**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 program can manipulate various types of operators, and this ability provides a rich environment of the calculations to users. The program will be helpful for educational use, for example, showing how to calculate product operators and explaining how pulse sequence components (Hahn-echo, INEPT etc.) work. Also, the program can be used to calculate evolutions of a density operator under a pulse sequence including phase cycling.

**Requirement**

MATLAB and MATLAB Symbolic Math Toolbox are required for 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. The calculation speed of the program is slower than that of a similar program written for *Mathematica* (P. Güntert *et al.*, 1993, P. Güntert, 2006).

**Introduction of the Program**

Before describing details of the program, I will show a couple of examples demonstrating the ability and flexibility of this program.

The first example is how to create spin operators in the workspace. The command

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

creates parameters, Ix, Iy, Iz, Ip, Im, Ia, Ib, Sx, Sy, Sz, Sp, Sm, Sa, Sb, and hE. They can be used as the spin angular momentum operators (*I*x, *I*y, *I*z, …) , lowering/raising operators (*I*+, *I*-, …), polarization operators (*I*α, *I*β, …) and unity operator (1/2*E*) of the *I* – *S* spin system. They are called **PO objects**.

Almost any product operators can be calculated by using these **PO objects**. For example, to calculate [*I*x, *I*y] = *I*x*I*y – *I*y*I*x,

>> rho = Ix\*Iy – Iy\*Ix

rho =

PO with properties:

txt: 'Iz\*1i'

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

basis: 'xyz'

disp: 1

axis: [3 0]

coef: [1×1 sym]

Ncoef: [1×1 sym]

sqn: [1×1 sym]

M:

[1i/2, 0, 0, 0]

[ 0, 1i/2, 0, 0]

[ 0, 0, -1i/2, 0]

[ 0, 0, 0, -1i/2]

coherence:

[aa, 0, 0, 0]

[ 0, ab, 0, 0]

[ 0, 0, ba, 0]

[ 0, 0, 0, bb]

The result is stored in a new **PO object**, rho, in this case. A **PO object** stores several types of information to describe (a) product operator(s). They are called **PO properties**. As you can see the property named **txt**, the result is Iz\*1i that corresponds to the well-known equation [*I*x, *I*y] = i\**I*z.

Another example is to get a matrix representation of 2*I*x*S*x – 2*I*y*S*y,

>> rho = 2\*Ix\*Sx – 2\*Iy\*Sy;

Then, the matrix representation can be accessed via the property **M** by

>> rhoMatrix = rho.M

[ 0, 0, 0, 1]

[ 0, 0, 0, 0]

[ 0, 0, 0, 0]

[ 1, 0, 0, 0]

As you can see, it is a pure DQ state. It is possible to express rho using the shift operators *I*+, *I*-, *S*+ and *S*-,

>> rho\_pmz = xyz2pmz(rho);

>> rho\_pmz.txt

ans =

'IpSp + ImSm'

If there is a given matrix, the program can create a corresponding **PO object**. For example,

>> M\_in = [1 0;0 -1];

>> rho = PO.M2xyz(M\_in, {'I'});

>> rho.txt

ans =

'Iz\*2'

The program can be used to check how NMR interactions influence a spin state, using provided functions called **PO methods**. For example, to see how *I*z evolves under a 90°y pulse followed by the chemical shift interaction,

>> rho = pulse(Iz,'I','y',pi/2).cs('I',o1\*t);

Pulse: I 90y

Ix

CS: I o1\*t

Ix\*cos(o1\*t) + Iy\*sin(o1\*t)

The program has a flexibility to change the number of spins and spin labels. For example, if you like to create a *I*1-*I*2-*I*3 spin system,

>> PO.create({'I1' 'I2' 'I3'})

then I1x, I1y, …, I3a, I3b, and hE will be created.

As show above, the program provides various methods to construct and manipulate product operators that will be helpful for understanding of the ideas of NMR spectroscopy.

**How to Use the Program**

**0. Set the Path for the Program**

All codes are stored in the MATLAB m-file **PO.m**. Put this file in a folder and set the path so that the program can be called from any working directories.

**1. Creating an Initial State of a System**

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

**1.1. PO.create()**

The first method is to use preset spin operators to construct a density operator by their combinations. The **PO** method **PO.create(spin\_label\_cell)** creates spin operators with labels defined in the cell array **spin\_label\_cell**. For example, in the case of the I-S two spin system,

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

creates the **PO objects** Ix, Iy, Iz, Ip, Im, Sx, Sy, Sz, Sp, Sm, and hE in the workspace. It is possible to create a desired density operator by combining the **PO objects** with the ‘\*’, ‘+’, ‘-‘ , ‘/’ and ‘^’ operators and coefficients. Frequently used coefficients are created as the sym-class when **PO.create()** is executed (see the explanation for **PO.symcoef()** for details). As an example, to create ρ = *I*x\*cosθ + 2*I*y*S*z\*sinθ,

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

There are a couple of rules for the use of the ‘\*’, ‘+’, ‘-‘, ‘/’ and ‘^’ operators with **PO objects**. Firstly, these operators should be used between **PO objects** with the same number of spin types. Rules for each operator are shown below.

• ‘\*’ operator can be used to calculate

1. **obj1**\***obj2**. **obj1** and **obj2** are **PO objects**. If the **basis** properties of these objects are different, the rules below are applied to determine the basis of the returned object.

Old bases New basis

('xyz', 'pmz') 🡺 'pmz'

('xyz', 'pol') 🡺 'pol'

('pmz', 'pol') 🡺 'pol'

where 'xyz' is for the Cartesian operator basis (*I*x, *I*y, *I*z), 'pmz' is for the lowering/raising operator basis (*I*+, *I*-, *I*z), and 'pol' is for the polarization operator basis (*I*α, *I*β, *I*+, *I*-). These rules mean that the 'xyz' basis is overloaded by the 'pmz' or 'pol' bases, and the 'pmz' basis is overloaded by the 'pol' basis. If it is necessary to get the result with a different basis, use a basis-conversion method such as **pmz2xyz()**, **pol2xyz()** or **pol2pmz()**. For example, to obtain the result of Ix\*Ia with the 'xyz' basis instead of the 'pol' basis,

>> Ix\*pol2xyz(Ia)

2. **a\*obj** or **obj\*a**. **obj** is a **PO object** and **a** can be a double, sym, or char class. For example, expressions 2\*Ix, sym(2)\*Ix and '2'\*Ix are equivalent.

3. **v\_row\*obj** or **obj\*v\_col** to obtain the vectors **v\_row\*obj.M** or **obj.M\*v\_col**. **obj** is a **PO object.** **v\_row** and **v\_col** are row and column vectors, respectively, in the double or sym class. The lengths of these vectors should be same with the row or column size of **obj.M**.

• **‘+’** operator can be used to calculate

1. **obj1 + obj2**. **obj1** and **obj2** are **PO objects**. If the **basis** properties of these objects are different, the same rules for the '\*' operator are applied.

2. **obj** + **a** or **a + obj**. **obj** is a **PO object** and **a** can be a double, sym, or char class. This calculation means an addition of **a\*E** to **obj**.

• **‘-’** operator can be used to calculate

1. **obj1 - obj2**. **obj1** and **obj2** are **PO objects**. If the **basis** properties of these objects are different, the same rules for the '\*' operator are applied.

2. **obj** - **a** or **a - obj**. **obj** is a **PO object** and **a** can be a double, sym, or char class. This calculation means **obj - a\*E** or **a\*E - obj**, respectively.

• **‘/’** operator can be used to calculate

**obj/a**. **obj** is a **PO object** and **a** can be a double, sym, or char class. Note that a **PO object** cannot be a divisor, i.e., **a/obj** or **obj1/obj2** can not be calculated.

• **‘^’** operator can be used to calculate

**obj^n**. **obj** is a **PO object** and **n** should be a double scalar. **n** should be 0 or a natural number.

**1.2. PO()**

The second method is the use of the class constructor **PO()**. As an example, if you like to construct ρ = *I*1x\*cosθ + 2*I*1y*I*2z\*sinθ,

>> syms q;

>> rho = PO(2, {'Ix' 'I1yI2z'},{cos(q) sin(q)},{'I1' 'I2'});

The general syntax of the constructor is

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

where **obj** is a **PO** **object**, **spin\_no** is the number of spin types in the system, **sp\_cell** is a cell array describing product operators (without 2*N*s – 1 coefficients, e.g., 2 in 2*I*z*S*z, 4 in 4*I*x*S*y*K*z), **coef\_cell** is a cell array for coefficients (coefficients can be given by double, char or sym classes. They should not also include the 2*N*s – 1 coefficients) and **spin\_label\_cell** is a cell array for spin labels that will be used as the property **spin\_label**.

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

ρ = *I*x + *S*y

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

If only the first two parameters are given, the third, **coef\_cell**, is automatically set as {1 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'}. If a user like to use a different set of labels as a default, change a constant property **spin\_label\_cell\_default** in the code.

ρ = a\**I*1x + b\**I*2y

>> 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 'a' and 'b' 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 = *I*x\*cosθ + *I*y\*sinθ

>> syms q;

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

The coefficients cosθ and sinθ can be described by the sym class. In this case, create q as the sym class and give cos(q) and sin(q) as **coef\_cell**.

ρ = *I*1x in a 3-spin system (*I*1, *I*2 and *I*3)

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

The number of total spin types in the system should be set by the constructor, i.e., it cannot be changed later. **spin\_label\_cell** should be set for this particular 3-spin system.

ρ = *I­*1*x* + 4*I­*1x *I­*2y *I­*3z

>> rho = PO(3, {'I1x' 'I1xI2yI3z'},{1 1},{'I1' 'I2' 'I3'});

Both **sp\_cell** and **coef\_cell** should not include the 2*N*s - 1 coefficient, 4 in 4*I­*1x *I­*2y *I­*3z. 2*N*s - 1 coefficients of product operators are calculated automatically and are stored in the property **Ncoef**.

ρ = *I*+*S*-*K*z

>> 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. Note that it is not allowed to use different bases in a single **PO** **object**. In the case of the **pmz** basis, the 2*N*s - 1 coefficients are not considered, and **Ncoef** is set as 1 for each term. To convert the basis from **xyz** to **pmz**, **xyz2pmz()** can be used. Reversely, **pmz2xyz()** is for the conversion from **pmz** to **xyz** basis.

ρ = *I*α*S*β

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

**PO** **methods** can handle the polarization operator basis (**pol** basis). In this basis, '**a**', '**b**' , '**p**' and '**m**' can be used where '**a**' and '**b**' are the polarization operators. In the case of the **pol** basis, the 2*N*s - 1 coefficients are not considered, and **Ncoef** is set as 1 for each term. To convert the basis from **xyz** to **pol**, **xyz2pol()** can be used. Reversely, **pol2xyz()** is for the conversion from **pol** to **xyz** basis.

Note that **PO()** cannot construct a product operator of same spin-type operators, i.e., ρ = *I*x\**I*y.

As a special case,

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

creates the **1/2E** operator. In fact, any types of characters that is not defined in **spin\_label** are considered as the **1/2E** operator.

The first method, **PO.create()**, will be helpful for demonstrations of product operators in teaching classes. The second method, **PO()**, will be useful in a script simulating a pulse sequence.

As mentioned above, information characterizing current product operators are stored as **PO properties**. Useful properties for users are '**txt**' that shows a text output of the operators, '**M**' that shows a matrix representation of the operators, and '**coherence**' that shows populations and coherences in the matrix. Values of a **PO** **property** can be obtained by the syntax **obj.PropertyName**. For example, to get a matrix representation,

>> rho\_matrix = rho.M;

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

Effects of NMR interactions such as RF pulse, chemical shift and *J*-coupling to a spin system can be calculated by **PO methods**.

**2.1. 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 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 in **spin\_label** (1 for 'I', 2 for 'S' in {'I' '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 a double or sym class, such as pi/2 (double) or syms q (sym).

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 input line simple.

Let's assume a 5-spin system, *I*1, *I*2, *I*3, *S*4 and *S*5.

>> rho = PO(5,{'I1z' 'I2z' 'I3z' 'S4z' 'S5z'},{1 1 1 1 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 sym 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)** .

**2.2. 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});

**2.3. *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 a label 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 **'\*'** cannot be used 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});

**2.4. Pulse Field Gradient**

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 ratios of spins in the system.

**G** can be a double or sym class. Components of **gamma\_cell** can be also a double or sym 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**, **gamma\_cell** and an internal, symbolic constant **Z** as their products, and they are used as input parameters of **simcs()**. In the case above, the angles are {G\*gH\*Z G\*gH\*Z G\*gC\*Z}. 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 obtained from the reference (Güntert, 2006).

The method to delete terms influenced by a pulse filed gradient is

**obj = dephase(obj, coef\_cell)** or **obj = obj.dephase (coef\_cell)**.

If **coef\_cell** is not assigned, terms including **Z** in **obj.coef** are deleted from **obj**. If **coef\_cell** (cell array) is assigned, terms including **Z\*coef\_cell{1}\*coef\_cell{2}\*…\*coef\_cell{end}** are deleted thus terms deleted can be limited. This method was obtained from the reference (Güntert, 2006).

**Independence of Methods from Basis Type**

The methods shown above can be applied to a **PO** **object** with any basis type. The basis type of the input **PO object** is applied to the resulted **PO object**. For example,

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

>> pulse(xyz2pmz(Iz),'I','y',pi/2);

Pulse: I 90y

Ip\*1/2 + Im\*1/2

>> pulse(xyz2pol(Iz),'I','y',pi/2);

Pulse: I 90y

IpSa\*1/2 + IpSb\*1/2 + ImSa\*1/2 + ImSb\*1/2

**Applying Multiple PO Methods in One Line**

You can apply multiple methods in one line using the dot **'.'** as a 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 additional methods as utilities. Some of them are **static methods**, i.e., they are called by the syntax **PO.MethodName()**. They can be used, for example, when a pulse sequence is simulated.

There are methods for the basis-conversion among the 'xyz', 'pmz' and 'pol' bases. For example,

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

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

and

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

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

are methods for the conversions between the Cartesian operator basis (**obj\_xyz)** and raising/lowering operator basis (**obj\_pmz**) (Güntert, 2006). **xyz2pol()** and **pol2xyz()** are for the conversion between 'xyz' and 'pol', and **pmz2pol()** and **pol2pmz()** are for the conversion between 'pmz' and 'pol'.

**obj\_pol = PO.M2pol(M\_in, spin\_label\_cell)**

**obj\_pol**, a **PO object** with the ‘pol’ basis, is created from the matrix **M\_in**. **M\_in** should be 2*N* x 2*N* in the double or sym class. **spin\_label\_cell** is a cell array for spin labels. If **spin\_label\_cell** is not assigned, {'I' 'S' 'K' 'L' 'M'} is used for the spin label. Similarly, **obj\_pmz = PO.M2pmz(M\_in, spin\_label\_cell)** and **obj\_xyz = PO.M2xyz(M\_in, spin\_label\_cell)** are used to create **PO objects** with the ‘pmz’ and ‘xyz’ bases, respectively, from **M\_in**.

**PO.symcoef(spin\_label\_cell, add\_cell)**

creates frequently-used symbolic constants based on the information of **spin\_label\_cell**. For example,

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

creates oI, oS, o1, o2, JIS, J12, gI, gS, g1, g2 systematically from {'I' 'S'} in addition to a, b, d, f, t1, t, q, w, B, and G. It is possible to list additional parameters by **add\_cell**.

**obj = UrhoUinv(obj, H, q)** or

**obj = obj.UrhoUinv (H, q)**

calculates an evolution of a density operator ρ under a Hamiltonian *H*, ρ(t) = exp(-i*H*t)ρ(0)exp(i*H*t). *H* can be described by frequency and operator parts, *H* = ω*H*’(e.g, *H* = ω*I*\**I*z, *H* = ωnut\**I*x, *H* = π*JIS*\*2*I*z*S*z,) . **obj** and **H** are the PO objects corresponding to ρ under *H*’, respectively. **q** is a rotation angle in radian corresponding to ω\*t. By setting **q** as 1, the full form of the Hamiltonian, i.e., *H*t= ω\*t\**H*’ can be put as **H**. This can be applied, for example, to the case *H* = ω*I*\**I*z *+* π*JIS*\*2*I*z*S*z (H = oI\*t\*Iz + pi\*JIS\* t\*2\*Iz\*Sz). If 1) **H** includes only one term with the 'xyz' basis, and 2) **H** or **obj** is a product of up to two operators, a method based on the cyclic commutation rules (**UrhoUinv\_mt()**) is called. Otherwise, a method based on the matrix calculation (**UrhoUinv\_M()**) is called. Although **UrhoUinv\_mt()** has some limitations regarding **H**, the calculation speed is much faster than that of **UrhoUinv\_M()**, especially the number of spins gets increased.

**obj3 = commutator(obj1, obj2)** or

**obj3 = obj1.commutator(obj2)**

calculates the commutation of **obj1** and **obj2**, i.e., **obj3** = [**obj1**, **obj2**] = **obj1**\***obj2** – **obj2**\***obj1** where **obj1** , and **obj2** and **obj3** are **PO objects**.

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

outputs a phase value, **phout**, from the 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 = *I*x\*cosθ + *I*y\* 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 characters describing product operators, terms for 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\_cell** is a cell array describing spin types to be observed (e.g., {'I'}, {'I' 'S'}, {'I1' 'I2'}, {1}, {1 2}). The wildcard character **'\*'** can be used for **sp\_cell** same as **simpulse()**.

**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-').

**obj = set\_coef(obj, new\_v)** or

**obj = obj.set\_coef(new­\_v)**

overwrites values in **obj.coef** to **new\_v**. This method can be used if it is preferred to rewrite the obtained coef values to a different expression.

**obj = set\_basis(obj, basis\_in)** or

**obj = obj.set\_coef(basis\_in)**

rewrites the **PO** object **obj** to a different basis, **basis\_in**.

**Technical Details**

**Design of the Program**

The code is written with the object-oriented programming (OOP) style. In manner of the OOP, parameters for characterizing product operators and functions for NMR interactions are called **properties** and **methods** of a **class** named **PO**, respectively. A **PO class object** stores the **PO class properties**, and the **object** is processed by the **PO class methods**. By designing **properties** and **methods** properly, **PO class objects** can be handled in manner of the product operator formalism. For example, the ‘\*’, ‘+’, ‘-‘ , ‘/’ and ‘^’ operators are implemented as **PO methods** so that they work as corresponding operators for product operators (it is called operator overloading).

**PO Class Properties**

Any product operator can be described by three characteristic properties, spin types, axis labels (x, y or z) and a coefficient. For example, in the case of -4*I*x*S*z*K*z\*cosθ, the axis labels are x, z and z for the 1st (*I*), 2nd (*S*) and 3rd (*K*) spins, respectively, and the coefficient is -cosθ. Note that the coefficient “4” is related to the number of the active spin types in the product operator (in this case the number (*N*s) is 3 for *I*, *S* and *K*, and 4 can be obtained from 2*N*s – 1) and thus it is not considered as an independent coefficient. In the **PO class properties**, information on the axis labels for the spins and the coefficients are stored as **axis** and **coef**, respectively.

**axis**: This property stores axis labels of product operators. It is a *M* x *N* matrix where *M* is the number of product operators in the system and *N* is the number of spin types in the system. Column positions of the matrix correspond to the spin types. Each component in the matrix has a value of 0, 1, 2 or 3 corresponding to *E*, *I*x, *I*y or *I*z, respectively, in the case of the Cartesian basis. In the case of the lowering/raising operator basis, 4 and 5 are used for *I*+ and *I*-, respectively. Also, in the case of the polarization operator basis, 6 and 7 are used for *I*α and *I*β, respectively. For example, the **axis** property of *I*1x + *I*2x + *I*2z (*M* = 3) in the *I*1-*I*2 system (*N* = 2) is [1 0; 0 1; 0 3].

**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 another property, **Ncoef**.

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

**spin\_label**: This property defines labels of spin types in a 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 2*N*s - 1 coefficients for product operators in the current 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.

**basis**: This property stores a type of the operator basis of the current system. There are three types of bases, ‘xyz’ for the Cartesian operator basis (Ix, Iy, Iz), ‘pmz’ for the lowering/raising operator basis (Ip, Im, Iz) and ‘pol’ for the polarization operator basis (Ia, Ib, Ip, Im).

**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.

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

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

All properties except for **disp** are protected, and they cannot be accessed from the workspace or scripts. The properties **coef** and **basis** can be accessed via **PO methods**, **set\_coef()** and **set\_basis()**, respectively.

**Merits to Use the Axis Property**

The **axis** property is beneficial for some important calculations in this program. One is a multiplication of **PO objects**. This calculation can be handled as an addition of the **axis** properties of the **PO objects**. For example, **axis** properties of Iz, Sy, Kx and their product Iz\*Sy\*Kx are

Iz : [3 0 0]

Sy : [0 2 0]

Kx : [0 0 1]

Iz\*Sy\*Kz: [3 2 1]

As you can see, the **axis** property of Iz\*Sy\*Kx is the sum of the three vectors (i.e., the **axis** properties) of the three operators.

As a note, there is an exception for a multiplication of the same spin type. For example, if you consider the product of Iz, Ix\*Sy and Kx,

Iz : [3 0 0]

IxSy : [1 2 0]

Kx : [0 0 1]

Iz\*Ix\*Sy\*Kz: [4 2 1]

In this case, a special calculation is necessary for the *I*-spin because the **axis** value obtained by the addition is not correct. The **axial** value of Iz\*Ixshould be 2 because *I*z\**I*x = i/2\**I*y. In the code, there is a branch to calculate an appropriate **axis** value for this type of cases.

Another benefit is that **axis** values of two operators can be used as indexes of a matrix that describes the cyclic commutations of the two operators. The details are explained in the next section.

**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 *I*x, *I*y, *I*z are in the cyclic commutation ([*I*z, *I*x] = i*I*y) 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) = *I*x evolves under a chemical shift Hamiltonian *H* = ω*I*z during a time period of t as

ρ(t) = exp(-i*H*t) ρ(0) exp(i*H*t) = exp(-iωt*I*z) *I*x exp(iωt*I*z) = *I*x cos(ωt) + *I*y 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 *I*x, *I*y and *I*z. For example, in the case of exp(-iθ*I*z) *I*x exp(iθ*I*z) = *I*x cosθ + *I*y 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 3rd-row, 1st-column component in the table, i.e., 2 meaning *I*y. 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* = *I*x). 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 (e.g., expm(-1i\*q\*Iz.M)\**I*x.M\*expm(1i\*q\*Iz.M) ) 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.

Note that these rules can be used for spin-1/2 with the condition that one of *H* and ρ is a product of up to two spin operators (e.g. *H* = 2*I*z*S*z but not like 4*I*z*S*z*K*z). Since the spin Hamiltonians of the pulse, chemical shift evolution, and *J*-coupling evolution satisfy this condition naturally, the master table can be used for the calculation. An example that doesn't satisfy the rule 1 and 2 but satisfies the cyclic commutation is the set of 4*I*x*S*y*K*z, 4*I*y*S*z*K*x and 4*I*z*S*x*K*y.

The rule 1 is obvious but how about the rule 2. Here is the analysis. Suppose *H* = 2*I*a*S*b and ρ = 8*I*b*S*b*KL* where *K*, *L* are spin operators that are different types each other in addition to *I* and *S*. Suppose [*I*a, *I*b] = i*I*c 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] = 2*I*a*S*b 8*I*b*S*b*KL* – 8*I*b*S*b*KL* 2*I*a*S*b = 16 *I*a*I*b*S*b2*KL* - 16*I*b*I*a*S*b2*KL*

= 16 (*I*a*I*b – *I*b*I*a)*S*b2*KL* = 4(*I*a*I*b – *I*b*I*a)*KL* = i4*I*c*KL*. Note that *S*b2 = 1/4*E* for spin-1/2.

[4*I*c*KL*, *H*] = 4*I*c*KL* 2*I*a*S*b - 2*I*a*S*b 4*I*c*KL* = 8*I*c*I*a*S*b*KL* - 8*I*a*I*c*S*b*KL* = 8(*I*c*I*a – *I*a*I*c)*S*b*KL* = i8*I*b*S*b*KL* = iρ.

[ρ, 4*I*c*KL*] = 8*I*b*S*b*KL* 4*I*c*KL* - 4*I*c*KL* 8*I*b*S*b*KL* = 32*I*b*I*c*S*b*K*2*L*2 – 32*I*c*I*b*S*b *K*2*L*2 = 2(*I*b*I*c – *I*c*I*b) *S*b = i2*I*a*S*b = i*H*. Note that *K*2 = *L*2 = 1/4*E* 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* = 2*I*b*S*b and ρ = 8*I*b*S*b*KL*,

[*H*, ρ] = 2*I*b*S*b 8*I*b*S*b*KL* – 8*I*b*S*b*KL* 2*I*b*S*b = 16 *I*b2*S*b2*KL* - 16*I*b2*S*b2*KL* = 0

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

In the case of *H* = 2*I*a*S*b and ρ = 8*I*b*S*c*KL*, [*H*, ρ] is calculated as 0 from [2*I*a*S*b, 2*I*b*S*c] = 0 that can be obtained from the matrix representation.

[*H*, ρ] = 2*I*a*S*b 8*I*b*S*c*KL* - 8*I*b*S*c*KL* 2*I*a*S*b = 2*I*a*S*b 2*I*b*S*c 4*KL* - 2*I*b*S*c 4*KL* 2*I*a*S*b = 2*I*a*S*b 2*I*b*S*c 4*KL* - 2*I*b*S*c 2*I*a*S*b 4*KL* = [2*I*a*S*b, 2*I*b*S*c]\*4*KL* = 0 (Read *Spin Dynamics* (2nd Ed.), p. 403, Eq. 15.24 and p. 407, Note 3).

Examples for these rules

ρ = *I*y and *H* = *I*z

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

ρ = *S*y and *H* = *I*z

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

ρ = *I*y and *H* = *I*y

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

ρ = 2*I*z*S*y and *H* = *I*z

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

ρ = 2*I*z*S*y and *H* = 2*I*z*S*z

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

ρ = 2*I*x*S*x and *H* = *I*z

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

ρ = 2*I*x*S*x and *H* = 2*I*z*S*z

Both ρ and *H* include *I*- and *S*-type product operators (Rule 1: yes) but both spin types have different axis labels (*I*x vs. *I*z and *S*x vs. *S*z) (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 cyclic commutation using the two rules above with the master table is calculated in the **PO** method, **UrhoUinv\_mt()**. In this method, the two rules are evaluated 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 = 2*I*z*S*z ([r1 r2] = [3 3]) and *H* = *I*y ([h1 h2] = [2 0]) 🡺 *C*: 2*I*x*S*z ([a1 a2] = [a1 r2] = [1 3])

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

ρ = *I*x ([r1 r2] = [1 0]) and *H* = 2*I*z*S*z ([h1 h2] = [3 3]) 🡺 *C*: 2*I*yS*z* ([a1 a2] = [a1 h2] = [2 3])

**Tips for MATLAB Symbolic Math Toolbox**

In most 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 certain terms. There are helpful Symbolic Math Toolbox functions for those operations. The functions below are some of them. Please read the MATLAB documentations for details.

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

<https://www.mathworks.com/help/symbolic/simplify.html>

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)**

[https://www.mathworks.com/help/symbolic/rewrite.html](https://www.mathworks.com/help/symbolic/simplify.html)

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)**

[https://www.mathworks.com/help/symbolic/sym.coeffs.html](https://www.mathworks.com/help/symbolic/simplify.html)

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)]

**snew = subs(s, old,new)**

<https://www.mathworks.com/help/symbolic/subs.html>

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.; Schaefer, N.; Otting, G.; Wüthrich, K., *POMA: A Complete Mathmatica Implementation of the NMR Product-Operator Formalism*, *J. Magn. Reson. Ser. A,* **1993**, *101*, 103 – 105.

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