**Ver. 1.0.0**

**MATLAB® Program for Product Operator Formalism of Spin-1/2**

Kosuke Ohgo

URL: <https://github.com/ohgo1977/ProductOperator>

Email: please add @gmail.com to my GitHub account name

**Purpose**

This program is designed to handle the product operator formalism of spin-1/2 using MATLAB. The calculation is based on the cyclic commutation rules thus it is faster than the actual matrix calculation. The program can handle a multiple-spin system with the flexibility of labeling of spins. This program will be helpful for educational use, for example, showing how to calculate product operators and explaining how components of pulse sequences (Hahn-echo, INEPT etc.) work. Also, this program can be used to calculate an evolution of a density operator under a pulse sequence including phase cycling.

**Requirement**

MATLAB and MATLAB Symbolic Math Toolbox are required to run this program. It is tested under the MATLAB R2020b environment. Basic knowledge of MATLAB programming and Symbolic toolbox is required.

**Limitation**

This program can only handle weakly coupled spin-1/2 systems as same as the product operator formalism.

**Design of the Programming and How to Use It**

The code is written with the object-oriented programming (OOP) style. In manner of the OOP, parameters for constructing product operators and functions for NMR interactions are described as **properties** and **methods** of a **class** named **PO**, respectively.

**PO Class Properties**

**PO** class properties store the status of the current system. Any product operator can be described by two properties, axis labels (x, y or z) and a coefficient. For example, in the case of -4IxSzKz\*cos(q), the axis labels are x, z and z for the 1st (I), 2nd (S) and 3rd (K) spins, respectively and the coefficient is -cos(q). Note that “4” is related to the number of the active spin types in the system (in this case the number of the active spin types is Ns = 3 and 4 is obtained from 2Ns – 1) and thus it is not considered as an independent coefficient.

In the **PO** class, information on the axis label and coefficient are stored as **axis** and **coef** properties, respectively.

**axis**: This property stores axis labels of product operators. It is a *M* x *N* matrix where *M* is a number of product operators in a system and *N* is a number of the spin types. Each component in the matrix has a value of 0, 1, 2, 3 corresponding to 1-, x-, y- and z-operators, respectively. For example, Ix + Sx + Sy (*M* = 3) in the IS system (*N* = 2) has [1 0; 0 1; 0 2] as **axis**.

**coef**: This property stores coefficients including signs for product operators. It is a *M* x 1 vector. Note that the 2Ns – 1 coefficient at the beginning of a product operator is stored in other property, **Ncoef**.

There are additional properties in the **PO** class.

**spin\_label**: This property defines labels of the spins in the current system such as, I,S,K, … or I1, I2, I3, … etc. The default labels are I, S, K, L, M.

**Ncoef**: This property stores the 2Ns - 1 coefficients for the product operators in the system. It is automatically calculated from the axis property.

**txt**: This property stores a text output of the current system. It is automatically generated.

**M**: This property stores a matrix representation of the current system. It is automatically generated.

**sqn**: This property stores a spin quantum number. As a default, it stores sym(1/2) for spin-1/2.

**disp**: This property stores 1 or 0 to control the display of the applied method and result on the command window. The default value is 1 for display ON.

**coherence**: This property is a 2*N* x 2*N* matrix displaying populations of spin states on the diagonal and coherences between states on the off-diagonal. **a** and **b** mean |α> and |β> states, respectively, and **m** and **p** mean coherences of |α> => |β> and |β> => |α>, respectively.

As a note, values of the **PO** properties can be obtained by a syntax **obj.PropertyName**. For example, to get a matrix representation,

>> rho\_matrix = rho.M;

In the current version, it is possible to overwrite many of properties directly from the command line, for example,

>> rho.axis = [1 2 3];

This type of operation is not expected except for the **disp** property, and it is not recommended to do this.

**PO Class Methods**

**1. Creating an Initial State of a System (Class Constructor)**

An initial state of a system, i.e., a density operator at the beginning, can be created in two different ways using the **PO** methods.

The first method is the use of the class constructor **PO()**. As an example, if you like to construct ρ = Ix\*cos(q) + 2IySz\*sin(q),

>> syms q;

>> rho = PO(2, {'Ix' 'IySz'},{cos(q) sin(q)},{'I' 'S'})

rho =

PO with properties:

axis: [2×2 double]

coef: [2×1 sym]

spin\_label: {'I' 'S'}

disp: 1

coherence : [4×4 sym]

Ncoef: [2×1 sym]

txt: 'Ix\*cos(q) + 2IySz\*sin(q)'

M: [4×4 sym]

sqn: [1×1 sym]

The general syntax of the constructor is

**obj = PO(spin\_no, sp\_cell, coef\_cell, spin\_label\_cell)**

where **obj** is a **PO**-class object, **spin\_no** is the number of spin types in the system, **sp\_cell** is a cell array for product operators (without 2Ns – 1 coefficients), **coef\_cell** is a cell array for coefficients (coefficients can be given by double, char or sym class) and **spin\_label\_cell** is a cell array for spin labels that will be used as the class property **spin\_label**.

There are some examples showing how to use the **PO** constructor.

ρ = Ix + Sy

>> rho = PO(2, {'Ix' 'Sy'});

If only the first two parameters are given, the third, **ccoef\_cell**, is automatically set as {sym(1) sym(1)} and the fourth, **spin\_label\_cell**, is set as {'I' 'S'} that is from the 1st and 2nd components of the default cell array {'I' 'S' 'K' 'L' 'M'}. If **spin\_no** is given as 3, {'I' 'S' 'K'} is used instead of {'I' 'S'}. Note that if **spin\_label\_cell** is not given explicitly, spin types used for **sp\_cell** should be selected from the first **spin\_no** components of {'I' 'S' 'K' 'L' 'M'}.

ρ = a\*I1x + b\*I2y

>> rho = PO(2, {'I1x' 'I2y'},{'a' 'b'},{'I1' 'I2'});

Since I1 and I2 are used instead of I and S, it is necessary to input {'I1' 'I2'} as **spin\_label\_cell.** To assign the coefficients a and b, text inputs can be used in **coef\_cell**. Note that the format of each label should be one letter or two characters with a letter and a number. For example, 'SP' or 'I10' is not allowed to use as a label.

ρ = Ix\*cos(θ) + Iy\*sin(θ)

>> syms q;

>> rho = PO(1,{'Ix' 'Iy'},{cos(q) sin(q)});

The coefficients cos(θ) and sin(θ) can be described by a symbolic class. In this case, create q as a symbolic class and give cos(q) and sin(q) to **coef\_cell**.

ρ = I1x in a 3-spin system (I1, I2 and I3)

>> rho = PO(3, {'I1x'},{1},{'I1' 'I2' 'I3'});

It is necessary to set the number of total spin types in the system by the constructor. **spin\_label\_cell** should be set for this particular 3-spin system.

ρ = Ix + 4IxSyKz

>> rho = PO(3, {'Ix' 'IxSyKz'});

The 2Ns - 1 coefficient, 4 in 4IxSyKz, should not be included in both **sp\_cell** and **coef\_cell**.

ρ = I+S-Kz

>> rho = PO(3, {'IpSmKz'});

**PO** methods can handle the raising/lowering operator basis (**pmz** basis). In this basis, '**p**', '**m**' and '**z**' can be used where '**p**' and '**m**' are the raising and lowering operators, respectively. The Cartesian basis (**xyz** basis) and **pmz** basis cannot be used in a single **PO** object simultaneously. In the case of the **pmz** basis, **Ncoef** is set as 1. The values for the **axis** property are 4 for '**p**' and 5 for '**m**'. To convert the basis from **xyz** to **pmz**, **xyz2pmz()** can be used. Reversely, **pmz2xyz()** is for the conversion from **pmz** to **xyz** basis. They are explained in the later section.

As a special case,

>> rho = PO(1,{'1'});

creates a 1/2E operator with rho.axis = [0], rho.coef = [1], rho.Ncoef = [1/2] and rho.txt = '1/2E' and rho.M = [1/2 0; 0 1/2]. Actually, any types of characters that is not defined in **spin\_label** are considered as 1/2E operator.

The second method is to use preset spin operators to construct a density operator by their combinations. The static **PO** method **PO.create(spin\_label\_cell)** creates spin operators with the labels from **spin\_label\_cell**. For example,

>> PO.create({'I' 'S'})

creates Ix, Iy, Iz (xyz basis), Ip, Im, Sx, Sy, Sz (xyz basis), Sp, Sm, and hE (1/2E, xyz basis) in the workspace. Then it is possible to create a desired density operator by using the operators ‘\*’, ‘+’, ‘-‘ and ‘/’ . To create ρ = Ix\*cos(θ) + 2IySz\*sin(θ),

>> rho = Ix\*cos(q) + 2\*Iy\*Sz\*sin(q);

Note that spin operators with the xyz and pmz bases cannot be used together to create a product operator, such as Ix\*Ip. The exempts are Iz and hE. If it is necessary to calculate Ix\*Ip, then use Ix\*pmz2xyz(Ip) (xyz basis) or xyz2pmz(Ix)\*Ip (pmz basis).

The first method will be useful in a script describing a pulse sequence. The second method will be helpful for demonstrations of the calculations of product operators in classes.

**2. Applying NMR Interactions to a System**

There are **PO**-class methods for NMR interactions i.e., RF pulse, chemical shift and *J*-coupling to apply a spin system.

**RF Pulses**

**Single Pulse**

The method to apply a single pulse is

**obj = pulse(obj, sp, ph, q)** or

**obj = obj.pulse(sp, ph, q)** .

where **obj** is a **PO**-class object, **sp** is a type of spin for the pulse, **ph** is a quadrature phase, and **q** is a flip angle in radian.

**sp** can be characters ('I', 'S', 'I1' or 'I2' etc. defined in **spin\_label**) or the number of the order of the spins (1 for 'I', 2 for 'S' etc.). **ph** can be characters such as 'x', 'X' or '-y' or numbers 0, 1, 2 or 3 for x, y, -x or -y, respectively.

**q** can be double or symbolic class, such as pi/2 (double) or syms q (symbolic).

Examples

>> rho = PO(1, {'Iz'});

>> rho = pulse(rho, 'I', 'x', pi/2);

or equivalently,

>> rho = rho.pulse(1, 0, pi/2);

**Simultaneous Pulses**

The method to apply simultaneous pulses is

**obj = simpulse(obj, sp\_cell, ph\_cell, q\_cell)** or

**obj = obj.simpulse(sp\_cell, ph\_cell, q\_cell)** .

**sp\_cell**, **ph\_cell** and **q\_cell** are cell arrays corresponding to **sp**, **ph** and **q** for **pulse()**, respectively.

>> rho = PO(2, {'Iz' 'Sz'});

>> rho = simpulse(rho, {'I' 'S'}, {'x' 'y'}, {pi/2 pi/2});

It applies a 90x pulse to I and 90y to S. Equivalently,

>> rho = simpulse(rho, {1 2}, {0 1}, {pi/2 pi/2});

The wildcard character **'\*'** can be used for **sp\_cell** to make a command simpler.

Let's assume a 5-spin system, I1, I2, I3, S4 and S5.

>> rho = PO(4,{'I1z' 'I2z' 'I3z' 'S4z' 'S5z'},{sym(1) sym(1) sym(1) sym(1) sym(1)},{'I1' 'I2' 'I3' 'S4' 'S5'});

If applying a 90x pulse to all I spins, the wildcard character can be used as

>> rho = simpulse(rho, {'I\*'},{'x'},{pi/2});

If applying 90x pulses to all spins,

>> rho = simpulse(rho, {'\*'},{'x'},{pi/2});

If applying a 90x pulse to all I spins and a 180y pulse to S spins,

>> rho = simpulse(rho, {'I\*' 'S\*'},{'x' 'y'},{pi/2 pi});

**Pulses with Phase Shift**

The method to apply a single pulse with a phase shift is

**obj = pulse\_phshift(obj, sp, ph, q)** or

**obj = obj.pulse\_phshift(sp, ph, q)** .

The difference from **pulse()** is that **ph** is an arbitrary phase in radian. **ph** can be a double or symbolic class.

Accordingly, simultaneous pulses with phase shifts can be applied by

**obj = simpulse\_phshift(obj, sp\_cell, ph\_cell, q\_cell)** or

**obj = obj.simpulse\_phshift(sp\_cell, ph\_cell, q\_cell)** .

**Chemical Shift**

The method to apply a chemical shift evolution is

**obj = cs(obj, sp, q)** or

**obj = obj.cs(sp, q)** ,

where **sp** is a type of the spin and **q** is a rotation angle in radian. The formats of **sp** and **q** are same as the ones used for **pulse()**.

It is possible to obtain chemical shift evolutions of multiple spins with

**obj = simcs(obj, sp\_cell, q\_cell)** or

**obj = obj.simcs(sp\_cell, q\_cell)**

where **sp\_cell** and **q\_cell** have the same formats as the one used for **simpulse()**. The wildcard character **'\*'** also can be used for **sp\_cell** same as **simpulse()**.

>> syms oI oS t

>> rho = PO(3, {'I1x' 'I2x' 'S3x'},{1 1 1},{'I1' 'I2' 'S3'});

>> rho = cs(rho,'I1',oI\*t);

>> rho = cs(rho,'I2',oI\*t);

>> rho = cs(rho,'S3',oS\*t);

or equivalently,

>> rho = simcs(rho,{1 2 3},{oI\*t oI\*t oS\*t});

or

>> rho = simcs(rho,{'I\*' 'S3'},{oI\*t oS\*t});

***J*-coupling**

The method to apply a *J*-coupling evolution is

**obj = jc(obj, sp, q)** or

**obj = obj.jc(sp, q)** ,

where **sp** is labels of a spin pair and **q** is a rotation angle in radian.

**sp** can be characters 'IS', 'I1I3' etc. or a 1 x 2 vector showing the index of spins such as [1 2] or [1 3].

>> rho = PO(2,{'Ix'});

>> syms J12 t

>> rho = jc(rho,'IS',pi\*J12\*t);

It is possible to obtain *J*-coupling evolutions of multiple spin-pairs by

**obj = simjc(obj, sp\_cell, q\_cell)** or

**obj = obj.simjc(sp\_cell, q\_cell)** ,

where **sp\_cell** is a cell array of spin pairs. Note that the wildcard character **'\*'** can not be accepted for **sp\_cell** in this method. The spin pairs must be explicitly given in **sp\_cell**.

>> rho = PO(3,{'I1x' 'I2x'},{1 1},{'I1' 'I2' 'S3'});

>> syms J13 J23 t

>> rho = simjc(rho,{'I1S3' 'I2S3'},{pi\*J13\*t pi\*J23\*t});

**Pulse Field Gradients**

The method to apply a pulse field gradient is

**obj = pfg(obj, G, gamma\_cell)** or **obj = obj.pfg(G, gamma\_cell)**.

**G** is a strength of the gradient field and **gamma\_cell** is a cell array to store gyromagnetic ratio of spins in the system.

**G** can be double or symbolic class. Components of **gamma\_cell** can be also double or symbolic class.

>> syms G gH gC

>> rho = PO(3,{'I1x' 'I2x' 'S3x'},{1 1 1},{'I1' 'I2' 'S3'});

>> rho = pfg(rho, G, {gH gH gC});

Internally, angles are calculated from **G** and **gamma\_cell** as their products and they are used as input parameters of **simcs()**. In the case above, the angles are {G\*gH G\*gH G\*gC}. Note that a length of the gradient pulse is not considered in the calculation. If necessary, involve a time constant into **G** (e.g. G\*t). This method was inspired by a reference (Güntert, 2006).

**Applying Multiple Methods in One Line**

Since the program is written with the OOP style, you can apply multiple methods in one line using the dot **'.'** as the separator between the methods,

>> rho = rho.pulse('I','y',pi/2).cs('I',q).jc('IS',pi\*J12\*t);

that is equivalent to

>> rho = rho.pulse('I','y',pi/2);

>> rho = rho.cs('I',q);

>> rho = rho.jc('IS',pi\*J12\*t);

Note that the order to be applied to the system is left to right (**pulse()** => **cs()** => **jc()**) but not right to left (**jc()** => **cs()** => **pulse()**).

**3. Utilities**

There are several methods as utilities. Some of them are static methods, i.e., they are called by **PO.MethodName()**. These utilities can be used, for example, when a pulse sequence is simulated.

**obj\_pmz = xyz2pmz(obj\_xyz)** or

**obj\_pmz = obj\_xyz. xyz2pmz()**

converts ρ from Cartesian operator basis (**obj\_xyz)** to raising/lowering operator basis (**obj\_pmz**) (Güntert, 2006). All methods for NMR interactions (**pulse()**, **cs()**, **jc()** etc.) are compatible with **obj\_pmz**.

**obj\_xyz = pmz2xyz(obj\_pmz)** or

**obj\_xyz = obj\_pmz. pmz2xyz()**

converts ρ from raising/lowering operator basis (**obj\_pmz**) to Cartesian operator basis (**obj\_xyz)** (Güntert, 2006).

**phout = PO.phmod(phx, ii)**

outputs a phase value, **phout**, from a phase table (vector), **phx**. If **ii** is smaller or equal to length(phx), phout = phx(ii) otherwise phout = phx(mod(ii,length(phx)). This method can be used in cases where there are phase tables with different lengths and phase-cycle them together.

Example

ph1 = [0 1 2 3]; % 4 steps

ph2 = [0 1 2 3 2 3 0 1]; % 8 steps

To phase-cycle them together, 8 steps are necessary. In such case, **PO.phmod(ph1,ii)** returns

ii = 1 🡺 0, ii = 2 🡺 1, ii = 3 🡺 2, ii = 4 🡺 3, ii = 5 🡺 0, ii = 6 🡺 1, ii = 3 🡺 2, ii = 8 🡺 3

while **PO.phmod(ph2,ii)** returns

ii = 1 🡺 0, ii = 2 🡺 1, ii = 3 🡺 2, ii = 4 🡺 3, ii = 5 🡺 2, ii = 6 🡺 3, ii = 0 🡺 2, ii = 8 🡺 1 .

**dispPOtxt(obj)** or

**obj.dispPOtxt()**

displays the **txt** property of **obj** to the Command Window.

**dispPO(obj)** or

**obj.dispPO()**

displays terms in **obj** in the following manner.

ID Product-Operator Coefficient

For example, in the case of ρ = Ix\*cosθ + Iy\* sinθ

>> syms q; rho = PO(1,{'Ix' 'Iy'},{cos(q) sin(q)});

>> dispPO(rho)

1 Ix cos(q)

2 Iy sin(q)

**id\_vec = findcoef (obj, coef\_in\_cell)** or

**id\_vec = obj. findcoef (coef\_in\_cell)**

finds particular terms that includes the coefficients defined in **coef\_in\_cell** and returns index numbers for these terms as **id\_vec**.

In the case of the example above,

>> id\_vec = rho.findcoef({cos(q)});

returns id\_vec = 1.

**obj = delPO(obj, id\_in)** or

**obj = obj.delPO(id\_in)**

Delete particular terms from **obj** using the information given by **id\_in**.

If **id\_in** is a vector including numbers, these numbers are indexes for terms to be deleted. The ID number of each term can be found by **obj.dispPO()** or can be obtained from **obj.findcoef()**. The example is

>> rho = PO(3,{'Ix' 'Sx' 'Kx'});

>> rho.dispPO()

1 Ix 1

2 Sx 1

3 Kx 1

>> rho = delPO(rho,[1 2])

>> rho.dispPO()

1 Kx 1

If **id\_in** is a cell array with product operators, terms including these operators are deleted.

There are some examples.

>> rho = delPO(rho,{'Ix'}); % delete Ix term

>> rho = delPO(rho,{'IzSz'}) % delete 2IzSz term

>> rho = delPO(rho,{'Ix' 'IzSz'})% delete Ix and 2IzSz term

The wildcard character **'\*'** can be used to choose all three phases.

>> rho = delPO(rho,{'IxS\*'})% delete 2IxSx, 2IxSy and 2IxSz if they exist

>> rho = delPO(rho,{'I\*S\*' 'I\*S\*K\*'})% delete any terms including 2I\*S\* and 4I\*S\*K\*

**obj = selPO(obj, id\_in)** or

**obj = obj.delPO(id\_in)**

Select particular terms from **obj** using the information given by **id\_in**. The format of **id\_in** is same as in **delPO()**.

**[a0\_V,rho\_V] = SigAmp(obj, sp\_cell, phR)** or

**[a0\_V,rho\_V] = obj.SigAmp(sp\_cell, phR)**

Calculation of signal amplitudes *a* corresponding to (-1) quantum coherences in the equation

*a*[-] = 2\*i\*ρ[-](0) \*exp(-i\*φrec)

in Spin Dynamics, p. 288 (eq. 11.48).

For example, in the case of homonuclear 2-spin system, the equation is

[*a*[-β] *a*[-α] *a*[β-] *a*[α-]] = 2\*i\*[ρ[-β](0) ρ[-α](0) ρ[β-](0) ρ[α-](0)]\*exp(-i\*φrec)

shown in p. 379.

Related topics: Spin Dynamics (2nd Ed.), p.262, p. 287, p. 371, p.379, pp.608-610.

**sp** describes spin types to be observed (e.g., 'I', 'IS', 'I1I2', [1], [1 2]). **phR** is a quadrature receiver phase (e.g. 'x', 'y', 0, 1).

**a0\_V** is a vector corresponding to 2\*i\*[ρ[-β…](0) ρ[-α…](0) ρ[β-…](0) ρ[α-…](0) …] \*exp(-i\*φrec).

**rho\_V** is a vector describing which component of **a0\_V** comes from which coherence in the density operator (e.g., 'ma' for '-α', 'bm' for 'β-').

**dispProp(obj, PropertyName)** or

**obj.dispProp(PropertyName)**

displays a property **PropertyName** of **obj** to the Command Window. This method is useful to check the protected properties which is invisible from the Command Window. For example,

>> dispProp(obj, 'basis')

displays the property 'basis' which is internally used to distinguish the basis used.

**Technical Details**

**Description of a product operator in the code**

In the code, a product operator is described as a 1 x *N* vector where *N* is a number of spin types in a system. The axis label of the product operator is described as 1 for x, 2 for y, and 3 for z, and 0 for 1/2\**E*. For example, Ix and IzSz in the IS system are described as [1 0] and [3 3], respectively. The intrinsic coefficient 2Ns - 1 (e.g., 2 for 2IzSz) is calculated automatically and is stored separately. In the case of multiple terms, 1 x *N* vectors are combined to a *M* x *N* matrix where *M* is a number of the terms. A coefficient of each term is stored separately.

**Cyclic commutation rules**

If operators *A*, *B*, and *C* shows [*A*, *B*] = i*C*, [*B*, *C*] = i*A* and [*C*, *A*] = i*B* (cyclic commutation) then

exp(-iθ*A*) *B* exp(iθ*A*) = *B* cosθ + *C* sinθ,

exp(-iθ*B*) *C* exp(iθ*B*) = *C* cosθ + *A* sinθ, and

exp(-iθ*C*) *A* exp(iθ*C*) = *A* cosθ + *B* sinθ

It is known that the spin angular momentum operators Ix, Iy, Iz are in the cyclic commutation ([Iz, Ix] = iIy) and the formula above can be used to describe an evolution of a density operator under a Hamiltonian. For example, a density operator ρ(0) = Ix evolves under a chemical shift Hamiltonian *H* = ωIz during a time period of t as

ρ(t) = exp(-i*H*t) ρ(0) exp(i*H*t) = exp(-iωtIz) Ix exp(iωtIz) = Ix cos(ωt) + Iy sin(ωt)

**Use of the master table to accelerate calculation**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  | B |  |
|  |  |  | x | y | z |
|  |  |  | 1 | 2 | 3 |
|  | x | 1 | **0** | **3** | **-2** |
| A | y | 2 | **-3** | **0** | **1** |
|  | z | 3 | **2** | **1** | **0** |
|  |  |  |  | C |  |

The cyclic commutation rules can be summarized as a table (master table) using the equation exp(-iθ*A*) *B* exp(iθ*A*) = *B* cosθ + *C* sinθ above and Ix, Iy and Iz. For example, in the case of exp(-iθIz) Ix exp(iθIz) = Ix cosθ + Iy sinθ, the axis numbers of the *A* and *B* positions are 3 (z) and 1 (x), respectively. Then the axis number for *C* is the (3, 1) component in the table, i.e., 2 meaning Iy. If the value for *C* is 0 for given *A* and *B*, then *A* and *B* are not in the cyclic commutation (e.g., *A* = *B* = Ix). If the value for *C* is negative, then -*C* sinθ is used instead of +*C* sinθ.

This is the basic idea of the calculation in the code. This process does not handle a matrix calculation which usually has a high calculation cost.

**When can the master table be used for the calculation?**

If the two rules below are satisfied, an evolution of ρ under *H* can be calculated with using the master table. Otherwise ρ does not evolve under *H*.

**Rule 1**. There should be at least one spin type matching between *H* and ρ.

AND

**Rule 2**. Only one spin type in the matching spin types has different axis labels between *H* and ρ.

These rules can be used for spin-1/2 with *H* that is a product of up to two spin operators (e.g. *H* = 2IzSz but not like 4IzSzKz).

The rule 1 is obvious but how about the rule 2. Here is the analysis. Suppose *H* = 2IaSb and ρ = 8IbSbKL where K, L are product operators which are different types each other in addition to I and S. Suppose [Ia, Ib] = iIc in the cyclic commutation. There are two spin-types matching between *H* and ρ (thus satisfying the Rule 1) and only one of them (I spin) has different labels between *H* and ρ (thus satisfying the Rule 2).

Then [*H*, ρ] = 2IaSb 8IbSbKL – 8IbSbKL 2IaSb = 16 IaIbSb2KL - 16IbIaSb2KL

= 16 (IaIb – IbIa)Sb2KL = 4(IaIb – IbIa)KL = i4IcKL. Note that Sb2 = 1/4E for spin-1/2.

[4IcKL, *H*] = 4IcKL 2IaSb - 2IaSb 4IcKL = 8IcIaSbKL - 8IaIcSbKL = 8(IcIa – IaIc)SbKL = i8IbSbKL = iρ.

[ρ, 4IcKL] = 8IbSbKL 4IcKL - 4IcKL 8IbSbKL = 32IbIcSbK2L2 – 32IcIbSbK2L2 = 2(IbIc – IcIb)Sb = i2IaSb = i*H*. Note that K2 = L2 = 1/4E for spin-1/2.

Thus, *H* and ρ are in the cyclic commutation.

What if the rule 2 is not satisfied?

In the case of *H* = 2IbSb and ρ = 8IbSbKL, [*H*, ρ] = 2IbSb 8IbSbKL – 8IbSbKL 2IbSb = 16 Ib2Sb2KL - 16Ib2Sb2KL = 0

thus, *H* and ρ are not in the cyclic commutation.

In the case of *H* = 2IaSb and ρ = 8IbScKL, [*H*, ρ] is calculated as 0 from [2IaSb, 2IbSc] = 0 that can be obtained from the matrix representation.

[*H*, ρ] = 2IaSb 8IbScKL - 8IbScKL 2IaSb = 2IaSb 2IbSc 4KL - 2IbSc 4KL 2IaSb = 2IaSb 2IbSc 4KL - 2IbSc 2IaSb 4KL = [2IaSb, 2IbSc]\*4KL = 0 (Read *Spin Dynamics* (2nd Ed.), p.403, Eq. 15.24 and p. 407, Note 3).

Examples for these rules

ρ = Iy and *H* = Iz

Both ρ and *H* include I-spin (Rule 1: yes) and they have different axis labels (Iy vs. Iz) (Rule 2: yes). 🡺 Master Table: Yes

ρ = Sy and *H* = Iz

ρ and *H* don't have a same type of spin (Rule 1: No) 🡺 Master Table: No

ρ = Iy and *H* = Iy

Both ρ and *H* include I-spin (Rule 1: yes) but they have same axis labels (Iy) (Rule 2: no). 🡺 Master Table: no

ρ = 2IzSy and *H* = Iz

Both ρ and *H* include I-spin (Rule 1: yes) but I-spins have same axis labels (Iz) (Rule 2: no). 🡺 Master Table: no

ρ = 2IzSy and *H* = 2IzSz

Both ρ and *H* include I- and S-type product operators (Rule 1: yes) and only S-spins have different axis labels (Sy vs. Sz) (Rule 2: yes). 🡺 Master Table: yes

ρ = 2IxSx and *H* = Iz

Both ρ and *H* include I-spin (Rule 1: yes) and they have different axis labels (Ix vs. Iz) (Rule 2: yes). 🡺 Master Table: Yes

ρ = 2IxSx and *H* = 2IzSz

Both ρ and *H* include I- and S-type product operators (Rule 1: yes) but both spin types have different axis labels (Ix vs. Iz and Sx vs. Sz) (Rule 2: no). 🡺 Master Table: No

**Special cases for *J*-coupling evolutions**

According to *Spin Dynamics* (2nd Ed.), p. 483, there are four cases where a product operator does not evolve under a *J*-coupling Hamiltonian *I*jz*I*kz.

Case 1. If both spin *I*j and *I*k are missing in the product operator.

Case 2. If only one spin *I*j or *I*k is present, and that spin carries a z label.

Case 3. If both spins *I*j and *I*k are present, but both spins carry a z label.

Case 4. If both spins *I*j and *I*k are present, but neither spin carries a z label.

These all four cases are excluded by the two rules above.

Case 1. Rule 1 is not satisfied.

Case 2. Rule 1 is satisfied but Rule 2 is not satisfied.

Case 3. Rule 1 is satisfied but Rule 2 is not satisfied.

Case 4. Rule 1 is satisfied but Rule 2 is not satisfied.

**Implementation of the two rules in the programing code**

The two rules above can be evaluated in the programming code as shown below.

type\_mask\_vec = (rho\_axis.\*H\_axis)~=0;

% Check how many spin types get matched, matched: 1, unmatched: 0

% Ex. rho\_axis = [1 0 0] and H\_axis = [3 3 0] 🡺 type\_mask\_vec = [1 0 0];

axis\_diff\_vec = rho\_axis ~= H\_axis;

% Check the difference of the direction of each spin type, unmatched: 1, matched: 0

% Ex. rho\_axis = [1 0 0] and H\_axis = [3 3 0] 🡺 axis\_diff\_vec = [1 1 0];

axis\_mask\_vec = type\_mask\_vec.\*axis\_diff\_vec;

% Comparing the spin-type matching and spin-label unmatching.

% Ex. type\_mask\_vec = [1 0 0] and axis\_diff\_vec = [1 1 0] 🡺 axis\_mask\_vec = [1 0 0]

axis\_mask = sum(axis\_mask\_vec);

axis\_mask becomes 1 ONLY when the two rules are satisfied.

If axis\_mask is 1, then prepare

H\_axis = [h1 h2 h3 …] corresponding to *A* and

rho\_axis = [r1 r2 r3 …] corresponding to *B*

to calculate a new product operator

axis\_tmp = [a1 a2 a3 …] corresponding to *C*.

For each n (n = 1, 2, 3, …), the steps below are calculated.

- if both rn and hn are not 0, an takes an absolute value of the (hn, rn) component of the master table. If the value from the master table is negative, the sign of the coefficient is inverted.

- if rn is not 0 but hn is 0, an is same as rn. This is for cases such as

ρ = 2IzSz ([r1 r2] = [3 3]) and *H* = Iy ([h1 h2] = [2 0]) 🡺 *C*: 2IxSz ([a1 a2] = [1 3])

- if rn is 0 but hn is not 0, an is same as hn. This is for cases such as

ρ = Ix ([r1 r2] = [1 0]) and *H* = 2IzSz ([h1 h2] = [3 3]) 🡺 *C*: 2IySz ([a1 a2] = [2 3])

**Tips for MATLAB Symbolic Math Toolbox**

In many cases, calculated coefficients by Symbolic Math Toolbox have simplified and readable expressions. However, it may be necessary to rewrite the coefficients in another expression or to organize them with some certain terms. There are helpful Symbolic Math Toolbox functions for those operations. The functions below are some selected ones. Please read the MATLAB documentations for details.

**S = simplify(expr, v)**

performs algebraic simplification of **expr** with **v** steps. Inside the code, **simplify()** is used to simplify **obj.coef**. Each component of **obj.coef** is a result of the 10-step simplification. The number of the steps can be changed by a **PO** method **obj = set\_SimplifySteps(obj, new\_v)** to **new\_v**. A smaller value makes a calculation faster, but the output result may not be simplified enough. Reversely, a larger value may provide a more simplified expression, but it makes the calculation longer.

**R = rewrite(expr, target)**

This function is helpful to rewrite any trigonometric function in terms of the exponential function by specifying the target 'exp'.

>> syms q; coef = cos(q) + 1i\*sin(q); coefnew = rewrite(coef,'exp')

coefnew = exp(q\*1i)

**[C, T] = coeffs(p, vars)**

This function is helpful to organize terms in **p** with respect to **vars**. For example, if you like to separate **coef** above to terms with cos(q) from terms with sin(q),

>> [C,T] = coeffs(coef, [cos(q) sin(q)])

C = [1, 1i]

T = [cos(q), sin(q)]

**References**

Levitt, M. H., *Spin Dynamics*, 2nd Edition, Wiley, 2008.

Keeler, J., *Understanding NMR Spectroscopy*, 1st Edition, Wiley, 2005.

Güntert, P., *Symbolic NMR Product Operator Calculations*, *Int. J. Quant. Chem.* **2006**, *106*, 344 – 350.