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

John A. Marohn[*]
Dept. of Chemistry and Chemical Biology
Cornell University, Ithaca, NY 14851-1301, *USA*

November 9, 2024

Abstract

We have developed a *Mathematica* algorithm for symbolically calculating the unitary transformations of quantum-mechanical operators. The algorithm obtains closed-form analytical results, does not rely on a matrix representation of operators, and is applicable to both bounded systems like coupled spins and unbounded systems like harmonic oscillators. The rotations are "self derived" from the operators' underlying commutation relations. Example calculations are presented involving magnetic resonance and quantum optics.

1 Introduction

Given a time-independent Hamiltonian $\mathcal{H}$ and an initial (density) operator $\rho(0)$, the Evolver algorithm described below implements the following unitary rotation:

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

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 algorithm obtains a closed-form solution to Eq. 1 by extending an approach introduced by Slichter in Section 2.3 of his *Principles of Magnetic Resonance* text [1].

[*] jam99@cornell.edu

Slichter was interested in calculating the unitary evolution of angular momentum operators arising in spin-physics problems. Below we introduce the Slichter procedure by considering a few representative examples. We show that his procedure works remarkably well for calculating the unitary evolution in other cases as well — calculating the time evolution of position and momentum operators evolving under the harmonic-oscillator Hamiltonian, for example. It will become apparent that automating Slichter's procedure in a computer algebra program like *Mathematica* is possible but challenging. We introduce a generalization of the Slichter algorithm that is well suited for automation by a computer algebra program. We have implemented this algorithm as a *Mathematica* function, Evolver. This function evaluates Eq. 1 given an $\mathcal{H}$, a $\rho(0)$, and the commutation relations between the operators comprising $\mathcal{H}$ and $\rho(0)$.

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

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

Method 1 — One approach to computing $\rho(t)$ is to expand Eq. 2 in a Taylor series,

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

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

$$\rho^{(1)}(t) = e^{-i\,\omega t\, I_z}\,(-i\,\omega\, I_z)\, I_+ e^{+i\,\omega t\, I_z} + e^{-i\,\omega t\, I_z}\, I_+ (+i\,\omega\, I_z)\, e^{+i\,\omega t\, I_z} \tag{4a}$$

$$= e^{-i\,\omega t\, I_z}\,(-i\,\omega [I_z, I_+])\, e^{+i\,\omega t\, I_z} \tag{4b}$$

$$\rho^{(1)}(0) = -i\,\omega\, I_+ \tag{4c}$$

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

$$\rho^{(2)}(t) = e^{-i\,\omega t\, I_z}\,((-i\,\omega)^2 [I_z, [I_z, I_+]])\, e^{+i\,\omega t\, I_z} \tag{5a}$$

$$\rho^{(2)}(0) = (-i\,\omega)^2\, I_+ \tag{5b}$$

By induction, we see that

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

Substituting this finding into the Taylor expansion gives

$$\rho(t) = I_{+} \left(1 + (-i\,\omega\,t) + \frac{1}{2!}(-i\,\omega t)^2 + \frac{1}{3!}(-i\,\omega t)^3 + \cdots \right) \tag{7}$$

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

$$\rho(t) = I_{+}\, e^{-i\,\omega\,t}. \tag{8}$$

Method 2 — A second approach to evaluating Eq. 2 is to expand the exponential using the Löwdin projection-operator theorem [2]. 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,

$$e^{-i\,\omega t\, I_z} = e^{-i\,\omega t/2}\,|\alpha\rangle\langle\alpha| + e^{+i\,\omega t/2}\,|\beta\rangle\langle\beta|\,. \tag{9}$$

Substituting this result into Eq. 2 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

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

While both these approaches yield closed-form solutions, each is hardly extensible. The first method requires the resumming of a Taylor series; this step would be difficult or impossible to automate for any evolution more complicated that the one above. The second method requires obtaining the eigenvalues of the Hamiltonian, usually by reducing it to matrix form and diagonalizing it. It is hard to see how to apply this diagonalization procedure in an infinite-level system like the idealized harmonic oscillator. Even for a spin problem where the number of levels is finite, diagonalizing $\mathcal{H}$ would force us to write down the matrix representation of $\mathcal{H}$ which would in turn commit us to specifying the total angular momentum I of the spin we are interested in. In many problems, we would like to obtain a solution valid for a spin of *any* I.

Method 3, Example 1 — Now consider Slichter's procedure. Let us take another look at the derivative of ρ:

$$\dot{\rho}(t) = e^{-i\,\omega t\, I_z}\left(-i\,\omega[I_z, I_+]\right) e^{+i\,\omega t\, I_z} \tag{11a}$$

$$= -i\,\omega\left(e^{-i\,\omega t\, I_z}\, I_+\, e^{+i\,\omega t\, I_z}\right) \tag{11b}$$

As before we have used $[I_z, I_+] = I_+$ to reduce the commutator in Eq. 11a. The key insight in the Slichter procedure is that the term in parenthesis in Eq. 11b is nothing more than the original time dependent density operator, $\rho(t)$. This insight allows us to write Equation 11b as

$$\dot{\rho}(t) = -i\,\omega\,\rho(t) \tag{12}$$

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

$$\rho(t) = \rho(0)\, e^{-i\,\omega t} = I_+\, e^{-i\,\omega t}, \tag{13}$$

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 $\rho(0) = a^{\dagger}$:

$$\rho(t) = e^{-i\omega t(a^{\dagger}a+1/2)}\, a^{\dagger}\, e^{+i\omega t(a^{\dagger}a+1/2)} \tag{14}$$

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

$$\rho(t) = a^{\dagger} e^{-i\,\omega t}. \tag{15}$$

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. 12. In light of the two previous methods, the Slichter procedure is rather remarkable. It allows us to obtain a closed-form solution for Eq. 2 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

$$\rho(t) = e^{-i\,\omega t\, I_z}\, I_x\, e^{+i\,\omega t\, I_z}. \tag{16}$$

Taking the time derivative we obtain

$$\dot{\rho}(t) = e^{-i\,\omega t\, I_z}\,(-i\,\omega[I_z, I_x])\, e^{+i\,\omega t\, I_z} \tag{17a}$$

$$= \omega\left(e^{-i\,\omega t\, I_z}\, I_y\, e^{+i\,\omega t\, I_z}\right) \tag{17b}$$

where we have used $[I_z, I_x] = i\, I_y$ to reduce the commutator in Eq. 17a. In contrast to the previous case, $\dot{\rho}(t)$ is not proportional to $\dot{\rho}$. Taking another time derivative we obtain

$$\ddot{\rho}(t) = \omega\, e^{-i\,\omega t\, I_z}\,(-i\,\omega[I_z, I_y])\, e^{+i\,\omega t\, I_z} \tag{18a}$$

$$= -\omega^2\left(e^{-i\,\omega t\, I_z}\, I_x\, e^{+i\,\omega t\, I_z}\right) \tag{18b}$$

where we have used $[I_z, I_y] = -i\, I_x$ to reduce the commutator in Eq. 17a. The term in parenthesis in Eq. 18b is proportional to $\rho(t)$ and consequently

$$\ddot{\rho}(t) = -\omega^2\, \rho(t). \tag{19}$$

The solution to this second-order differential equation is

$$\rho(t) = \rho(0)\,\sin(\omega t) + \frac{\dot{\rho}(0)}{\omega}\,\cos(\omega t). \tag{20}$$

We are given that $\rho(0) = I_x$ and we see from Eq. 17b that $\dot{\rho}(0) = \omega\, I_y$. Plugging these initial conditions into the above equation we obtain

$$\rho(t) = I_x\,\sin(\omega t) + I_y\,\cos(\omega t) \tag{21}$$

as the solution to Eq. 16.

We can imagine automating the steps leading to Eqs. 13, 14, and 21. What is required is the ability to

1. perform non-commutative algebra,
2. evaluate commutators, and
3. 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 beautifully by Helton and co-workers, whose `NCALgebra` package [3] gives *Mathematica* the ability to manipulate non-commuting algebraic expressions. This 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). As we will show below, this package forms an excellent starting point for writing rules to expand and simplify commutators.

Based on our experience so far, developing an algorithm capable of self-deriving Eqs. 13, 14, and 21 would now seem to be a matter of calculating

$$\rho^{(1)} = U^\dagger \, [-i\mathcal{H}, \rho(0)] \, U \tag{22}$$

$$\rho^{(2)} = U^\dagger \, [-i\mathcal{H}, [-i\mathcal{H}, \rho(0)]] \, U = U^\dagger \, [-i\mathcal{H}, \rho^{(1)}] \, U \tag{23}$$

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

$$\rho(t) = e^{-i\,(\Delta\omega I_z + \omega_1 I_x)\,t} \, I_z \, e^{+i\,(\Delta\omega I_z + \omega_1 I_x)\,t} \equiv U^\dagger \, I_x \, U \tag{24}$$

Computing the first few derivatives, we find

$$\dot{\rho} = U^{\dagger}\left(-i[\Delta\omega I_z + \omega_1 I_x, I_z]\right)U \tag{25a}$$
$$= U^{\dagger}\left(-\omega_1 I_y\right)U \tag{25b}$$
$$\ddot{\rho} = U^{\dagger}\left(-i[\Delta\omega I_z + \omega_1 I_x, -\omega_1, I_y]\right)U \tag{25c}$$
$$= U^{\dagger}\left(\Delta\omega\,\omega_1 I_x - \omega_1^2 I_z\right)U \tag{25d}$$
$$\dddot{\rho} = U^{\dagger}\left(-i[\Delta\omega I_z + \omega_1 I_x, -\omega_1, \Delta\omega\,\omega_1 I_x - \omega_1^2 I_z]\right)U \tag{25e}$$
$$= U^{\dagger}\left(\omega_1\left(\omega_1^2 + \Delta\omega^2\right) I_y\right)U \tag{25f}$$

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

$$\ddot{\sigma} = -(\Delta\omega^2 + \omega_1^2)\,\sigma \tag{26}$$

with initial conditions

$$\sigma(0) = \dot{\rho}(0) = -\omega_1 I_y \tag{27a}$$
$$\dot{\sigma}(0) = \ddot{\rho}(0) = \Delta\omega\,\omega_1 I_x - \omega_1^2 I_z \tag{27b}$$

The solution to Eq. 26 is

$$\sigma(t) = \sigma(0)\cos(\omega_{\text{eff}} t) + \frac{\dot{\sigma}(0)}{\omega_{\text{eff}}}\sin(\omega_{\text{eff}} t) \tag{28}$$

with

$$\omega_{\text{eff}} \equiv \sqrt{\Delta\omega^2 + \omega_1^2} \tag{29}$$

an effective evolution frequency. Plugging in initial conditions,

$$\sigma(t) = \frac{\Delta\omega\,\omega_1}{\omega_{\text{eff}}} I_x \sin(\omega_{\text{eff}}\, t) - \omega_1 I_y \cos(\omega_{\text{eff}}\, t) - \frac{\omega_1^2}{\omega_{\text{eff}}} I_z \sin(\omega_{\text{eff}}\, t) \tag{30}$$

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

$$\rho(t) = -\frac{\Delta\omega\,\omega_1}{\omega_{\text{eff}}^2} I_x \sin(\omega_{\text{eff}}\, t) - \frac{\omega_1}{\omega_{\text{eff}}} I_y \sin(\omega_{\text{eff}}\, t) + \frac{\omega_1^2}{\omega_{\text{eff}}^2} I_z \cos(\omega_{\text{eff}}\, t) + c \tag{31}$$

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,

$$c = \frac{\Delta\omega \, \omega_1}{\Delta\omega^2 + \omega_1^2} I_x + \frac{\Delta\omega^2}{\Delta\omega^2 + \omega_1^2} I_z \tag{32}$$

Plugging this integration constant into Eq. 31 we obtain the following solution for the density operator in Eq. 24:

$$\begin{aligned} \rho(t) = \left(\frac{\Delta\omega^2}{\Delta\omega^2 + \omega_1^2} I_z + \frac{\Delta\omega \, \omega_1}{\Delta\omega^2 + \omega_1^2} I_x \right) + \frac{\omega_1}{\sqrt{\Delta\omega^2 + \omega_1^2}} \sin\left(\sqrt{\Delta\omega^2 + \omega_1^2}\, t\right) \\ + \left(\frac{\omega_1^2}{\Delta\omega^2 + \omega_1^2} I_z - \frac{\Delta\omega \, \omega_1}{\Delta\omega^2 + \omega_1^2} I_x \right) \cos\left(\sqrt{\Delta\omega^2 + \omega_1^2}\, t\right). \end{aligned} \tag{33}$$

We see in this example an answer for $\rho(t)$ that is markedly more complicated that in the previous examples, the solution to (essentially) 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 considered 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.

Method 4 — We now show by example that the single differential equation in each of these four 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 four 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 Evolver algorithm.

To understand the new method, consider transforming the second-order differential equation in Eq. 19 into two coupled first-order equations. Let the two new variables be

$$x_1 = \rho \text{ and } x_2 = \dot{\rho}. \tag{34}$$

Taking the time derivative of each of these variables we obtain

$$\dot{x}_1 = \dot{\rho} = x_2, \text{ and} \tag{35a}$$

$$\dot{x}_1 = \ddot{\rho} = -\omega^2 \rho = -\omega^2 x_1. \tag{35b}$$

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

$$\frac{d}{dt}\begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0 & -\omega^2 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} x_2 \\ x_1 \end{bmatrix} \text{ with } \begin{bmatrix} x_2(0) \\ x_1(0) \end{bmatrix} = \begin{bmatrix} \omega\, I_y \\ I_x \end{bmatrix}. \tag{36}$$

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

$$x_1 = \rho \text{ and } x_2 = \dot{\rho} \text{ and } x_3 = \ddot{\rho}, \tag{37}$$

and taking the time derivative gives

$$\begin{aligned} \dot{x}_1 &= \dot{\rho} = x_2, & (38a)\\ \dot{x}_2 &= \ddot{\rho} = x_3, \text{ and} & (38b)\\ \dot{x}_3 &= \dddot{\rho} & (38c)\\ &= \omega^2\, U^\dagger\, (-i\,\omega[I_z, I_x])\, U & (38d)\\ &= -\omega^3\, U^\dagger\, I_y\, U & (38e)\\ &= -\omega^2\, \dot{\rho} & (38f)\\ &= -\omega^2\, x_2. & (38g) \end{aligned}$$

In writing $\dot{x}_3$ we have used the shorthand $U(t) \equiv \exp[i\,\omega t\, I_z]$ and employed Eq. 17b to simplify the result. The three variables satisfy the following set of coupled first-order differential equations

$$\frac{d}{dt}\begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 0 & -\omega^2 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} x_3 \\ x_2 \\ x_1 \end{bmatrix} \text{ with } \begin{bmatrix} x_3(0) \\ x_2(0) \\ x_1(0) \end{bmatrix} = \begin{bmatrix} -\omega^2\, I_x \\ \omega\, I_y \\ I_x \end{bmatrix} \tag{39}$$

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. 36. 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

$$\begin{bmatrix} \rho^{(3)} \\ \rho^{(2)} \\ \rho^{(1)} \\ \rho^{(0)} \end{bmatrix} = U^\dagger \begin{bmatrix} -\Delta\omega(\Delta\omega^2 + \omega_1^2)\, I_y \\ -\Delta\omega\,(\Delta\omega\, I_x - \omega_1 I_z) \\ \Delta\omega\, I_y \\ I_x \end{bmatrix} U \tag{40}$$

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

$$U \equiv \exp\left[i\left(\Delta\omega I_z + \omega_1 I_x\right)t\right]. \tag{41}$$

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:

$$\underbrace{\frac{d}{dt}\begin{bmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix}}_{\dot{\boldsymbol{\lambda}}} = \underbrace{\begin{bmatrix} 0 & -(\Delta\omega^2 + \omega_1^2) & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\boldsymbol{\Omega}} \underbrace{\begin{bmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix}}_{\boldsymbol{\lambda}} \tag{42}$$

with

$$\underbrace{\begin{bmatrix} x_4(0) \\ x_3(0) \\ x_2(0) \\ x_1(0) \end{bmatrix}}_{\boldsymbol{\lambda}_0} = \begin{bmatrix} -\Delta\omega(\Delta\omega^2 + \omega_1^2)\, I_y \\ -\Delta\omega\,(\Delta\omega\, I_x - \omega_1 I_z) \\ \Delta\omega\, I_y \\ I_x \end{bmatrix} \tag{43}$$

Solving these coupled equations, we obtain Eq. 33 for $x_1 = \rho(t)$.

In the above equations we have defined $\boldsymbol{\lambda}$ as the vector of x_n's 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

$$\frac{d\boldsymbol{\lambda}}{dt} - \boldsymbol{\Omega} \cdot \boldsymbol{\lambda} = 0 \tag{44}$$

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

5. Feed Eq. 44, including the initial condition, to *Mathematica* to solve. The time-dependent density operator is $\rho(t) = \lambda_1(t)$.

2 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. Unit tests for the functions in this package are organized into a separate package, `OpQ-tests.m`, whose listing follows.

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[a_Symbol] :=
    (Clear[a];
    SimpleOperatorQ[a] ^:= True)
CreateOperator[a_,b__] :=
    (CreateOperator[a];
    CreateOperator[b];)
CreateOperator[a_?VectorQ] :=
    (CreateOperator  /@ a;)
```

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[a_Symbol] :=
        (Clear[a];
        ScalarQ[a] ^:= True)
    CreateScalar[a_List] :=
        (CreateScalar /@ a;)
    CreateScalar[a_,b__] :=
        (CreateScalar[a];
        CreateScalar[b];)
```

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

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

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.

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[a] == True]
    vtest["2", CreateScalar[b];
        ScalarQ[b] == True]
```

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

```
    vtest["3", OperatorQ[c] == False]
    vtest["4", CreateOperator[d];
        OperatorQ[d] == 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[d],d]] == True]
    vtest["6", OperatorQ[Times[b,d]] == True]
    vtest["7", OperatorQ[Times[a,b]] == 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[{{I_x, I_y, I_z},{S_x, S_y, S_z}}];

vtest["8", phylum[I_x] == 1]
vtest["9", order[I_x] == 1]
vtest["10", phylum[I_y] == 1]
vtest["11", order[I_y] == 2]
vtest["12", phylum[S_x] == 2]
vtest["13", order[S_x] == 1]
vtest["14", phylum[I_x] != phylum[S_x]]
vtest["15", order[I_x] == order[S_x]]

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

CreateOperator[{{I_x, I_y, I_z},{a, $a^\dagger$}}];

vtest["16", **Not**[Mult[a, $a^\dagger$] === $a a^\dagger$]]

```
vtest["17", phylum /@{$I_x$, $I_y$, $I_z$} === {1, 1, 1}]
vtest["18", phylum /@{$a$, $a^\dagger$} === {2, 2}]
vtest["19", order /@{$I_x$, $I_y$, $I_z$} === {1, 2, 3}]
vtest["20", order /@{$a$, $a^\dagger$} === {1, 2}]
```

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

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

3 Non-commutative multiplication

Listing 3: unidyn/Mult.m

First define bottom-out and empty cases:

```
Clear[Mult];
Mult[a_] := a
Mult[] := 1
```

Mult distributes over Plus and is associative:

```
Mult[a___,b_Plus,c___] :=
    Plus @@ (Mult[a, #, c] & ) /@ List @@ b
Mult[a___,Mult[b__,c__],d___] :=
    Mult[a,b,c,d]
```

Define rules for factoring out scalars:

```
Mult[a___, b_?ScalarQ, c___] :=
    Times[b, Mult[a,c]]
Mult[a___, b_?ScalarQ c__, d___] :=
    Times[b,Mult[a,c,d]]
```

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:

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

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 $\mathrm{Map}[\mathrm{phylum}[\#]\&, a]$ 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 $p_{\mathrm{new}} = \{3, 1, 2, 4\}$. We recognize p_{new} as the order in which the input operators should appear in a_{sorted}.

```
NCSort[a_List] :=
Module[{n, p, $a_{new}$, $p_{new}$},
    n = Length[a];
    p = (n+1) Map[phylum[#]&, a] + Table[i,{i ,1,n }];
    $p_{new}$=(Position[Sort[p],#] [[1,1]])&  /@ p;
    $a_{new}$= Table[0,{i,1,n}];
    Do[$a_{new}$[[$p_{new}$[[i]]]] = a[[i]],{i,1,n}];
    Return[$a_{new}$]]
```

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

```
(* SortedMult[a__] :=
    NonCommutativeMultiply[Sequence @@ NCSort[List[a]]] <<=== jam99 === *)

SortedMult[a__] :=
    Mult[Sequence @@ NCSort[List[a]]]
```

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

```
(* MultSort[a__] :=
    a /. NonCommutativeMultiply → SortedMult <<=== jam99 === *)

MultSort[a__] :=
    a /. Mult → SortedMult
```

Listing 4: unidyn/Mult–tests.m

First define some operators and scalars.

```
CreateOperator[{{$I_x$, $I_y$, $I_z$},{$S_x$, $S_y$, $S_z$}}];
CreateScalar[{a, b, c, d }];
```

In the following test, the left hand side should resolve to NonCommutativeMultiply$[I_x, I_y]$ 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[$I_x$,$I_y$] === $I_x I_y$]]
```

Before continuing, define a shorthand:

```
(* Mult = NonCommutativeMultiply; << ==== jam99 ==== *)
```

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[a, $b + c$] === $ab + a\,c$]
vtest ["02b", Mult[$a + b$, c] === $ac + b\,c$]

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] ab === ab Mult[I_y, I_x]]

vtest ["04a", Mult[a, $b + c$] === $ab + a\,c$]
vtest ["04b", Mult[$a + b$, c] === $ac + b\,c$]

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. Note that we do *not* have to call NCExpand[] for NonCommutativeMultiply[] to pull scalars out front.

vtest ["06a", Mult[I_x, 2 I a, S_x] === a2 I Mult[I_x,S_x]]
vtest ["06b", Mult[I_x, Mult[a, I_y], Mult[b, S_x]] === ab 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 NonCommutatativeMultiply[] 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 I_y, S_x, I_x]]
  === aMult[S_x,I_y, I_x]]

```
expr1 = Mult[$S_x$, a$S_z$] + Mult[$I_x$, b$S_x$];
expr2 = MultSort[Mult[expr1, $I_x$]];
expr3 = a Mult[$S_x$, $S_z$, $I_x$] + bMult[$S_x$, $I_x$, $I_x$];

vtest [“ 10a", expr2 === expr3]
```

4 Commutator

Listing 5: unidyn/Comm.m

Limiting cases:

```
(*
Comm[a_?NonCommutativeMultiply‘CommutativeQ, b_] := 0
Comm[a_, b_?NonCommutativeMultiply‘CommutativeQ] := 0
<<=== jam99 === *)

Comm[a_?ScalarQ, b_] := 0
Comm[a_, b_?ScalarQ] := 0
Comm[a_, a_] := 0
```

The commutator distributes over Plus:

```
Comm[a__, b_Plus] := Plus @@ (Comm[a, #] & ) /@ List @@ b
Comm[a_Plus, b__] := Plus @@ (Comm[#, b] & ) /@ List @@ a
```

Rules for factoring out scalars:

```
(*
Comm[a__ b_?NonCommutativeMultiply‘CommutativeQ, c__] := bComm[a, c]
Comm[a__, b_?NonCommutativeMultiply‘CommutativeQ c__] := bComm[a, c]
< ==== jam99 ==== *)

Comm[a__ b_?ScalarQ, c__] := bComm[a, c]
Comm[a__, b_?ScalarQ c__] := bComm[a, c]
```

Two rules for simplifying commutators involving products:

```
(*
Comm[a_NonCommutativeMultiply, C_] :=
   Module[{A,B},
      A = (List @@ a)[[1]] ;
      B = NonCommutativeMultiply @@ Rest[List @@ a] ;
      NonCommutativeMultiply[A, Comm[B, C]]
    + NonCommutativeMultiply[Comm[A, C], B]
    ]

Comm[A_, b_NonCommutativeMultiply] :=
   Module[{B,C},
            B = (List @@ b)[[1]] ;
            C = NonCommutativeMultiply @@ Rest[List @@ b] ;
            NonCommutativeMultiply[Comm[A, B], C]
          + NonCommutativeMultiply[B, Comm[A, C]]
        ]
    < ==== jam99 ==== *)

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

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

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

```
    Comm[a_,b_] := 0 /; phylum[a] != phylum[b];
```

Listing 6: unidyn/Comm–tests.m

```
    CreateScalar[{a, b, c, d}];
    CreateOperator[{{I_x, I_y, I_z},{S_x, S_y, S_z}}];
```

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[I_x, I_y+ I_z] === Comm[I_x, I_y] + Comm[I_x, I_z]]
    vtest ["01b", Comm[a, I_x] === 0]
    vtest ["01c", Comm[I_x, I_x] === 0]
    vtest ["01d", Comm[I_x, S_y] === 0]
    vtest ["01e", Comm[S_x, S_y] === Comm[S_x, S_y]]
```

Scalars are properly factored out of commutators.

```
    vtest ["02a", Comm[a S_x, S_y] === aComm[S_x, S_y]]
    vtest ["02b", Comm[S_x, bS_y] === bComm[S_x, S_y]]
    vtest ["02c", Comm[a S_x, bS_y] === abComm[S_x, S_y]]
```

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[a Mult[S_x, S_y], bS_z]
```

```
  === ab (Mult[$S_x$, Comm[$S_y$, $S_z$]] + Mult[Comm[$S_x$, $S_z$], $S_y$])]
vtest [“03b", Comm[a $S_x$, bMult[$S_y$, $S_z$]]
  === ab (Mult[$S_y$, Comm[$S_x$, $S_z$]] + Mult[Comm[$S_x$, $S_y$], $S_z$])]
vtest [“03c", Comm[$S_x$, Mult[$S_x$, $S_y$]]
  === Mult[$S_x$, Comm[$S_x$, $S_y$]]]
```

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[$I_x$, $I_y$] //. Comm[$S_x$_, $S_y$_] ⟶ Mult[$S_x$, $S_y$] – Mult[$S_y$, $S_x$])
  === Mult[$I_x$, $I_y$] – Mult[$I_y$, $I_x$]]

vtest [“04b", ((Comm[$I_x$, Comm[$I_y$, $I_z$]] +
    Comm[$I_y$, Comm[$I_z$, $I_x$]] + Comm[$I_z$, Comm[$I_x$, $I_y$]])
   //. Comm[$S_x$_, $S_y$_] ⟶ Mult[$S_x$, $S_y$] – Mult[$S_y$, $S_x$])
  === 0]
```

For other Commutator identities, see https://en.wikipedia.org/wiki/Commutator. Let us test one of these advanced identities.

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

vtest [“05a", Comm[Mult[A, B, C], D]
  === Mult[A, B, Comm[C, D]]
  + Mult[A, Comm[B, D], C] + Mult[Comm[A, D], B, C]]
```

5 Spins

Listing 7: unidyn/Spins.m

```
SpinSingle$CreateOperators[$I_x$_,$I_y$_,$I_z$_,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 = Or @@ (Mult‘CommutativeQ /@ {$I_x$, $I_y$, $I_z$});
< ==== jam99 === *)

nonexistent =
    Not[OperatorQ[$I_x$]] ||
    Not[OperatorQ[$I_y$]] ||
    Not[OperatorQ[$I_z$]];

 If [nonexistent == True,
    Clear[$I_x$, $I_y$, $I_z$];
        CreateOperator[{{$I_x$, $I_y$, $I_z$}}];
        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.

```
    $I_x$ /:  Comm[$I_x$,$I_y$] = I $I_z$;
    $I_x$ /:  Comm[$I_x$,$I_z$] = −I $I_y$;
    $I_y$ /:  Comm[$I_y$,$I_x$] = −I $I_z$;
    $I_y$ /:  Comm[$I_y$,$I_z$] = I $I_x$;
    $I_z$ /:  Comm[$I_z$,$I_y$] = −I $I_x$;
    $I_z$ /:  Comm[$I_z$,$I_x$] = I $I_y$;

    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;
    $I_y$ /:  Mult[a___,$I_y$,$I_y$,b___] := Mult[a,b]/4;
    $I_x$ /:  Mult[a___,$I_x$,$I_x$,b___] := Mult[a,b]/4;

    $I_z$ /:   Mult[a___,$I_z$,$I_x$,b___] := I Mult[a,$I_y$,b]/2;
    $I_z$ /:   Mult[a___,$I_z$,$I_y$,b___] := −I Mult[a,$I_x$,b]/2;

    $I_y$ /:   Mult[a___,$I_y$,$I_x$,b___] := −I Mult[a,$I_z$,b]/2;
    $I_y$ /:   Mult[a___,$I_y$,$I_z$,b___] := I Mult[a,$I_x$,b]/2;

    $I_x$ /:   Mult[a___,$I_x$,$I_z$,b___] := −I Mult[a,$I_y$,b]/2;
    $I_x$ /:   Mult[a___,$I_x$,$I_y$,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[{$I_x$, $I_y$, $I_z$}] (* <<==== IMPORTANT *)
    ]
```

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", Mult[Ix, Iy] === Mult[Ix, Iy]]
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[$I_x$, $I_y$] === I $I_z$]
vtest ["03b", Mult[$I_x$, $I_y$] === 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 = $I_x$;
rho2 = Comm[−I h, Comm[−I h,rho0]];
vtest ["04a", rho2 == − Mult[w, w, $I_x$]]
```

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

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

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

6 Harmonic Oscillator

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$_, $a^\dagger$_] :=

Module[{nonexistent},
```

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

```
(*
nonexistent = Or @@ (OperatorQ /@ {$a$, $a^\dagger$});
< === jam99 *)

nonexistent =
  Not[OperatorQ[$a$]] ||
    Not[OperatorQ[$a^\dagger$]];

 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†}] (* <<==== IMPORTANT *)
    ]
```

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†, a, Nop];

CreateScalar[{a}];
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[a a, a†] === a]
vtest ["01e", Comm[a, aa†] === a]
```

Test the commutation relations by defining the number operator, Nop = $N = a^{\dagger}a$, and checking the commutation relations $[N, a^{\dagger}] = a^{\dagger}$ and $[N, a] = -a$.

```
Nop = Mult[a†, a];

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

Clear[a, a†, a, 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];
```

7 Unitary Evolution

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 to coerce *Mathematica* into writing simpler looking expressions, from http://mathematica.stackexchange.com/questions/5403/how-to-get-fullsimplify-to-fully-simplify-my-expression-with-custom-complexity-f

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

The Evolver function follows. By default, the function will not print out intermediate results during the computation.

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

Evolver[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. Here we use the ability of the `NCALgebra` package to compute a symbolic inverse of an operator. 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 list are Hermitian or not.

```
a$vect={0,0,0,0};
r = Null;
r = Catch[
  Do[
    q = Mult[
        Divide[ρ[3],Mult[ρ[k]]]   (* Divide is a kludge *)
      ] // 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[Evolver::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]];
  ];
```

Listing 12: unidyn/Evolve–tests.m

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

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

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]]]
```

Double check that NCExpand[] bottoms out propertly when presented an operator and a product of scalars and operators.

```
(*
vtest ["01d > NCExpand bottom-out test 1", NCExpand[Q] === Q]
vtest ["01e > NCExpand bottom-out test 2", NCExpand[Q q r] === q r Q]
vtest ["01f > NCExpand bottom-out test 3", NCExpand[Mult[U, S] w v] === v w Mult[U, S]]
< === jam99 === *)
```

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]
```

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[I_x, I_y, I_z, S_x, S_y, S_z, ω, d, Δ, ρ, t]
Clear[A, r, lhs$list , rhs$list , time, eqns, system, ρ_{calc}, ρ_{known}, X]

CreateScalar[ω, d, Δ];
CreateOperator[{{I_x, I_y, I_z},{S_x, S_y, S_z}}]

SpinSingle$CreateOperators[$I_x$, I_y, I_z, L=1/2];
SpinSingle$CreateOperators[$S_x$, S_y, S_z, 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, IyΔ, -Ix Δ^2, -Iy Δ^3, IxΔ^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 ["04a > DSolve test", ρcalc=== ρknown]
```

Free evolution of I_x:

```
vtest ["05a > free evolution of Ix test ",
  FullSimplify [ExpToTrig[Expand[
   Evolver[ω Iz, t, Ix ]]]]
 === Ix Cos[ω t] + Iy Sin[ω t]]

vtest ["05a > free evolution of Ix ",
  FullSimplify [ExpToTrig[Expand[
   Evolver[ω Iz, t, Ix, quiet → quiet$query]]]]
 === Ix Cos[ω t] + Iy Sin[ω t]]
```

On-resonance nutation of I_z:

```
vtest ["05b > on-resonance nutation of Iz",
  FullSimplify [ExpToTrig[Expand[
   Evolver[ω Ix, t, Iz, quiet → quiet$query]]]]
 === Iz Cos[ω t] - Iy Sin[ω t]]
```

Free evolution of I_+:

```
vtest ["05c > free evolution of I+",
  FullSimplify [Evolver[ω Iz, t, Ix, quiet → quiet$query]
   + I Evolver[ω Iz, t, Iy, quiet → quiet$query]]
 === Exp[-I ωt](Ix + I Iy)]
```

Evolution under a scalar coupling:

```
vtest ["05d > scalar-coupling evolution of Ix ",
 Evolver[d Mult[Iz, Sz], t, Ix, quiet → quiet$query]
 === Ix Cos[d t/2] + 2 Mult[Iy, Sz] Sin[d 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[
  c1Iz+ c2Ix+ (c3 Iz− c2Ix) Cos[ωeff t] − c4IySin[ωeff t] //
    Expand, {Ix, Iy, Iz}];

ρcalc= Collect[
  Evolver[Δ Iz+ ωIx , 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, d, 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 ["06a1 > free evolution of lowering operator",
  Simplify [TrigToExp[Expand[Evolver[ℋ, t, a, quiet  → quiet$query]]]]
    === a Exp[I ω t ]]

vtest ["06a2 > free evolution of raising operator",
  Simplify [TrigToExp[Expand[Evolver[ℋ, t, a†, quiet  → quiet$query]]]]
    === a†Exp[−I ωt]]
```

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["06b > test Q definition ", Simplify[𝑄 /. QP$rules] === Q]
vtest["06c > test P definition ", Simplify[𝑃 /. QP$rules] === P]
```

Now we are ready to evolve position and momentum.

```
vtest["06d > free evolution of Q",
  Simplify[ExpToTrig[Expand[Evolver[ℋ, t, Q, quiet → quiet$query] /. QP$rules]]]
    == Q Cos[ω t] - P Sin[ω t]]

vtest["06e > free evolution of P",
  Simplify[ExpToTrig[Expand[Evolver[ℋ, t, P, quiet → quiet$query] /. QP$rules]]]
    == P Cos[ω t] + Q Sin[ω t]]
```

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

```
vtest["06f > position kick",
  Simplify[Evolver[δq P, t, Q] /. QP$rules ~Join~ {t → 1}]
   == Q - δq]

vtest["06g > momentum kick",
  Simplify[Evolver[δp Q, t, P] /. QP$rules ~Join~ {t → 1}]
   == P + δp]
```

Clean up:

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

References

[1] C. P. Slichter, *Principles of magnetic resonance, 3rd edition*, 1990.

[2] 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.

[3] J. W. Helton, M. de Oliveira, M. Stankus, and R. L. Miller, Ncalgebra — version 5.0.6, 2015.