{42a}$$
$$\dot{x}_1 = \ddot{\rho} = -\omega^2 \rho = -\omega^2 x_1. \tag{42b}$$

It is apparent that these two variables satisfy the following set of coupled first order equations

$$\frac{d}{dt}\begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0 & -\omega^2 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} x_2 \\ x_1 \end{bmatrix} \text{ with } \begin{bmatrix} x_2(0) \\ x_1(0) \end{bmatrix} = \begin{bmatrix} \omega\, I_y \\ I_x \end{bmatrix}. \tag{43}$$

Solving this set of coupled equations, we find $x_1(t) = \rho(t) = I_x \sin(\omega t) + I_y \cos(\omega t)$, the expected answer.

What if we did not know how many coupled equations were necessary to solve the problem? Let's consider what would happen if we guessed that three coupled equations were needed instead of two. Defining

$$x_1 = \rho \text{ and } x_2 = \dot{\rho} \text{ and } x_3 = \ddot{\rho}, \tag{44}$$

and taking the time derivative gives

$$\dot{x}_1 = \dot{\rho} = x_2, \tag{45a}$$
$$\dot{x}_2 = \ddot{\rho} = x_3, \text{ and} \tag{45b}$$
$$\dot{x}_3 = \dddot{\rho} \tag{45c}$$
$$= \omega^2 U^\dagger (-i\,\omega[I_z, I_x])\, U \tag{45d}$$
$$= -\omega^3 U^\dagger I_y U \tag{45e}$$
$$= -\omega^2 \dot{\rho} \tag{45f}$$
$$= -\omega^2 x_2. \tag{45g}$$

In writing $\dot{x}_3$ we have used the shorthand $U(t) \equiv \exp(i\,\omega\, t\, I_z)$ and employed Eq. 19b to simplify the result. The three variables satisfy the following set of coupled first-order differential equations

$$\frac{d}{dt}\begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0 & -\omega^2 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} \text{ with } \begin{bmatrix} x_3(0) \\ x_2(0) \\ x_1(0) \end{bmatrix} = \begin{bmatrix} -\omega^2 I_x \\ \omega\, I_y \\ I_x \end{bmatrix} \tag{46}$$

Solving this new set of three coupled equations gives $x_1(t) = \rho(t) = I_x \sin(\omega t) + I_y \cos(\omega t)$ — the *same answer* that was obtained by solving the set of two coupled equations, Eqs. 43. This finding suggests that we are at liberty to "overguess" the number of equations required to solve the problem.

Let us see how the new method would be applied to solve the Example 4 problem with $\mathcal{H} = \Delta\omega I_z + \omega_1 I_x$ and $\rho(0) = I_z$. The time derivatives up to third order may be summarized as

$$\begin{bmatrix} \rho^{(3)} \\ \rho^{(2)} \\ \rho^{(1)} \\ \rho^{(0)} \end{bmatrix} = U^\dagger \begin{bmatrix} -\Delta\omega(\Delta\omega^2 + \omega_1^2)\, I_y \\ -\Delta\omega\,(\Delta\omega\, I_x - \omega_1 I_z) \\ \Delta\omega\, I_y \\ I_x \end{bmatrix} U \tag{47}$$

with $\rho^{(n)}$ the n^{th} derivative of ρ with respect to time and

$$U \equiv \exp(i\,(\Delta\omega I_z + \omega_1 I_x)\,t). \tag{48}$$

We see that $\rho^{(3)} = -(\Delta\omega^2+\omega_1^2)\,\rho^{(1)}$, as we showed before. Defining $x_n = \rho^{(n-1)}$ for $n = 1$ through 4, we obtain the following set of four coupled equations describing the time evolution of the density operator:

$$\underbrace{\frac{d}{dt}\begin{bmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix}}_{\dot{\boldsymbol{\lambda}}} = \underbrace{\begin{bmatrix} 0 & -(\Delta\omega^2 + \omega_1^2) & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\boldsymbol{\Omega}} \underbrace{\begin{bmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix}}_{\boldsymbol{\lambda}} \tag{49}$$

with

$$\underbrace{\begin{bmatrix} x_4(0) \\ x_3(0) \\ x_2(0) \\ x_1(0) \end{bmatrix}}_{\boldsymbol{\lambda}_0} = \begin{bmatrix} -\Delta\omega(\Delta\omega^2 + \omega_1^2)\, I_y \\ -\Delta\omega\,(\Delta\omega\, I_x - \omega_1 I_z) \\ \Delta\omega\, I_y \\ I_x \end{bmatrix} \tag{50}$$

Solving these coupled equations, we obtain Eq. 35 for $x_1 = \rho(t)$. What about Example 5? In this case we have simply

$$\boldsymbol{\Omega} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \text{ and } \boldsymbol{\lambda}_0 = \begin{bmatrix} 0 \\ 0 \\ -\delta x \\ x \end{bmatrix} \tag{51}$$

Solving these equations, we recover Eq. 39, the expected result.

In the above equations we have defined $\boldsymbol{\lambda}$ as the vector of x_n's that we are interested in and $\boldsymbol{\Omega}$ as the matrix of coefficients coupling $\boldsymbol{\lambda}$ to $\dot{\boldsymbol{\lambda}}$. The coupled equations are summarized as

$$\frac{d\boldsymbol{\lambda}}{dt} - \boldsymbol{\Omega} \cdot \boldsymbol{\lambda} = 0 \tag{52}$$

with $\boldsymbol{\lambda}_0$ given.

The `Evolver1` algorithm proceeds as follows.

1. Evaluate the derivatives $\rho^{(1)}$ through $\rho^{(3)}$ by repeatedly applying computing $\rho^{(n+1)} = [-iH, \rho^{(n)}]$, starting from $\rho^{(0)} = \rho(0)$. Assign the initial-condition vector $\boldsymbol{\lambda}_0 = (\rho^{(3)}, \ldots, \rho^{(0)})$.
2. Fill in the lower 3 rows of $\boldsymbol{\Omega}$ by placing a 1 in entries one below the diagonal and 0 in all the other positions.
3. Look for an entry in the list $(\rho^{(2)}, \rho^{(1)}, \rho^{(0)})$ that is proportional to $\rho^{(3)}$, starting with the $\rho^{(2)}$ (entry $n = 3$) and working towards $\rho^{(0)}$ (entry $n = 1$). Call the entry where the match occurs n_{match}. Call the ratio between $\rho^{(3)}$ and the matching entry r.
4. Fill in the upper row of $\boldsymbol{\Omega}$ with zeros. Assign the value r to the entry in the n_{match} column of the upper row of $\boldsymbol{\Omega}$, counting from the right to the left. If no match can be found, declare the problem unsolvable.
5. Feed Eq. 52, including the initial condition, to *Mathematica* to solve. The time-dependent density operator is $\rho(t) = \lambda_1(t)$.

3.5 Method 5

An alternative approach is to identify, on a case-by-case basis, which differential equation $\rho(t)$ is governed by and write out the differential-equation solution by hand. This alternative approach constitutes the `Evolver2` algorithm. The algorithm proceeds by testing the following cases, stopping if the case is true. The cases are labelled 0, 1, 2, or 3, according the order of differential equation governing ρ, and are given the designation L for linear and E for exponential, depending on the kind of solution.

1. As above, evaluate the derivatives $\rho^{(1)}$ through $\rho^{(3)}$.
2. (Case 0) If $\rho^{(1)} = 0$ then the Hamiltonian commutes with the initial density operator and
$$\rho(t) = \rho(0) \tag{53}$$
3. (Case 1L) If $[\rho^{(1)}, H] = 0$ then
$$\rho(t) = \rho^{(0)} + \rho^{(1)} t \tag{54}$$

4. Compute $c_1 = \rho^{(1)}$ and the commutators

$$c_2 = [\rho^{(0)}, \rho^{(1)}] \tag{55a}$$
$$c_3 = [\rho^{(0)}, \rho^{(2)}] \tag{55b}$$
$$c_4 = [\rho^{(1)}, \rho^{(3)}] \tag{55c}$$

5. Compute the ratios

$$d_1 = \text{Inv}[\rho^{(0)}]\, \rho^{(1)} \tag{56a}$$
$$d_2 = \text{Inv}[\rho^{(0)}]\, \rho^{(2)} \tag{56b}$$
$$d_3 = \text{Inv}[\rho^{(1)}]\, \rho^{(3)} \tag{56c}$$

6. (Case 1E) If $c_2 = 0$ then

$$\rho(t) = e^{d_1 t} \rho^{(0)} \tag{57}$$

7. (Case 2E) If $c_3 = 0$ then define $\omega = \sqrt{-d_2}$ and compute

$$\rho(t) = \cos(\omega\, t)\, \rho^{(0)} + \frac{1}{\omega} \sin(\omega\, t)\, c_1 \tag{58}$$

8. (Case 3E) If $c_4 = 0$ then define $\omega = \sqrt{-d_3}$ and compute

$$\rho(t) = \rho^{(0)} + \frac{1}{\omega} \sin(\omega\, t)\, \rho^{(1)} + \frac{1}{\omega^2} (1 - \cos(\omega\, t))\ \rho^{(2)} \tag{59}$$

9. If none of these cases are satisfied, declare the problem unsolvable.

3.6 Code

To automate the steps in the `Evolver1` and `Evolver2` algorithms we require the ability to

1. perform non-commutative algebra,
2. evaluate commutators,
3. divide operators, and
4. determine which differential equation an evolved operator satisfies.

Mathematica has a native function for carrying out non-commutative multiplication, but this function has essentially no simplification rules associated with it. For example,

```
In[1] := a ** (2 b) // Simplify
Out[1] = a ** (2 b)
```

This limitation was resolved by Helton and co-workers, whose `NCALgebra` package [36] gives *Mathematica* the ability to manipulate non-commuting algebraic expressions. Their package allows the user to define variables as either commuting (e.g., ω) or non-commuting (e.g., I_+, I_x, $a^\dagger$, and so on). The `NCALgebra` package forms an excellent starting point for implementing non-commutative multiplication, and expanding and simplifying commutators. We find, however, that using the package adds significant computational overhead, increasing computation time over ten-fold.

To avoid this overhead, we created our own non-commutative multiplication and commutator functions, with accompanying simplification rules.

The resulting code consists of *Mathematica* files (`.m`) and notebooks (`.nb`). Each modular `m` file implements a desired function. Associated with each `m` file is a second `m` file containing *unit tests*. The code, at the time of writing, includes 160 unit tests. Code was written in the literate-programming style [37], with LaTeX-style text, equations, and tables interspersed with code in the form of commented-out text. The code and units tests are printed out, with typeset in-line commentary, in Appendix A.

To unit-test `Evolver1` and `Evolver2` algorithms, we computed the following rotating-frame unitary evolutions and checked the results against by-hand computations:

1. $e^{-i\omega t I_z} I_x e^{+i\omega t I_z}$, free evolution of transverse magnetization in a magnetic field
2. $e^{-i\omega t I_x} I_z e^{+i\omega t I_x}$, on-resonance nutation of longitudinal (i.e., thermal-equilibrium) magnetization in the rotating frame
3. $e^{-i\omega t I_z} I_+ e^{+i\omega t I_z}$, free evolution of complex transverse magnetization in a magnetic field
4. $e^{-idt I_z S_z} I_x e^{+idt I_z S_z}$, evolution of transverse magnetization under a scalar coupling
5. $e^{-i(\Delta I_z + \omega I_x)t} I_z e^{+i(\Delta I_z + \omega I_x)t}$, off-resonance nutation of longitudinal magnetization
6. $e^{-\omega t \frac{1}{2}(a^\dagger a + a\, a^\dagger)} a e^{+\omega t \frac{1}{2}(a^\dagger a + a\, a^\dagger)}$, evolution of the lowering (and raising) operator under the harmonic oscillator Hamiltonian
7. $e^{-\omega t \frac{1}{2}(a^\dagger a + a\, a^\dagger)} Q e^{+\omega t \frac{1}{2}(a^\dagger a + a\, a^\dagger)}$, evolution of the position (and momentum) operator under the harmonic oscillator Hamiltonian

8. $e^{-i\delta q\,P} Q e^{+i\delta q\,P}$, a displacement
9. $e^{-i\delta q\,Q} P e^{+i\delta q\,Q}$, a momentum kick

The off-resonance spin nutation, case 5, is an especially demanding test because it generates terms involving I_x, I_y, and I_z operators from solutions to a third-order differential equation.

For unitary rotation involving spin and harmonic-oscillator operators, `Evolver1` and `Evolver2` give identical answers. For the off-resonance Rabi nutation calculation, Example 5, the `Evolver2` algorithm is a factor of 25 faster than the `Evolver1` algorithm.

In the following section we present representative results. Detailed applications of the code are given in seven stand-alone *Mathematica* notebooks.

4 Representative results

4.1 Two spins

Define two spins, $\boldsymbol{I}$ and $\boldsymbol{S}$. The first spin has an unspecified total angular momentum while the second spin has total angular momentum $S = 1/2$. The `Evolver2` algorithm computes

$$e^{-i\,Jt\,I_z S_z}\; I_x\; e^{+i\,J\,I_z S_z} = I_x\,\cos\left(Jt/2\right) + 2I_y S_z\,\sin\left(Jt/2\right), \tag{60}$$

which is a non-trivial extension of the single-spin rotations encoded in the unit tests. For the evolution of S_x, `Evolver2` computes

$$e^{-i\,Jt\,I_z S_z}\; S_x\; e^{+i\,J\,I_z S_z} = S_x\,\cos\left(Jt\sqrt{I_z^2}\right) + \frac{2I_z S_y}{\sqrt{I_z^2}}\,\sin\left(Jt\sqrt{I_z^2}\right). \tag{61}$$

If we specify the spin angular momentum I then it is straightforward to write down the eigenvalues of I_z and expand the trigonometric operators in Eq. 61 using Lödwin projection-operator theorem [29].

4.2 Harmonic oscillator

Consider a harmonic oscillator with Hamiltonian

$$\mathcal{H}_0 = \omega_0\left(a^\dagger a + \frac{1}{2}\right) \tag{62}$$

and consider that we calculate oscillator dynamics in an interaction representation defined by $\mathcal{H}_0$. The application of an on-resonance force leads to a Hamiltonian

$$\mathcal{H}_1 = \frac{A}{\sqrt{2}}(e^{-i\phi}\,a^\dagger + e^{+i\phi}\,a) \tag{63}$$

where ϕ is the phase of the oscillating force. The `Evolver2` algorithm computes that the oscillator's position and momentum evolve under the action of $\mathcal{H}_1$ as follows:

$$e^{-i\mathcal{H}_1 t} Q e^{+i\mathcal{H}_1 t} = Q + At\sin\phi \tag{64a}$$

$$e^{-i\mathcal{H}_1 t} P e^{+i\mathcal{H}_1 t} = P + At\cos\phi. \tag{64b}$$

This example shows that the algorithm handles complex numbers correctly.

Modulating the spring constant of the oscillator at twice the oscillator's resonance frequency $2\omega_0$ gives rise a "squeezing" Hamiltonian

$$\mathcal{H}_2 = \frac{\Omega}{2}\left(e^{-i\phi}(a^\dagger)^2 + e^{+i\phi}a^2\right) \tag{65}$$

with ϕ the phase of the oscillating spring constant. Under the action of this Hamiltonian, the `Evolver2` algorithm computes that

$$e^{-i\mathcal{H}_2 t} a^\dagger e^{+i\mathcal{H}_2 t} = a^\dagger\cosh(\Omega t) - iae^{i\phi}\sinh(\Omega t). \tag{66}$$

When $\phi = \pi/2$,

$$\{Q, P\} \xrightarrow{\mathcal{H}_2} \{e^{-\Omega t}Q, e^{+\Omega t}P\} \tag{67}$$

we see that Q is attenuated while P is amplified. On the other hand, when $\phi = 3\pi/2$,

$$\{Q, P\} \xrightarrow{\mathcal{H}_2} \{e^{+\Omega t}Q, e^{-\Omega t}P\}, \tag{68}$$

and now Q is amplified while P is attenuated. For these two choices of phase, the expectation value of the product QP is conserved. This example shows that the `Evolver2` algorithm can capture the physics of degenerate parametric amplification [38, 39].

Consider the more complicated case of non-degenerate parametric amplification, when the spring constant is modulated off-resonance at frequency 2ω. To remove the time-dependence in the resulting Hamiltonian, we need an interaction representation defined by $\omega(a^\dagger a + 1/2)$. `Evolver2` computes that, in this interaction representation, $a^\dagger$ evolves as follows:

$$e^{-i(\Delta\omega\, a^\dagger a + \mathcal{H}_2)t} a^\dagger e^{-i(\Delta\omega\, a^\dagger a + \mathcal{H}_2)t}$$

$$= a^\dagger\cos(t\sqrt{\Delta\omega^2 - \Omega^2}) - \frac{i(\Delta\omega\, a^\dagger + \Omega\, e^{i\phi}\, a)\sin(t\sqrt{\Delta\omega^2 - \Omega^2})}{\sqrt{\Delta\omega^2 - \Omega^2}}. \tag{69}$$

with $\Delta\omega = \omega_0 - \omega$ a resonance offset. The Eq. 69 calculation is challenging because $[\Delta\omega\, a^\dagger a, \mathcal{H}_2] \neq 0$, and the resulting operator transformation is non-trivial. Equation 69 reduces to Eq. 66 in the on-resonance limit, $\Delta\omega \to$

0. In the limit of no parametric amplification, $\Omega \to 0$, Eq. 69 reduces to the expected $a^\dagger e^{-i\,\Delta\omega\,t}$. In the general, off-resonance case, Eq. 69 exhibits complicated dynamics — either oscillatory or exponential, depending on the relative size of $\Delta\omega$ and Ω.

4.3 Electron transfer

In Ref. 24, electron transfer between two sites (H and H^+ for example) the electronic Hamiltonian, vibrational Hamiltonian, and electronic-vibrational coupling is treated using a Holstein molecular crystal model. The electronic states are represented by an $I = 1/2$ spin system $\boldsymbol{I}$ and the vibrational states are represented by an harmonic oscillator with creation and annihilation operators $(a^\dagger, a)$. To remove the electronic-vibrational coupling, dynamics are calculated in an interaction representation defined by an operator (in our notation)

$$T = -2gI_z\left(a^\dagger - a\right). \tag{70}$$

The associated unitary transformation corresponds to an electronic-state-dependent translation. In this interaction representation, `Evolver2` correctly computes

$$e^{-i\,t\,T}\,I_\pm\,e^{+i\,t\,T} = e^{\pm 2g\left(a^\dagger - a\right)}\,I_\pm \tag{71}$$

This example shows that `Evolver2` can handle differential equations in which the coefficients are *operators*.

This example highlights the limitations of `Evolver1`, which fails to resolve Eq. 71. The first two derivatives in the I_+ case are

$$\rho^{(0)} = I_+ \tag{72}$$

$$\rho^{(1)} = 2g\left(I_+ a - I_+ a^\dagger\right) \tag{73}$$

and the `Divide` function employed in `Evolver1` is unable to see that $\rho^{(1)}/\rho^{(0)} = 2g\left(a^\dagger - a\right)$. The `Inv` function employed in `Evolver2` is able to implement this division correctly.

5 Conclusions

The `UniDyn` package offers a general approach to symbolically computing the unitary evolution of quantum-mechanical operators given only the operators' underlying commutation relations. The package includes code for manipulating and simplifying expressions involving a mixture of commuting and non-commuting symbols, commutators, and inverse operators. The Sec. 4 results show that the `Evolver` algorithms are applicable in cases well beyond the unit tests discussed in Sec. 4 and Appendix A. The `UniDyn`

package GitHub page includes seven *Mathematica* notebooks showing example `Evolver2` calculations involving one spin, two spins, the harmonic oscillator, quantum optics, and electron transfer.

One of us (JAM) has used the `UniDyn` package to teach both magnetic resonance, multidimensional spectroscopy, relaxation theory, and quantum optics in a graduate quantum mechanics course. The package significantly expands the range of phenomena that can be covered in a one-semester graduate course on time-dependent quantum mechanics.

Two of us (ASR and JAM) recently used `Evolver1` to compute multi-spin double-quantum coherence expressions for up to $N_{\text{spin}} = 9$ spin $I = 1/2$ particles [40]. Given that $N_{\text{spin}} = 9$ is near the limit of what can be simulated *numerically*, were were initially surprised to find that the evolution of so many spins could be treated *analytically*. We should consider, however, that the pulse sequences in Ref. 40 were relatively simple and that spin-spin couplings act in a pairwise fashion. These constraints limited the terms in the density operator to many fewer than would be included in a numerical calculation. Remarkably, `Evolver1` results could be averaged over orientations analytically and extrapolated to obtain analytical expressions for signal valid in the $N_{\text{spin}} \rightarrow \infty$ limit.

We shope that others will find the `UniDyn` package as useful as we have.

References

[1] C. P. Slichter, *Principles of Magnetic Resonance, 3rd Edition*, Springer, New York, 1990.

[2] J. W. Shriver, NMR product-operator calculations in *Mathematica*, *J. Magn. Res.*, **1991**, *94*, 612–616, URL https://doi.org/10.1016/0022-2364(91)90150-R.

[3] R. P. F. Kanters, B. W. Char, and A. W. Addison, A computer-algebra application for the description of NMR experiments using the product-operator formalism, *J. Magn. Reson. Ser. A*, **1993**, *101*, 23–29, URL https://doi.org/10.1006/jmra.1993.1003.

[4] P. Guntert, N. Schaefer, G. Otting, and K. Wuthrich, POMA: A complete *Mathematica* implementation of the NMR product-operator formalism, *J. Magn. Reson. Ser. A*, **1993**, *101*, 103–105, URL https://doi.org/10.1006/jmra.1993.1016.

[5] D. Isbister, M. S. Krishnan, and B. Sanctuary, Use of computer algebra for the study of quadrupole spin systems, *Molec. Phys.*, **1995**, *86*, 1517–1535, URL https://doi.org/10.1080/00268979500102891.

[6] I. Rodriguez and J. Ruiz-Cabello, Density matrix calculations in Mathematica, *Concepts Magn. Res.*, **2001**, *13*, 143–147, URL https://doi.org/10.1002/1099-0534(2001)13:2<143::AID-CMR1003>3.0.CO;2-4.

[7] P. Güntert, Symbolic NMR product operator calculations, *Int. J. Quantum Chem.*, **2006**, *106*, 344–350, URL https://doi.org/10.1002/qua.20754.

[8] C. K. Anand, A. D. Bain, and Z. Nie, Simulation of steady-state NMR of coupled systems using Liouville space and computer algebra methods, *J. Magn. Res.*, **2007**, *189*, 200–208, URL https://doi.org/10.1016/j.jmr.2007.09.012.

[9] I. Kuprov, N. Wagner-Rundell, and P. J. Hore, Bloch-Redfield-Wangsness theory engine implementation using symbolic processing software, *J. Magn. Res.*, **2007**, *184*, 196–206, URL https://doi.org/10.1016/j.jmr.2006.09.023.

[10] X. Filip and C. Filip, SD-CAS: Spin dynamics by computer algebra system, *J. Magn. Res.*, **2010**, *207*, 95–113, URL https://doi.org/10.1016/j.jmr.2010.08.014.

[11] T. Loke and J. B. Wang, An efficient quantum circuit analyser on qubits and qudits, *Comput. Phys. Commun.*, **2011**, *182*, 2285–2294, URL https://doi.org/10.1016/j.cpc.2011.06.001.

[12] Y. G. Chen and J. B. Wang, *Qcompiler*: Quantum compilation with the CSD method, *Comput. Phys. Commun.*, **2013**, *184*, 853–865, URL https://doi.org/10.1016/j.cpc.2012.10.019.

[13] T. Loke and J. B. Wang, *CUGatesDensity* — quantum circuit analyser extended to density matrices, *Comput. Phys. Commun.*, **2013**, *184*, 2834–2839, URL https://doi.org/10.1016/j.cpc.2013.07.007.

[14] R. Žitko, SNEG — Mathematica package for symbolic calculations with second-quantization-operator expressions, *Comput. Phys. Commun.*, **2011**, *182*, 2259–2264, URL https://doi.org/10.1016/j.cpc.2011.05.013.

[15] A. N. F. Aleixo and A. B. Balantekin, Exponential operators and the algebraic description of quantum confined systems, *J. Math. Phys.*, **2011**, *52*, 082106, URL https://doi.org/10.1063/1.3625627.

[16] V. N. Beskrovnyi, Applying Mathematica to the analytical solution of the nonlinear Heisenberg operator equations, *Comput. Phys. Commun.*, **1998**, *111*, 76–86, URL https://doi.org/10.1016/S0010-4655(98)00029-0.

[17] N.-A. Nguyen and T. T. Nguyen-Dang, Symbolic calculations of unitary transformations in quantum dynamics, *Comput. Phys. Commun.*, **1998**, *115*, 183–199, URL https://doi.org/10.1016/S0010-4655(98)00129-5.

[18] T. S. Untidt and N. C. Nielsen, Closed solution to the Baker-Campbell-Hausdorff problem: Exact effective Hamiltonian theory for analysis of nuclear-magnetic-resonance experiments, *Phys. Rev. E*, **2002**, *65*, 021108, URL https://doi.org/10.1103/PhysRevE.65.021108.

[19] D. Siminovitch, T. Untidt, and N. C. Nielsen, Exact effective Hamiltonian theory. II. Polynomial expansion of matrix functions and entangled unitary exponential operators, *J. Chem. Phys.*, **2004**, *120*, 51–66, URL https://doi.org/10.1063/1.1628216.

[20] M. Weyrauch and D. Scholz, Computing the Baker–Campbell–Hausdorff series and the Zassenhaus product, *Comput. Phys. Commun.*, **2009**, *180*, 1558–1565, URL https://doi.org/10.1016/j.cpc.2009.04.007.

[21] F. Casas, A. Murua, and M. Nadinic, Efficient computation of the Zassenhaus formula, *Comput. Phys. Commun.*, **2012**, *183*, 2386–2391, URL https://doi.org/10.1016/j.cpc.2012.06.006.

[22] C. Bengs and M. H. Levitt, SpinDynamica: Symbolic and numerical magnetic resonance in a mathematica environment, *Mag. Res. Chem.*, **2017**, *56*, 374–414, URL https://doi.org/10.1002/mrc.4642.

[23] A. Jerschow, MathNMR: Spin and spatial tensor manipulations in Mathematica, *J. Magn. Res.*, **2005**, *176*, 7–14, URL https://doi.org/10.1016/j.jmr.2005.05.005.

[24] K. V. Mikkelsen and M. A. Ratner, Electron tunneling in solid-state electron-transfer reactions, *Chem. Rev.*, **1987**, *87*, 113–153, URL https://doi.org/10.1021/cr00077a007.

[25] E. Jaynes and F. Cummings, Comparison of quantum and semiclassical radiation theories with application to the beam maser, *Proc. IEEE*, **1963**, *51*, 89–109, URL https://doi.org/10.1109/PROC.1963.1664.

[26] J. H. Eberly, N. B. Narozhny, and J. J. Sanchez-Mondragon, Periodic Spontaneous Collapse and Revival in a Simple Quantum Model, *Phys. Rev. Lett.*, **1980**, *44*, 1323–1326, URL https://doi.org/10.1103/PhysRevLett.44.1323.

[27] A. D. Greentree, J. Koch, and J. Larson, Fifty years of Jaynes–Cummings physics, *J. Phys. B: At. Mol. Opt. Phys.*, **2013**, *46*, 220201, URL https://doi.org/10.1088/0953-4075/46/22/220201.

[28] J. Larson, T. Mavrogordatos, S. Parkins, and A. Vidiella-Barranco, The Jaynes–Cummings model: 60 years and still counting, *J. Opt. Soc. Am. B*, **2024**, *41*, JCM1, URL https://doi.org/10.1364/JOSAB.536847.

[29] P.-O. Löwdin, Quantum theory of many-particle systems. III. Extension of the Hartree-Fock scheme to include degenerate systems and correlation effects,

Phys. Rev., **1955**, *97*, 1509–1520, URL https://doi.org/10.1103/PhysRev.97.1509.

[30] G. Strang, *Introduction to Linear Algebra*, Cambridge press, Wellesley, 5th edition ed., 2016.

[31] F. De Zela, Closed-Form Expressions for the Matrix Exponential, *Symmetry*, **2014**, *6*, 329–344, URL https://doi.org/10.3390/sym6020329.

[32] T. L. Curtright, D. B. Fairlie, and C. K. Zachos, A Compact Formula for Rotations as Spin Matrix Polynomials, *SIGMA*, **2014**, *10*, 084, URL https://doi.org/10.3842/SIGMA.2014.084.

[33] M. Gell-Mann, Symmetries of Baryons and Mesons, *Phys. Rev.*, **1962**, *125*, 1067–1084, URL https://doi.org/10.1103/PhysRev.125.1067.

[34] R. A. Bertlmann and P. Krammer, Bloch vectors for qudits, *J. Phys. A: Math. Theor.*, **2008**, *41*, 235303, URL https://doi.org/10.1088/1751-8113/41/23/235303.

[35] T. L. Curtright and C. K. Zachos, Elementary results for the fundamental representation of SU(3), *Rep. Math. Phys.*, **2015**, *76*, 401–404, URL https://doi.org/10.1016/S0034-4877(15)30040-9.

[36] J. W. Helton, M. de Oliveira, M. Stankus, and R. L. Miller, NCAlgebra — Version 5.0.6, 2015.

[37] D. E. Knuth, Literate Programming, *The Computer Journal*, **1984**, *27*, 97–111, URL https://doi.org/10.1093/comjnl/27.2.97.

[38] B. R. Mollow and R. J. Glauber, Quantum theory of parametric amplification. I, *Phys. Rev.*, **1967**, *160*, 1076–1096, URL https://doi.org/10.1103/PhysRev.160.1076.

[39] B. R. Mollow and R. J. Glauber, Quantum theory of parametric amplification. II, *Phys. Rev.*, **1967**, *160*, 1097–1108, URL https://doi.org/10.1103/PhysRev.160.1097.

[40] A. Sinha Roy, J. Marohn, and J. Freed, An analysis of double-quantum coherence ESR in an N-spin system: Analytical expressions and predictions, *J. Chem. Phys.*, **2024**, *160*, 134105, URL https://doi.org/10.1063/5.0200054.

A Code and associated unit tests

In this section we use LaTeX's `lstinputlisting` package to import the code and typeset it nicely. The package imports and un-comments the embedded comments so that LaTeXcan process them and implements a long list of character substitutions, allowing the code listings below to have variables appearing with subscripts and superscripts.

A.1 Operators and scalars

Our first task is to define functions that enable *Mathematica* to distinguish between (non-commutative) operators and (commutative) scalars. This is done in the `OpQ.m` package, whose listing appears below. A variable is either a commutative *scalar* or non-commutative *operator*. Operators have a *phylum* and an *order*. Operators of different phyla commute. The operator *order* is used for sorting.

Listing 1: unidyn/OpQ.m

A *simple operator* is a symbol whose upvalue for `SimpleOperatorQ[]` is defined to be **True**. The default return for `SimpleOperatorQ` is **False**, with the result that unless defined otherwise, all arguments to the query `SimpleOpertorQ[]` return **False**. By default then, all quantities are scalars.

```
Clear[SimpleOperatorQ]
SimpleOperatorQ[x_] := False;
```

An *operator* is any expression one of whose atoms is a simple operator. The function call to **Level** in `OperatorQ[]` remarkably pulls out a list of all symbols used in the expression x (all its *atoms*.) The function `ScalarQ[]` tests if its argument is a *scalar*. By scalar we mean *not an operator*. As the default return for `OperatorQ[]` is **False**, the default return for `ScalarQ[]` is **True**; all quantities default to scalars.

```
Clear[OperatorQ, ScalarQ]
OperatorQ[x_] :=
    Apply[Or,
        Map[SimpleOperatorQ,
            Level[x ,{-1}]]];
ScalarQ[x_]:= !OperatorQ[x];
```

You "create" an operator by defining an upvalue for it, e.g., arranging so that `SimpleOperator[the operator]` returns **True**. By using upvalues, you associate this definition not with the function `SimpleOperator[]`, but with the operator itself. This makes for faster computations. The added definitions make it easy to create many operators at once by passing multiple variables or a list of variables to the function `CreateOperator[]`.

```
Clear[CreateOperator];
CreateOperator[a0_Symbol] :=
```

```
    (Clear[a0];
    SimpleOperatorQ[a0] ^:= True)
CreateOperator[a0_,b0__] :=
    (CreateOperator[a0];
    CreateOperator[b0];)
CreateOperator[a0_?VectorQ] :=
    (CreateOperator  /@ a0;)
```

You don't really need to "create" a scalar, since this is the default category for any symbol, given the above definitions. Nevertheless, by defining the upvalue of `ScalarQ` to be **True**, you can speed up computations which involve testing to see whether or not an object is a scalar.

```
Clear[CreateScalar];
CreateScalar[a0_Symbol] :=
    (Clear[a0];
    ScalarQ[a0] ^:= True)
CreateScalar[a0_List] :=
    (CreateScalar  /@ a0;)
CreateScalar[a0_,b0__] :=
    (CreateScalar[a0];
    CreateScalar[b0];)
```

Passing a matrix to `SimpleOperator` invokes the following function call. This function assigns the operators in the matrix a *phylum* and an *order* which is determined by the operators location in the matrix.

```
CreateOperator[a0_?ListQ] :=
 Module[{val, m, n},
  (Clear[#]; SimpleOperatorQ[#] ^= True) &
            /@ Flatten[a0];
  Do[
   Do[val = a0[[m]][[n]];
       phylum[val] ^= m;
       order[val] ^= n,
    {n, Dimensions[a0[[m]]][[1]]}],
   {m, Dimensions[a0][[1]]}
   ]
  ]
```

The idea of an operator having a *phylum* and an *order* can be understood best with an example. Consider, for example, the function call

`CreateOperator`$[\{\{I_x, I_y, I_z\}, \{S_x, S_y, S_z\}\}]$

This call will create six operators which, when passed to the functions `phylum` and `order`, will return the following values:

Op	`phylum`[Op]	`order`[Op]
I_x	1	1
I_y	1	2
I_z	1	3
S_x	2	1
S_y	2	2
S_z	2	3

Having a `phylum` and an `order` assigned to each operator will be used below to sort operators and to decide whether or not two perators commute.

Unit tests for the functions in the `OpQ.m` package are organized into a separate package, `OpQ-tests.m`, whose listing follows. Unit testing is built on *Mathematica*'s `VerificationTest` function. When the return value of the `VerificationTest` function is passed to the `TestReport` in a *Mathematica* notebook, the notebook displays a nice graphical report card.

Listing 2: unidyn/OpQ–tests.m

Create a shorthand function for creating unit tests.

```
If [$VersionNumber < 10.,

  vtest[label_,test_] :=
    If [test === True,
      Print[" Pass"],
      Print[" Fail > ", StringJoin["OpQ > test",ToString[label ]]]],

  vtest[label_,test_] :=
      VerificationTest [test,
        True,
        TestID → StringJoin[
          "OpQ > test",
          ToString[label ]]]
]
```

Any variable will test `True` when queried by `ScalarQ[]`, whether it has been defined as a scalar or not.

```
vtest [" 1", ScalarQ[a0] == True]
vtest [" 2", CreateScalar[b0];
    ScalarQ[b0] == True]
```

For `OperatorQ[]` to return `True`, in contrast, the variable must have been created as an operator first.

```
vtest ["3", OperatorQ[c0] == False]
vtest ["4", CreateOperator[d0];
    OperatorQ[d0] == True]
```

Any expression containing an operator will be queried **True** by `OperatorQ[]`. This is so when the expression contains only an operator or when the expression contains any combination of scalars and operators. The only case in which `OperatorQ[]` returns **False** is is it is passed an expression containing only scalars.

```
vtest ["5", OperatorQ[Times[Exp[d0],d0]] == True]
vtest ["6", OperatorQ[Times[b0,d0]] == True]
vtest ["7", OperatorQ[Times[a0,b0]] == False]
```

Operators created in a batch using

```
CreateOperator[matrix]
```

are assigned a *phylum* and an *order* which is determined by their position in the matrix.

```
CreateOperator[{{Ix, Iy, Iz},{Sx, Sy, Sz}}];

vtest ["8", phylum[Ix] == 1]
vtest ["9", order[Ix] == 1]
vtest ["10", phylum[Iy] == 1]
vtest ["11", order[Iy] == 2]
vtest ["12", phylum[Sx] == 2]
vtest ["13", order[Sx] == 1]
vtest ["14", phylum[Ix] != phylum[Sx]]
vtest ["15", order[Ix] == order[Sx]]
```

Another, more stringent test of the *phylum* and *order* system:

```
CreateOperator[{{Ix, Iy, Iz},{a, a^+}}];

vtest ["16", Not[Mult[a, a^+] === a a^+]]
vtest ["17", phylum /@ {Ix, Iy, Iz} === {1, 1, 1}]
vtest ["18", phylum /@ {a, a^+} === {2, 2}]
vtest ["19", order /@ {Ix, Iy, Iz} === {1, 2, 3}]
vtest ["20", order /@ {a, a^+} === {1, 2}]
```

Very that the operators created in a big list are in fact recognized as operators.

```
vtest ["21", OperatorQ /@ {a0, Ix, a} == {False, True, True}]
```

A.2 Non-commutative multiplication

We next define our own non-commutative multiplication function, `Mult`. This is done in the `Mult.m` package. We provide a sorting function which rearranges operators being multiplied so that, if possible, operators of a higher

phylum appear in front of operators of a lower *phylum*. During this rearrangement, non-commuting operators are not allowed to pass each other.

Listing 3: unidyn/Mult.m

First define bottom-out and empty cases:

```
Clear[Mult];
Mult[a0_] := a0
Mult[]  := 1
```

Mult distributes over `Plus` and is associative:

```
Mult[a0___,b0_Plus,c0___] :=
    Plus @@ (Mult[a0, #, c0] & )  /@ List @@ b0
Mult[a0___,Mult[b0__,c0__],d0___] :=
    Mult[a0,b0,c0,d0]
```

Define rules for factoring out scalars:

```
Mult[a0___, b0_?ScalarQ, c0___] :=
    Times[b0, Mult[a0,c0]]
Mult[a0___, b0_?ScalarQ c0__, d0___] :=
    Times[b0,Mult[a0,c0,d0]]
```

The function `NCSort` will sort the operators in a list into canonical order, being careful not to allow non-commuting operators to "pass" each other. The list that you pass to `NCSort` must contain *only* operators. How this functions works is probably easiest seen with an example. Suppose the following operators were initialized as follows:

$$\texttt{CreateOperator}[\{\{I_x, I_y, I_z\}, \{S_x, S_y, S_z\}\}]$$

Asked to sort the list of operators

$$a = \{S_x, I_y, I_x, S_y\},$$

we would report the sorted list

$$a_{\text{sorted}} = \{I_y, I_x, S_x, S_y\}.$$

The S operator, being of a higher phylum that an I operator, should be passed through the I operators. We do not pass I_x to the left of I_y because I_x and I_y do not commute.

In the code below, the statement `Map[phylum[#]&, a]` will create a list populated by the operators' phyla, in this case $\{2, 1, 1, 2\}$. To create p we multiply by the number of operators in the list plus one, 5 in this example, and add to this $\{1, 2, 3, 4\}$. This gives $p = \{11, 7, 8, 14\}$. An operator's ranking in this list depends on both its phylum and on it's location in the original list. Sorting p will tell us how to order the operators in the original list a. The variable p_{new} keeps track of the locations of the elements of p in the *sorted* version of p; in this example $p_{\text{new}} = \{3, 1, 2, 4\}$. We recognize p_{new} as the order in which the input operators should appear in a_{sorted}.

```
NCSort[a0_List] :=
Module[{n, p, anew, pnew},
    n = Length[a0];
    p = (n+1) Map[phylum[#]&, a0] + Table[i,{i,1,n}];
    pnew=(Position[Sort[p],#] [[1,1]])&  /@ p;
    anew= Table[0,{i,1,n}];
    Do[anew[[pnew[[i]]]] = a0[[i]],{i,1,n}];
    Return[anew]]
```

The function `SortedMult` is the same as `Mult` except that it reorders the operators in the call list by applying `NCSort` before passing the result to `Mult`.

```
SortedMult[a0__] :=
    Mult[Sequence @@ NCSort[List[a0]]]
```

The function `MultSort` reorders all the operators in a `NonCommutativeMultiply` call.

```
MultSort[a0__] :=
    a0 /. Mult → SortedMult
```

Extensive unit testing shows that `Mult` distributes over addition, is associative, and pulls scalars out in front as desired. Testing is also carried out on the sorting function.

Listing 4: unidyn/Mult–tests.m

First define some operators and scalars.

```
CreateOperator[{{Ix, Iy, Iz},{Sx, Sy, Sz}}];
CreateScalar[{a0, b0, c0, d0}];
```

In the following test, the left hand side should resolve to `Mult`$[I_x, I_y]$ while the right-hand side should resolve to `Times`$[I_x, I_y]$. These are *not* the same. If `Mult` does not properly recognize that I_x and I_y are operators and instead treats them as scalars, then the tests below will inadvertently pass.

```
vtest ["01", Not[Mult[Ix,Iy] === Ix Iy]]
```

Test that `Mult` distributes over addition and is associative. Note how Mult handles products of scalars and products of operators differently. Below we test for sameness (`===`) instead of equality (`==`). This is because if the two sides are *not* the same, then the equality test (using `==`) is undefined — neither true nor false — and the unit test does not fail as we wish.

```
vtest ["02a", Mult[a0, b0+ c0] === a0 b0+ a0 c0]
vtest ["02b", Mult[a0 + b0, c0] === a0 c0+ b0 c0]
```

Test that `Mult` distributes over addition and is associative. Note how `Mult` handles products of scalars and products of operators differently. Below we test for sameness (`===`) instead of equality (`==`). This is because if the two sides are *not* the same, then the equality test (using `==`) is undefined — neither true nor false — and the unit test does not fail as we wish.

```
vtest ["03", Mult[I_y, I_x] a_0 b_0 === a_0 b_0 Mult[I_y, I_x]]

vtest ["04a", Mult[a_0, b_0 + c_0] === a_0 b_0 + a_0 c_0]
vtest ["04b", Mult[a_0 + b_0, c_0] === a_0 c_0 + b_0 c_0]

vtest ["04c", Mult[I_x, I_y + I_z] === Mult[I_x, I_y] + Mult[I_x, I_z]]
vtest ["04d", Mult[I_x + I_y, I_z] === Mult[I_x, I_z] + Mult[I_y, I_z]]
```

Here is an example where we apparently do *not* have to call `NCExpand[]`.

```
vtest ["05a", Mult[I_x, Mult[S_x, S_y], I_z] === Mult[I_x, S_x, S_y, I_z]]
```

Test that scalars get factored out properly.

```
vtest ["06a", Mult[I_x, 2 I a_0, S_x] === a_0 2 I Mult[I_x, S_x]]
vtest ["06b", Mult[I_x, Mult[a_0, I_y], Mult[b_0, S_x]] === a_0 b_0 Mult[I_x, I_y, S_x]]
```

Test our sorting function:

```
vtest ["07a", NCSort[{S_x, I_x, I_y, S_z}] === {I_x, I_y, S_x, S_z}]
```

The function `MultSort[]` is used to order the operators in a standing `Mult[]` function.

```
vtest ["08a", SortedMult[I_y, S_x, I_x] === Mult[I_y, I_x, S_x]]
vtest ["08b", MultSort[Mult[I_y, S_x, I_x]] === Mult[I_y, I_x, S_x]]
```

If we now define the operators to have a different natural order, then the above tests fail. This confirms that the operators are being sorted according to our rules.

```
CreateOperator[{{S_x, S_y, S_z}, {I_x, I_y, I_z}}];
vtest ["08b", Not[SortedMult[I_y, S_x, I_x] === Mult[I_y, I_x, S_x]]]
vtest ["08d", Not[MultSort[Mult[I_y, S_x, I_x]] === Mult[I_y, I_x, S_x]]]
```

Check that `MultSort[]` works as expected when scalars are peppered into the list of operators being multiplied.

```
vtest ["09a", MultSort[Mult[a_0 I_y, S_x, I_x]]
  === a_0 Mult[S_x, I_y, I_x]]

expr1 = Mult[S_x, a_0 S_z] + Mult[I_x, b_0 S_x];
expr2 = MultSort[Mult[expr1, I_x]];
expr3 = a_0 Mult[S_x, S_z, I_x] + b_0 Mult[S_x, I_x, I_x];

vtest ["10a", expr2 === expr3]
```

A.3 Commutator

Define a commutator function which factors out scalars and has rules for simplifying more complex commutators, like and $[AB, C]$ and $[A, BC]$, in-

volving three non-commuting operators.

Listing 5: unidyn/Comm.m

```
Comm[a0_?ScalarQ, b0_] := 0
Comm[a0_, b0_?ScalarQ] := 0
Comm[a0_, a0_] := 0
```

The commutator distributes over `Plus`:

```
Comm[a0__, b0_Plus] := Plus @@ (Comm[a0, #] & ) /@ List @@ b0
Comm[a0_Plus, b0__] := Plus @@ (Comm[#, b0] & ) /@ List @@ a0
```

Rules for factoring out scalars:

```
Comm[a0__ b0_?ScalarQ, c0__] := b0Comm[a0, c0]
Comm[a0__, b0_?ScalarQ c0__] := b0Comm[a0, c0]
```

Two rules for simplifying commutators involving products:

```
Comm[a0_Mult, C_] :=
   Module[{A,B},
      A = (List @@ a0)[[1]] ;
      B = Mult @@ Rest[List @@ a0] ;
      Mult[A, Comm[B, C]] + Mult[Comm[A, C], B]
    ]

Comm[A_, b0_Mult] :=
   Module[{B,C},
      B = (List @@ b0)[[1]] ;
      C = Mult @@ Rest[List @@ b0] ;
      Mult[Comm[A, B], C] + Mult[B, Comm[A, C]]
   ]
```

Finally, a commutator between different operators of a different *phyla* is zero:

```
Comm[a0_,b0_] := 0 /; phylum[a0] != phylum[b0];
```

Extensive testing shows that the commutator function factors out scalars correctly and correctly simplifies over a dozen commutation identities, including the Jacobi identity.

Listing 6: unidyn/Comm–tests.m

```
CreateScalar[{a0, b0, c0, d0}];
CreateOperator[{{Ix, Iy, Iz},{Sx, Sy, Sz}}];
```

The commutator is distributive. The commutator of a scalar with an operator is zero. The commutator of an operator with iteslf is zero. The commutator between two operators of different phyla is zero. The commutator of two operators of the same phyla is left unevaluated.

```
vtest ["01a", Comm[Ix, Iy + Iz] === Comm[Ix, Iy] + Comm[Ix, Iz]]
vtest ["01b", Comm[a0, Ix] === 0]
vtest ["01c", Comm[Ix, Ix] === 0]
vtest ["01d", Comm[Ix, Sy] === 0]
vtest ["01e", Comm[Sx, Sy] === Comm[Sx, Sy]]
```

Scalars are properly factored out of commutators.

```
vtest ["02a", Comm[a0 Sx, Sy] === a0Comm[Sx, Sy]]
vtest ["02b", Comm[Sx, b0Sy] === b0Comm[Sx, Sy]]
vtest ["02c", Comm[a0 Sx, b0Sy] === a0b0Comm[Sx, Sy]]
```

Test the $[AB, C]$ and $[A, BC]$ expansion rules, adding in scalars to both positions. In test 03b, we see that the S_x operator in the second place on the commutator can be pulled out front since it commutes with the operator in the first place of commutator.

```
vtest ["03a", Comm[a0 Mult[Sx, Sy], b0Sz]
  === a0b0(Mult[Sx, Comm[Sy, Sz]] + Mult[Comm[Sx, Sz], Sy])]
vtest ["03b", Comm[a0 Sx, b0Mult[Sy, Sz]]
  === a0b0(Mult[Sy, Comm[Sx, Sz]] + Mult[Comm[Sx, Sy], Sz])]
vtest ["03c", Comm[Sx, Mult[Sx, Sy]]
  === Mult[Sx, Comm[Sx, Sy]]]
```

Test the Jacobi identity. For this identity to resolve to zero, we must tell *Mathematica* that [A,B] equals A**B-B**A. First, check that this substitution does what we expect, then test the Jacobi identity.

```
vtest ["04a", (Comm[Ix, Iy] //. Comm[Sx_, Sy_] → Mult[Sx, Sy] – Mult[Sy, Sx])
  === Mult[Ix, Iy] – Mult[Iy, Ix]]

vtest ["04b", ((Comm[Ix, Comm[Iy, Iz]] +
     Comm[Iy, Comm[Iz, Ix]] + Comm[Iz, Comm[Ix, Iy]])
   //. Comm[Sx_, Sy_] → Mult[Sx, Sy] – Mult[Sy, Sx])
  === 0]
```

Test the table of ten identities under "Advanced Identities" at the Wikipedia page for "Commutator",https://en.wikipedia.org/wiki/Commutator. I find that the first equality number 9 in "Advanced Identities" does not actually evaluate to True.

```
CreateOperator[{{A, B, C, D, E$sym}}];

vtest ["05.01",
Comm[A, Mult[B, C]]
  === Mult[Comm[A, B], C]
  + Mult[B, Comm[A, C]]
]

vtest ["05.02",
Comm[A, Mult[B, C, D]]
  === Mult[Comm[A, B], C, D]
  + Mult[B, Comm[A, C], D]
```

```
  + Mult[B, C, Comm[A, D]]
]

vtest ["05.03",
Comm[A, Mult[B, C, D, E$sym]]
  === Mult[Comm[A, B], C, D, E$sym]
  + Mult[B, Comm[A, C], D, E$sym]
  + Mult[B, C, Comm[A, D], E$sym]
  + Mult[B, C, D, Comm[A, E$sym]]
]

vtest ["05.04",
Comm[Mult[A, B], C]
  === Mult[A, Comm[B, C]]
  + Mult[Comm[A, C], B]
]

vtest ["05.05",
Comm[Mult[A, B, C], D]
  === Mult[A, B, Comm[C, D]]
  + Mult[A, Comm[B, D], C]
  + Mult[Comm[A, D], B, C]
]

vtest ["05.06",
Comm[Mult[A, B, C, D], E$sym]
  === Mult[A, B, C, Comm[D, E$sym]]
  + Mult[A, B, Comm[C, E$sym], D]
  + Mult[A, Comm[B, E$sym], C, D]
  + Mult[Comm[A, E$sym], B, C, D]
]

vtest ["05.07",
Comm[A, B+ C]
  === Comm[A, B]
  + Comm[A, C]
]

vtest ["05.08",
Comm[A + B, C+ D]
  === Comm[A, C]
  + Comm[A, D]
  + Comm[B, C]
  + Comm[B, D]
]

vtest ["05.09",
Comm[Mult[A, B], Mult[C, D]]
  === Mult[A, Comm[B, C], D]
  + Mult[A, C, Comm[B, D]]
  + Mult[Comm[A, C], D, B]
```

```
    + Mult[C, Comm[A, D], B]
]

vtest ["05.10",
Module[{left, right },

   left  = Comm[Comm[A, C], Comm[B, D]]
   //. Comm[A_?OperatorQ, B_?OperatorQ]  → Mult[A,B] – Mult[B,A];

   right  = Comm[Comm[Comm[A, B], C], D]
   + Comm[Comm[Comm[B, C], D], A]
   + Comm[Comm[Comm[C, D], A], B]
   + Comm[Comm[Comm[D, A], B], C]
   //. Comm[A_?OperatorQ, B_?OperatorQ]  → Mult[A,B] – Mult[B,A];

   left  === right]
]
```

A.4 Spins

Declare angular momentum operators for one spin I_x, I_y, and I_z and specify the relevant commutation relations. To speed up computation, each operator's commutation relations are stored as an *upvalue* of the operator instead of as a *downvalue* of the commutator function. Additional product-operator simplification rules are defined if the spin is declared to have an angular momentum of $\ell = 1/2$.

Listing 7: unidyn/Spins.m

```
SpinSingle$CreateOperators[Ix_,Iy_,Iz_,L_:Null] :=

Module[{nonexistent},
```

Test if the operators exist; if they do not already exist, then create them. If an operator `Op` has been created already, then `CommutativeQ[Op]` will return **True**. Unless I_x, I_y, and I_z all already exist as operators, then create all three operators afresh. In the code below it is important that we call `Mult`‘`CommutativeQ` and not just `CommutativeQ`.

```
nonexistent =
    Not[OperatorQ[Ix]] ||
    Not[OperatorQ[Iy]] ||
    Not[OperatorQ[Iz]];

If [nonexistent == True,
    Clear[Ix, Iy, Iz];
        CreateOperator[{{Ix, Iy, Iz}}];
        Message[SpinSingle$CreateOperators::create],
    Message[SpinSingle$CreateOperators::nocreate];];
```

The commutation relations are defined as *upvalues* of the various spin angular momentum operators. That is, the commutation relations are associated with the operators and not with the `Comm` function. The following commutation relations hold for any L.

```
Iₓ /:  Comm[Iₓ,Iᵧ] = I Iᵤ;
Iₓ /:  Comm[Iₓ,I_z] = –I Iᵧ;
Iᵧ /:  Comm[Iᵧ,Iₓ] = –I I_z;
Iᵧ /:  Comm[Iᵧ,I_z] = I Iₓ;
I_z /:  Comm[I_z,Iᵧ] = –I Iₓ;
I_z /:  Comm[I_z,Iₓ] = I Iᵧ;

Message[SpinSingle$CreateOperators::comm]

Switch[L,
```

When the total angular momentum $L = 1/2$, additional rules are defined to simplify products of the angular momentum operators.

```
1/2,

I_z /:  Mult[a___,I_z,I_z,b___] := Mult[a,b]/4;
Iᵧ /:  Mult[a___,Iᵧ,Iᵧ,b___] := Mult[a,b]/4;
Iₓ /:  Mult[a___,Iₓ,Iₓ,b___] := Mult[a,b]/4;

I_z /:   Mult[a___,I_z,Iₓ,b___] := I  Mult[a,Iᵧ,b]/2;
I_z /:   Mult[a___,I_z,Iᵧ,b___] := –I Mult[a,Iₓ,b]/2;

Iᵧ /:   Mult[a___,Iᵧ,Iₓ,b___] := –I Mult[a,I_z,b]/2;
Iᵧ /:   Mult[a___,Iᵧ,I_z,b___] := I  Mult[a,Iₓ,b]/2;

Iₓ /:   Mult[a___,Iₓ,I_z,b___] := –I Mult[a,Iᵧ,b]/2;
Iₓ /:   Mult[a___,Iₓ,Iᵧ,b___] := I  Mult[a,I_z,b]/2;

Message[SpinSingle$CreateOperators::simplify],
```

When the total angular momentum L is unspecified, no such simplification rules are defined.

```
Null,

Message[SpinSingle$CreateOperators::nosimplify]

];
Return[{Iₓ, Iᵧ, I_z}]
]
```

Test the commutation relations for one spin and for a system of two spins, one of them $\ell = 1/2$ and the other with ℓ unspecified.

Listing 8: unidyn/Spins–tests.m

If there is one operator in the requested list of three spin operators that is not defined, then create all three spin operators afresh. Here we check that the test does what we want. If one of the operators is undefined, the test should come out false.

```
Clear[Ix, Iy, Iz];
CreateOperator[{{Ix, Iy}}];
tests = OperatorQ /@ {Ix, Iy, Iz};
vtest["00a", And @@ tests == False]
```

If, on the other hand, all three spin operators have been defined already, then the test should come out true.

```
Clear[Ix, Iy, Iz];
CreateOperator[{{Ix, Iy, Iz}}];
tests = OperatorQ /@ {Ix, Iy, Iz};
vtest["00b", And @@ tests == True]
```

We should also check the limiting case that *none* of the operators have been defined yet.

```
Clear[Ix, Iy, Iz];
tests = OperatorQ /@ {Ix, Iy, Iz};
vtest["00c", And @@ Not /@ tests == True]
```

Create spin angular momentum operators with the total angular momentum unspecified. Test that the canonical angular momentum commutation relation holds true. With the total angular momentum unspecified, the product $I_x I_y$ cannot be simplified further.

```
Clear[Ix, Iy, Iz]
SpinSingle$CreateOperators[Ix, Iy, Iz];

vtest["01a", Comm[Ix, Iy] === I Iz]
vtest["01b", Not[Mult[Ix, Iy] === Mult[Iy, Ix]]]
vtest["01c", Not[Mult[Ix, Iy] === I Iz/2]]
vtest["01d", Not[Mult[Iz, Iz] === 1/4]]
```

Create two sets of operators, one set for I spins and one set for S spins. Assign the I-spin operators the properties of $I = 1/2$ angular momentum operators.

```
Clear[Ix, Iy, Iz, Sx, Sy, Sz];
CreateOperator[{{Ix, Iy, Iz},{Sx, Sy, Sz}}];
SpinSingle$CreateOperators[Ix, Iy, Iz, 1/2];
```

We have not defined the S operators to be spin operators yet. Nevertheless, the commutator of one S operator with another should be non-zero. On the other hand, the commutator of an S operator with an I operator should be zero.

```
vtest["02a", Not[Comm[Sx, Sz] === 0]]
vtest["02b", Comm[Iz, Sz] === 0]
```

The canonical commutation relations hold true for the I-spin operators. In addition, the product of two spin angular momentum operators can be further simplified.

```
vtest["03a", Comm[Ix, Iy] === I Iz]
```

```
vtest ["03b", Mult[Iₓ, I_y] === I I_z/2]
vtest ["03c", Mult[I_z, I_z] === 1/4]
```

Test that the double commutator of I_x with the free-evolution Hamiltonian returns the expected reult.

```
Clear[h, w, rho0, rho2];
CreateScalar[w];
h = w I_z;
rho0 = Iₓ;
rho2 = Comm[-I h, Comm[-I h,rho0]];
vtest ["04a", rho2 == - Mult[w, w, Iₓ]]
```

Try this test again with another spin operator on the backend.

```
Clear[Iₓ, I_y, I_z, Sₓ, S_y, S_z]
CreateOperator[{{Iₓ, I_y, I_z},{Sₓ, S_y, S_z}}];
SpinSingle$CreateOperators[Iₓ, I_y, I_z];
SpinSingle$CreateOperators[Sₓ, S_y, S_z];

Clear[h, w, rho0, rho2];
CreateScalar[w];
h = w Mult[I_z, S_z];
rho0 = Iₓ;
rho2 = Comm[-I h, Comm[-I h,rho0]];
vtest ["04b", rho2 == - Mult[w, w, Iₓ, S_z, S_z]]
```

A.5 Harmonic oscillator

Declare the annihilation (or lowering) operator a and the creation (or raising) operator $a^\dagger$ for a harmonic oscillator, and specify the relevant commutation relations.

Listing 9: unidyn/Osc.m

The commutation relations are defined as *upvalues* of the lowering and raising operators. Here aR = a, the lowering operator, and aL = $a^\dagger$, the raising operator.

```
OscSingle$CreateOperators[a_, a†_] :=

Module[{nonexistent},
```

Test if the operators exist; if they do not already exist, then create them.

```
nonexistent =
  Not[OperatorQ[a]] ||
    Not[OperatorQ[a†]];

If [nonexistent == True,
    Clear[a, a†];
```

```
        CreateOperator[{{a, a†}}];
        Message[OscSingle$CreateOperators::create],
    Message[OscSingle$CreateOperators::nocreate];];

a  /: Comm[a, a†] = 1;
a† /: Comm[a†, a] = -1;

Message[OscSingle$CreateOperators::comm]

Return[{a, a†}]
]
```

Check commutation relations among the harmonic-oscillator operators: a, $a^\dagger$, the number operator $N = a^\dagger a$, the unitless position operator $Q = (a^\dagger + a)/\sqrt{2}$, and the unitless momentum operator $P = i(a^\dagger - a)/\sqrt{2}$. Confirm that harmonic oscillator can exist "on top of" spin operators and are sorted correctly.

Listing 10: unidyn/Osc–tests.m

```
Create raising and lowering operators for a single harmonic oscillator.  Check that the
ooperators are indeed created. Check that they have the expected commutation relations.

Clear[a,  a†, a0, Nop];

CreateScalar[{a0}];
OscSingle$CreateOperators[a, a†];

vtest ["01a", OperatorQ  /@ {a, a†} == {True, True}]
vtest ["01b", Comm[a, a†] === 1]
vtest ["01c", Comm[a†, a] === -1]

Check that scalars are factored out of the commutations relations involving the harmonic
oscillator raising and lower operators.

vtest ["01d", Comm[a0 a, a†] === a0]
vtest ["01e", Comm[a, a0 a†] === a0]

Test the commutation relations by defining the number operator, Nop = N = a† a, and
checking the commutation relations [N, a†] = a† and [N, a] = -a.

Nop = Mult[a†, a];

vtest ["01d", Comm[Nop, a†] === a†]
vtest ["01e", Comm[Nop, a] === -a]

Clear[a,  a†, a0, Nop];

Check that we can define harmonic oscillator operators "on top of" an existing operator that
commutes with the harmonic-oscillator operators.
```

```
Clear[a, a†, a, Q];

CreateScalar[{a}];
CreateOperator[{{Q},{a†, a}}]
OscSingle$CreateOperators[a, a†];

vtest ["02a", Comm[a, a†] === 1]
vtest ["02b", Comm[a†, a] === -1]
vtest ["02c", Comm[a, Q] == 0]
vtest ["02d", Comm[a, a] == 0]
vtest ["02e", Comm[Q, a] === 0]
vtest ["02f", MultSort[Mult[a, Q]] === Mult[Q, a]]

Clear[a, a†, a, Q];
```

Test the commutations for the position and momentum operators.

```
Clear[a, a†, Q, P];

OscSingle$CreateOperators[a, a†];
Q = (a† + a)/√2;
P = I (a† - a)/√2;

vtest ["03a", Comm[Q, P] === I]
vtest ["03b", Comm[P, Q] === -I]

Clear[a, a†, Q, P];
```

Abother check that we can define harmonic oscillator operators "on top of" existing operators. In the following tests we prove that an operator like I_x (not defined as a spin operator, just an operator) commutes with one of the harmonic oscillator operators while the harmonic oscillator commutation relations are retained.

```
Clear[Ix, Iy, Iz, a, a†, Nop];
CreateOperator[{{Ix, Iy, Iz},{a†, a}}]
OscSingle$CreateOperators[a, a†];
Nop = Mult[a†, a];

vtest ["04a", Not[Comm[Ix, Iy] === 0]]
vtest ["04b", Comm[Ix, a] == 0]
vtest ["04c", Comm[Mult[Ix, Iy], a] == 0]
vtest ["04d", Comm[a, a†] === 1]
vtest ["04e", Comm[Nop, a†] === a†]
vtest ["04f", Comm[Nop, a] === -a]
```

We defined the I operators to have higher precedence than the harmonic oscillator operators. Check that `MultSort` pulls the I operators out front as expected.

```
vtest ["05", MultSort[Mult[a, Ix, a†, a, Iy]]
    === Mult[Ix, Iy, a, a†, a]]
```

```
Clear[Ix, Iy, Iz, a, a†, Nop];
```

A.6 Unitary Evolution

Before coding the `Evolver` algorithms, it is useful to define an `Evolve` function. This function serves as a preprocessor for the `Evolver` algorithms defined below. The `Evolve` function distributes evolution over sums and products in the density operator, and leading scalars are pulled out front. If all terms in the Hamiltonian commute, then `Evolve` is distributed over all terms in the Hamiltonian.

Listing 11: unidyn/Evolve.m

The evolution operator distribute over `Plus` and `Mult`.

```
Clear[Evolve];
Evolve[ℋ__, t__, ρ_Plus] :=
  Plus @@ (Evolve[ℋ, t, #]&) /@ List @@ ρ

Evolve[ℋ__, t__, ρ_Mult] :=
  Mult @@ (Evolve[ℋ, t, #]&) /@ List @@ ρ
```

Scalars in front of the density operator should be pulled out front.

```
Evolve[ℋ__, t__, Times[a_?ScalarQ, ρ__]] := a Evolve[ℋ, t, ρ]
```

A test to see if all the terms in a sum commute with each other. It is important to include an `AllCommutingQ::usage` statement at the top of this package so that this function is available in the `General`` context of the notebook.

```
AllCommutingQ[ℋ_] := Module[{H$list, Comm$matrix},
  If [Head[ℋ] === Plus,
    H$list = List @@ ℋ;
  Comm$matrix = Outer[Comm, H$list, H$list];
  Return[And @@ ((# === 0)& /@ Flatten[Comm$matrix])],
  Return[False]
  ]
]
```

If all the terms in the Hamiltonian commute, then we may distribute the `Evolve` operator over the terms in the Hamiltonian.

```
Evolve[ℋ_?AllCommutingQ, t_, ρ_] :=
  Mult @@ (Evolve[#, t, ρ]&) /@ List @@ ℋ
```

A function, from `mathematica.stackexchange`, to coerce *Mathematica* into writing simpler looking expressions.

```
VisualComplexity:=(Count[ToBoxes[#],Except[" "|"("|")", _String], Infinity ]&)
```

Unit testing verifies that `Evolver` distributes over sums and factors out scalars correctly.

Listing 12: unidyn/Evolve–tests.m

Show that the `Evolve` operator is distributive over both sums and non-commutative products of operators. Show that scalars are pulled out front, even if buried in a oomplicated product of operators and scalars. For this test, we'll make three sets of two commuting operators representing, for example, three indepdendent harmonic oscillators.

```
Clear[H, t, H1, H2, H3, Q, R, S, U, V, W, q, r, s, u, v, w];

CreateOperator[{{Q,R},{S,U},{V,W}}]
CreateScalar[{q,r,s,u,v,w}]

vtest["01a > distribute addition",
  Evolve[H, t, Q + R + S]
  === Evolve[H, t, Q] + Evolve[H, t, R] + Evolve[H, t, S]]
vtest["01b > distribute multiplication",
  Evolve[H, t, Mult[Q, R, S]]
  === Mult[Evolve[H, t, Q], Evolve[H, t, R], Evolve[H, t, S]]]
vtest["01c > distribute complicated expression",
  Evolve[H, t, Mult[(Q q), (r R), (s S)] + u U]
    === u Evolve[H, t, U] + q r s Mult[Evolve[H, t, Q], Evolve[H, t, R], Evolve[H, t, S]]]
```

Make up some Hamiltonians and see if they pass the all-terms-commuting test. The first test is particularly important. In test 03a and 03b below, nothing happens; the Hamiltonian is either so simple that it can be broken into pieces, 03a, or contains terms which do not commut, 03b. In test 03c we have a Hamiltonian whose three terms commute, and in this case the `Evolve` operator can be expanded.

```
H0 = Q;
H1 = q Mult[Q, R] + s Mult[S, U] + Mult[U, S] + Mult[V, V, W];
H2 = q Q + s S + v Mult[Q, S];

vtest["02a > commuting test 1", AllCommutingQ[H0] === False]
vtest["02b > commuting test 2", AllCommutingQ[H1] === False]
vtest["02c > commuting test 3", AllCommutingQ[H2] === True]

vtest["03a > Evolve expand test 1", Evolve[H0, t, Q] === Evolve[Q, t, Q]]
vtest["03b > Evolve expand test 2", Evolve[H1, t, Q]
  === Evolve[q Mult[Q,R] + s Mult[S,U] + Mult[U, S] + Mult[V, V, W], t, Q]]
vtest["03c > Evolve expand test 3", Evolve[H2, t, Q]
  === Mult[Evolve[q Q, t, Q], Evolve[s S, t, Q], Evolve[v Mult[Q,S], t, Q]]]

Clear[H0, H1, H2]
Clear[Q, R, S, U, V, W, q, r, s, u, v, w]
```

A.7 The Evolver1 algorithm

Implement the `Evolver1` algorithm described above. The function can be passed an option to print out intermediate variables, which is helpful for debugging.

Listing 13: unidyn/Evolver1.m

The `Evolver1` function, by default, will not print out intermediate results during the computation.

```
Options[Evolver1] = {quiet  → True};

Evolver1[ℋ_, t_, ρ0_, opts : OptionsPattern[]] :=

Module[{k, a$vect, q, r, r$value, X, x4, x3, x2, x1, time, system, sol},

  Clear[ρ];
  ρ[0] = ρ0;
```

Evalute the derivates of the density operator by repeatedly applying the commutator. Simplify them as much as possible. Getting the simplication right is tricky. A special function has to be fed to `FullSimplify` to get it to return useful results.

```
  Do[
  ρ[k+1] = (-I  Comm[ℋ, ρ[k]]  /. Mult  → SortedMult)
         //  FullSimplify [#,  ComplexityFunction → VisualComplexity]&,
    {k,0,4}
  ];
```

Print out the vector of density-operator derivatives if asked.

```
  If [OptionValue[quiet] == False,
   Print[" \[Rho] matrix = ",  ρ[#]&  /@ {0,1,2,3,4} //  MatrixForm]];
```

Look for an entry in the $(\rho^{(2)}, \rho^{(1)}, \rho^{(0)})$ list that is proportional to $\rho^{(3)}$. Stop when you find it. Determining *proportional to* is tricky when non-commuting operators are involved. Here we simply use the *Mathematica* `Divide[]` function. When $(\rho^{(n)})^{-1} ** \rho^{(3)}$ is a scalar, then we have found a match. We are implicity assuming that the entries $(\rho^{(2)}, \rho^{(1)}, \rho^{(0)})$ *have* an inverse. This will be true of then entries involve Hermitian operators. If the entries involve *non-Hermitian operators*, however, like I_+ or I_- then the entries might not have a proper inverse. We do not, at present, test whether the operators in the Hamiltonian are Hermitian or not.

```
  a$vect={0,0,0,0};
  r = Null;
  r = Catch[
    Do[
      q = Mult[
          Divide[ρ[3],  Mult[ρ[k]]]
       ]  //  FullSimplify [#,  ComplexityFunction → VisualComplexity]&;
```

```
      If [ScalarQ[q]==True, Throw[{3−k, q}]],
      {k,0,2}
    ]
  ];
```

If we have not found a match by now, then throw up our hands and exit the function. Spit out the full $(\rho^{(3)}, \rho^{(2)}, \rho^{(1)}, \rho^{(0)})$ vector, in case the user can spot the *proportional to* condition.

```
  If [r=== Null,
    Message[Evolver1::unsolvable];
    Return[ρ[#]& /@ {0,1,2,3,4}],
    r$value=r
  ];
```

Set up the coupling matrix Ω based on the matching condition.

```
  a$vect[[r$value [[1]]]] = r$value [[2]];
  A = {a$vect,{1, 0 ,0, 0},{0, 1, 0, 0}, {0, 0 ,1, 0}};
```

If asked, spit out the coupling matrix for inspection.

```
  If [OptionValue[quiet] == False, Print[" \[ CapitalOmega] = ", A // MatrixForm]];
```

Set up the four coupled equations and solve them. Respect *Mathematica* version 8 standards here and feed `DSolve[]` a list where each element of the list is an equation. First set up the equations.

```
  X[time_] = {x4[time], x3[time], x2[time], x1[time]};
   lhs$list = D[X[time],time];
   rhs$list = A . X[time];
  eqns = ( lhs$list [[#]] == rhs$list [[#]])& /@ {1,2,3,4};

  system = {eqns,
    x4[0]== ρ[3], x3[0]== ρ[2], x2[0]== ρ[1], x1[0]== ρ[0]};

  If [OptionValue[quiet] == False, Print[" system of equations = ", system // MatrixForm]];
```

Now solve them and print out the solution with the private variables replaced by public ones.

```
  sol = DSolve[system,{x1,x2,x3,x4},time];

  If [OptionValue[quiet] == False, Print[" 1st solution = ", sol [[1]][[1]] ]];
  If [OptionValue[quiet] == False, Print[" 1st solution w/ substitution = ", x1[time]
 /. sol [[1]][[1]] /. time → t]];
```

Return only the first element of the solution, $\lambda_1(t)$.

```
  Return[FullSimplify[x1[time] /. sol [[1]] /. time → t]];
];
```

We implemented the unit tests listed in Sec A Additionally, we also verify that *Mathematica's* differential equation solver is behaving as expected. We are not worried about bugs in the solver. We are instead concerned about changes in function syntax. The solver function's syntax changed between *Mathematica* version 8 and 10 and could change again. We added an equation-solver unit test to flag a future syntax change.

Listing 14: unidyn/Evolver1–tests.m

Global debugging flag to force `Evolver1` to be noisy instead of quiet.Set to False for noisy output and to True for quiet output.

```
quiet$query = True;
```

To test the unitary evolution operator on spins, let us set up a model two-spin system. Ths system is comprised of an $L = 1/2$ I spin and an unspecified-L S spin.

```
Clear[Ix, Iy, Iz, Sx, Sy, Sz, ω, d0, Δ, ρ, t]
Clear[A, r, lhs$list, rhs$list, time, eqns, system, ρcalc, ρknown, X]

CreateScalar[ω, d0, Δ];
CreateOperator[{{Ix, Iy, Iz},{Sx, Sy, Sz}}]

SpinSingle$CreateOperators[Ix, Iy, Iz, L=1/2];
SpinSingle$CreateOperators[Sx, Sy, Sz, L=1/2];
```

Test the differential equation solver first. In *Mathematica* version 10 gives more leeway in how the equations are set up – you can set one list equal to another, for example. In *Mathematica* version 8, in contrast the syntax is not so forgiving. Let us mock-up an equation by hand and feed it to the solver.

```
A = {{0,-Δ^2,0,0},{1,0,0,0},{0,1,0,0},{0,0,1,0}};
r = {Ix, Iy Δ, -Ix Δ^2, -Iy Δ^3, Ix Δ^4};
(ρ[#-1] = r[[#]])& /@ {1,2,3,4,5};

X[time_] = {x4[time], x3[time], x2[time], x1[time]};
lhs$list = D[X[time],time];
rhs$list = A . X[time];
eqns = (lhs$list[[#]] == rhs$list[[#]])& /@ {1,2,3,4};

system = {eqns,
  x4[0]== ρ[3], x3[0]== ρ[2], x2[0]== ρ[1], x1[0]== ρ[0]};
  sol = DSolve[system,{x1,x2,x3,x4},time];

ρcalc= (x1[time] /. sol[[1]] /. time → t) // Expand // ExpToTrig // FullSimplify ;
ρknown= Ix Cos[t Δ] + Iy Sin[t Δ];

vtest ["01 > DSolve test", ρcalc=== ρknown]
```

Free evolution of I_x:

```
vtest ["02a > free evolution of Ix",
  FullSimplify [ExpToTrig[Expand[
    Evolver1[ω Iz, t, Ix, quiet → quiet$query]]]]
  === Ix Cos[ω t] + Iy Sin[ω t]]
```

On-resonance nutation of I_z:

```
vtest ["02b > on−resonance nutation of Iz",
  FullSimplify [ExpToTrig[Expand[
    Evolver1[ω Ix, t, Iz, quiet → quiet$query]]]]
  === Iz Cos[ω t] − Iy Sin[ω t]]
```

Free evolution of I_+:

```
vtest ["02c > free evolution of I+",
  FullSimplify [Evolver1[ω Iz, t, Ix, quiet → quiet$query]
    + I Evolver1[ω Iz, t, Iy, quiet → quiet$query]]
  === Exp[−I ωt](Ix + I Iy)]
```

Evolution under a scalar coupling:

```
vtest ["02d > scalar−coupling evolution of Ix",
  Evolver1[d0 Mult[Iz, Sz], t, Ix, quiet → quiet$query]
  === Ix Cos[d0 t/2] + 2 Mult[Iy, Sz] Sin[d0 t/2]]
```

Off-resonance nutation of I_z. It is important to carefully set the assumptions used by `Simplify`.

```
$Assumptions = {Element[Δ, Reals], Δ> 0, Element[ω, Reals], ω>= 0};

c1= (Δ^2)/(Δ^2 + ω^2);
c2= (Δ ω)/(Δ^2 + ω^2);
c3= (ω^2)/(Δ^2 + ω^2);
c4= ω/Sqrt[Δ^2 + ω^2];
ωeff= Sqrt[Δ^2 + ω^2];

ρknown= Collect[
  c1 Iz + c2 Ix + (c3 Iz − c2 Ix) Cos[ωeff t] − c4 Iy Sin[ωeff t] //
    Expand, {Ix, Iy, Iz}];

ρcalc= Collect[
  Evolver1[Δ Iz + ω Ix, t, Iz, quiet → quiet$query]
  // FullSimplify, {Ix, Iy, Iz}, Expand];

vtest ["05e > Off−resonance nutation of of Iz", ρcalc === ρknown]
```

Clean up:

```
Clear[Ix, Iy, Iz, Sx, Sy, Sz, ω, d0, Δ, ρ, t]
Clear[A, r, lhs$list, rhs$list, time, eqns, system, ρcalc, ρknown, X]
```

Harmonic oscillator evolution. First, create the harmonic oscillator Hamiltonian in symmetric form. Evolve the lowering operator and confirm that it picks up the expected phase factor. Evolve the raising operator and confirm that it picks up the expected (conjugate) phase factor.

```
Clear[a, a†, ω, Q, P, ℋ, Q, P, δq, δp];
CreateScalar[ω, delta$x$sym, δp];
OscSingle$CreateOperators[a, a†];

ℋ = ω (Mult[a, a†] + Mult[a†, a])/2;

vtest ["03a1 > free evolution of lowering operator",
  Simplify[TrigToExp[Expand[Evolver1[ℋ, t, a, quiet → quiet$query]]]]
    === a Exp[I ω t]]

vtest ["03a2 > free evolution of raising operator",
  Simplify[TrigToExp[Expand[Evolver1[ℋ, t, a†, quiet → quiet$query]]]]
    === a† Exp[-I ω t]]
```

Evolve the position operator. To do this, write the position operator in terms of the raising and lowering operators, calculate the evolution, and rewite the answer in terms of the position and momentum operator. For the re-write step, create a *new* set of non-commuting operators, to avoid chasing our tail.

```
CreateOperator[{{Q,P}}];
{Q, P} = {(a† + a)/√2, I (a† - a)/√2};
QP$rules = {a† → (Q - I P)/√2, a → (Q + I P)/√2};
```

Write Q and P in terms of raising and lower operators, write the raising and lowering operators in terms of position and momentum, and you should get the original Q and P back again.

```
vtest ["03b > test Q definition ", Simplify[Q /. QP$rules] === Q]
vtest ["03c > test P definition ", Simplify[P /. QP$rules] === P]
```

Now we are ready to evolve position and momentum.

```
vtest ["03d > free evolution of Q",
  Simplify[ExpToTrig[Expand[Evolver1[ℋ, t, Q, quiet → quiet$query] /. QP$rules]]]
    == Q Cos[ω t] - P Sin[ω t]]

vtest ["03e > free evolution of P",
  Simplify[ExpToTrig[Expand[Evolver1[ℋ, t, P, quiet → quiet$query] /. QP$rules]]]
    == P Cos[ω t] + Q Sin[ω t]]
```

Check that the operator $e^{-i\delta q\, P}$ delivers a position kick and that the operator $e^{-i\delta p\, X}$ delivers a momentum kick. These are examples of evolution where the commuting series terminates.

```
vtest ["03f > position kick ",
  Simplify[Evolver1[δq P, t, Q] /. QP$rules ~Join~ {t → 1}]
   == Q - δq]
```

```
vtest ["03g > momentum kick",
  Simplify[Evolver1[δp Q, t, P] /. QP$rules ~Join~ {t → 1}]
   == P + δp]
```

Clean up:

```
Clear[a, a†, ω, Q, P, ℋ, Q, P, δq, δp];
```

A.8 The Evolver2 algorithm

Implement the `Evolver2` algorithm described above. The function can be passed an option to print out intermediate variables, which is helpful for debugging.

Listing 15: unidyn/Evolver2.m

The `Evolver2` function, by default, will not print out intermediate results during the computation.

```
Options[Evolver2] = {quiet → True};

Evolver2[ℋ_, t_, ρ0_, opts : OptionsPattern[]] :=

Module[{ρ, divisions, commutators, ρcalc, ω},

  ρ[0] = ρ0;
```

Evalute the derivatives of the density operator $\rho^{(n)}$ by repeatedly applying the commutator. Simplify the derivatives as much as possible. Getting the simplication right is tricky; a special function has to be fed to `FullSimplify` to get it to return useful results.

```
  Do[
  ρ[k+1] = (-I Comm[ℋ, ρ[k]] /. Mult → SortedMult)
         // FullSimplify [#, ComplexityFunction → VisualComplexity]&,
    {k,0,4}
  ];

 If [OptionValue[quiet] == False,
  Print["  \!\(\* SuperscriptBox [\(\[ Rho]\), \(( n )\)]\)=  ", ρ[#]& /@ {0,1,2,3}]];
```

Work through cases 0 and 1L.

```
 If [ρ[1] == 0,
    If [OptionValue[quiet] == False, Print[" Case 0"]];
   Return[ρ[0]]
 ];

 If [Comm[ρ[1],ℋ ] == 0,
    If [OptionValue[quiet] == False, Print[" Case 1L"]];
```

```
  Return[ρ[0] + ρ[1] t]
];
```

Compute the commutators c_n.

```
commutators = {
  ρ[1],
  Comm[ρ[0], ρ[1]],
  Comm[ρ[0], ρ[2]],
  Comm[ρ[1], ρ[3]]
};

If[OptionValue[quiet] == False, Print[" commutators = ",commutators]];
```

Compute the ratios d_n.

```
divisions = {
  Mult[Inv[ρ[0]], Inv[Inv[ρ[1]]]],
  Mult[Inv[ρ[0]], Inv[Inv[ρ[2]]]],
  Mult[Inv[ρ[1]], Inv[Inv[ρ[3]]]]
};

If[OptionValue[quiet] == False, Print[" divisions = ", divisions]];
```

Work through cases 1E, 2E, and 3E.

```
If[commutators[[2]] == 0,
  If[OptionValue[quiet] == False, Print[" Case 1E"]];

  ρcalc= Mult[Exp[MultSort[divisions[[1]]] t], ρ[0]];
  Return[ρcalc]
];

If[commutators[[3]] == 0,
  If[OptionValue[quiet] == False, Print[" Case 2E"]];

  ω = PowerExpand[Sqrt[-MultSort[divisions[[2]]]]];
  If[OptionValue[quiet] == False, Print[" ω = ", ω]];

  ρcalc= Mult[Cos[ω t], ρ[0]] + Mult[Sin[ω t], commutators[[1]]/ω];
  Return[ρcalc]
];

If[commutators[[4]] == 0,
  If[OptionValue[quiet] == False, Print[" Case 3E"]];

  ω = PowerExpand[Sqrt[-MultSort[divisions[[3]]]]];
  If[OptionValue[quiet] == False, Print[" ω = ", ω]];

  ρcalc= ρ[0] + Mult[Sin[ω t], ρ[1]/ω] + Mult[1 - Cos[ω t], ρ[2]/ω^2];
  Return[ρcalc]
```

```
];
```

If we get this far, then the evolution is unsolvable. In this case, return the derivatives of the density operator $\rho^{(n)}$ for debugging purposes.

```
Message[Evolver2::unsolvable];
Return[{ρ[#]& /@ {0,1,2,3},commutators,divisions}];
];
```

Unit testing is the same as for `Evolver1`, Sec. A.7, minus the testing of *Mathematica's* differential equation solver.

Listing 16: unidyn/Evolver2–tests.m

Global debugging flag to force `Evolver22` to be noisy instead of quiet.Set to False for noisy output and to True for quiet output.

```
quiet$query = True;
```

To test the unitary evolution operator on spins, let us set up a model two-spin system. Ths system is comprised of an $L = 1/2$ I spin and an unspecified-L S spin.

```
Clear[Ix, Iy, Iz, Sx, Sy, Sz, ω, d0, Δ, ρ, t]
Clear[A, r, lhs$list , rhs$list , time, eqns, system, ρcalc, ρknown, X]

CreateScalar[ω, d0, Δ];
CreateOperator[{{Ix, Iy, Iz},{Sx, Sy, Sz}}]

SpinSingle$CreateOperators[Ix, Iy, Iz, L=1/2];
SpinSingle$CreateOperators[Sx, Sy, Sz, L=1/2];
```

Free evolution of I_x:

```
vtest ["01a > free evolution of Ix",
  FullSimplify [ExpToTrig[Expand[
    Evolver2[ω Iz, t, Ix, quiet → quiet$query]]]]
  === Ix Cos[ω t] + Iy Sin[ω t]]
```

On-resonance nutation of I_z:

```
vtest ["01b > on–resonance nutation of Iz",
  FullSimplify [ExpToTrig[Expand[
    Evolver2[ω Ix, t, Iz, quiet → quiet$query]]]]
  === Iz Cos[ω t] – Iy Sin[ω t]]
```

Free evolution of I_+:

```
vtest ["01c > free evolution of I+",
  FullSimplify [Evolver2[ω Iz, t, Ix, quiet → quiet$query]
    + I Evolver2[ω Iz, t, Iy, quiet → quiet$query]]
  === Exp[–I ωt](Ix + I Iy)]
```

Evolution under a scalar coupling:

```
vtest ["01d > scalar−coupling evolution of Ix ",
  Evolver2[d0 Mult[Iz, Sz], t, Ix, quiet → quiet$query]
  === Ix Cos[d0 t/2] + 2 Mult[Iy, Sz] Sin[d0 t/2]]
```

Off-resonance nutation of I_z. It is important to carefully set the assumptions used by `Simplify`.

```
$Assumptions = {Element[Δ, Reals], Δ> 0, Element[ω, Reals], ω>= 0};

c1= (Δ^2)/(Δ^2 + ω^2);
c2= (Δ ω)/(Δ^2 + ω^2);
c3= (ω^2)/(Δ^2 + ω^2);
c4= ω/Sqrt[Δ^2 + ω^2];
ωeff= Sqrt[Δ^2 + ω^2];

ρknown= Collect[
  c1 Iz + c2 Ix + (c3 Iz − c2 Ix) Cos[ωeff t] − c4 Iy Sin[ωeff t] //
    Expand, {Ix, Iy, Iz}];

ρcalc= Collect[
  Evolver2[Δ Iz + ω Ix , t , Iz, quiet → quiet$query]
  // FullSimplify , {Ix, Iy, Iz}, Expand];

vtest ["01e > Off−resonance nutation of of Iz ", ρcalc=== ρknown]
```

Clean up:

```
Clear[Ix, Iy, Iz, Sx, Sy, Sz, ω, d0, Δ, ρ, t]
Clear[A, r , lhs$list , rhs$list , time, eqns, system, ρcalc, ρknown, X]
```

Harmonic oscillator evolution. First, create the harmonic oscillator Hamiltonian in symmetric form. Evolve the lowering operator and confirm that it picks up the expected phase factor. Evolve the raising operator and confirm that it picks up the expected (conjugate) phase factor.

```
Clear[a, a†, ω, Q, P, ℋ, Q, P, δq, δp];
CreateScalar[ω, delta$x$sym, δp];
OscSingle$CreateOperators[a, a†];

ℋ= ω (Mult[a, a†] + Mult[a†, a])/2;

vtest ["02a1 > free evolution of lowering operator",
  Simplify [TrigToExp[Expand[Evolver2[ℋ, t, a, quiet → quiet$query]]]]
    === a Exp[I ω t ]]

vtest ["02a2 > free evolution of raising operator",
  Simplify [TrigToExp[Expand[Evolver2[ℋ, t, a†, quiet → quiet$query]]]]
    === a† Exp[−I ω t]]
```

Evolve the position operator. To do this, write the position operator in terms of the raising and lowering operators, calculate the evolution, and rewite the answer in terms of the position and momentum operator. For the re-write step, create a *new* set of non-commuting operators, to avoid chasing our tail.

```
CreateOperator[{{Q,P}}];
{Q, P} = {(a† + a)/√2, I (a† − a)/√2};
QP$rules = {a† → (Q − I P)/√2, a → (Q + I P)/√2};
```

Write Q and P in terms of raising and lower operators, write the raising and lowering operators in terms of position and momentum, and you should get the original Q and P back again.

```
vtest ["02b > test Q definition ", Simplify [Q /. QP$rules] === Q]
vtest ["02c > test P definition ", Simplify [P /. QP$rules] === P]
```

Now we are ready to evolve position and momentum.

```
vtest ["02d > free evolution of Q",
  Simplify [ExpToTrig[Expand[Evolver2[ℋ, t, Q, quiet → quiet$query] /. QP$rules]]]
    == Q Cos[ω t] − P Sin[ω t]]

vtest ["02e > free evolution of P",
  Simplify [ExpToTrig[Expand[Evolver2[ℋ, t, P, quiet → quiet$query] /. QP$rules]]]
    == P Cos[ω t] + Q Sin[ω t]]
```

Check that the operator $e^{-i\delta q\, P}$ delivers a position kick and that the operator $e^{-i\delta p\, X}$ delivers a momentum kick. These are examples of evolution where the commuting series terminates.

```
vtest ["02f > position kick ",
  Simplify [Evolver2[δq P, t, Q] /. QP$rules ~Join~ {t → 1}]
   == Q − δq]

vtest ["02g > momentum kick",
  Simplify [Evolver2[δp Q, t, P] /. QP$rules ~Join~ {t → 1}]
   == P + δp]
```

Clean up:

```
Clear[a, a†, ω, Q, P, ℋ, Q, P, δq, δp];
```

A.9 Spin-boson system

Define spin $I = 1/2$ angular momentum and creation/annihilation operators describing a two-level electronic system. Define creation and annihilation operators describing either photons or a harmonic oscillator. These definitions enable us to describe electron transfer using Marcus theory and address problems in quantum optics using the Jaynes-Cummings Hamiltonian.

Listing 17: unidyn/SpinBoson.m

```
SpinBoson$CreateOperators[Ix_, Iy_, Iz_, I+_, I-_, a†_, a_] :=

 Module[{nonexistent},

Test if all the operators exist; if any of them do not already exist, then create all of them.

 nonexistent =
     Not[OperatorQ[Ix]] ||
     Not[OperatorQ[Iy]] ||
     Not[OperatorQ[Iz]] ||
     Not[OperatorQ[I+]] ||
     Not[OperatorQ[I-]] ||
     Not[OperatorQ[a†]] ||
     Not[OperatorQ[a]];

  If [nonexistent == True,
    Clear[Ix, Iy, Iz, I+, I-, a†, a];
      CreateOperator[{{Ix, Iy, Iz, I+, I-}, {a, a†}}];
      SpinSingle$CreateOperators[Ix, Iy, Iz, 1/2];
      OscSingle$CreateOperators[a, a†];
      Message[SpinBoson$CreateOperators::create],
    Message[SpinBoson$CreateOperators::nocreate];];

Add raising and lowering operator commutation rules. The commutations relations are defined as upvalues of the spin raising and lowering operators.

 I+ /: Comm[I+, I-] = 2 Iz;
 I+ /: Comm[I+, Iz] = -I+;
 I- /: Comm[I-, I+] = -2 Iz;
 I- /: Comm[I-, Iz] = I-;
 Iz /: Comm[Iz, I+] = I+;
 Iz /: Comm[Iz, I-] = -I-;

 Message[SpinBoson$CreateOperators::comm];

Add raising and lowering operator simplification rules.

 I+ /: Mult[a___, I+, I+, b___] := 0;
 I- /: Mult[a___, I-, I-, b___] := 0;

 I+ /: Mult[a___, I+, Iz,b___] := -(1/2)Mult[a, I+, b];
 I+ /: Mult[a___, I+, I-,b___] := 1/2 Mult[a, b] + Mult[a, Iz, b];

 I- /: Mult[a___, I-, Iz, b___]:= 1/2 Mult[a, I-, b];
 I- /: Mult[a___, I-, I+, b___] := 1/2 Mult[a, b] - Mult[a, Iz, b];

 Iz /: Mult[a___, Iz, I+, b___] := 1/2 Mult[a, I+, b];
 Iz /: Mult[a___, Iz, I-, b___] := -(1/2)Mult[a, I-, b];
```

```
    Message[SpinBoson$CreateOperators::simp];

Try to achieve normal ordering of the harmonic oscillator raising and lowering operators.

    a† /: Mult[a___, a, a†, b___] := Mult[a, a†, a, b] + Mult[a, b];

    Message[SpinBoson$CreateOperators::normord];

    Return[{Iₓ, I_y, I_z, I₊, I₋, a†, a}]
    ]
```

Carry out a few unit tests to verify normal ordering and angular momentum identities.

Listing 18: unidyn/SpinBoson–tests.m

```
Create some operators to play with.

    SpinBoson$CreateOperators[Iₓ, I_y, I_z, I₊, I₋, a†, a];

Check that we get the expected Hamiltonian after normal ordering.

    vtest ["01", Simplify[1/2 (Mult[a, a†] + Mult[a†, a])] == Mult[a†, a] + 1/2]

A normal-ordering check for products of two creation/annihilation operators.

    vtest ["02a", Mult[a, a†] == Mult[a†, a] + 1]

A normal-ordering check for products of three creation/annihilation operators.

    vtest ["03a", Mult[a, a†, a] == Mult[a†, a, a] + a]
    vtest ["03b", Mult[a, a, a†] == Mult[a†, a, a] + 2 a]
    vtest ["03c", Mult[a†, a, a†] == Mult[a†, a†, a] + a†]
    vtest ["03d", Mult[a, a†, a†] == Mult[a†, a†, a] + 2 a†]

Two angular momentum identities.

    vtest ["04a", Mult[I₋, I₊] == Simplify[3/4 - Mult[I_z, I_z] - I_z]]
    vtest ["04b", Mult[I₊, I₋] == Simplify[3/4 - Mult[I_z, I_z] + I_z]]
```