Ver. 1.0.0

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

Dr. 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 nuclei for Nuclear Magnetic Resonance (NMR) using MATLAB. The program can manipulate various types of operators, and this ability provides the user with a rich environment to perform calculations. 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. 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 the similar program written for *Mathematica* (P. Güntert *et al.*, 1993, P. Güntert, 2006).

Introduction of Program

Before describing details, 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 the 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 a half of unity operator ($1/2E$) 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 $[I_x, I_y] = I_xI_y - I_yI_x$,

```
>> rho = Ix*Iy - Iy*Ix
rho =
  PO with properties:
```

```
          txt: 'Iz*1i'
    spin_label: {'I'  'S'}
         basis: 'xyz'
          logs: 'Iz*1i'
          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 the new **PO object**, `rho`, in this case. A **PO object** stores several types of information to describe product operators. 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 $2I_xS_x - 2I_yS_y$,

```
>> 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 dedicated 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 would like to create the I_1-I_2-I_3 spin system,

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

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

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

How to Use Program

0. Setting Path for Program

All codes are described in the MATLAB m-file **PO.m** in the class folder **@PO**. Add the parent folder of **@PO** to the MATLAB path so that the program can be called from any working directory. You can define a **PO method** in a separate file in **@PO** (for example, **UserFun1.m** in **@PO**). Details are in https://www.mathworks.com/help/matlab/matlab_oop/methods-in-separate-files.html .

1. Creating Initial State of 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 construct a density operator by the combination of spin operators. The **PO** method **PO.create(spin_label_cell)** creates **PO objects** (spin operators) with labels defined in a 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 these **PO objects** with the '*', '+', '-' , '/' and '^' operators and coefficients. Simultaneously, frequently used coefficients are created as the sym-class (see the explanation of **PO.symcoef()** for details of the coefficients created). As an example, to create $\rho = I_x*\cos\theta + 2I_yS_z*\sin\theta$,

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

There are a couple of rules to use 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. Hereafter, **obj** indicates a **PO object** unless otherwise noted.

- '*' operator can be used to calculate:

  1. **obj1*obj2**. 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 overwritten with the 'pmz' or 'pol' bases, and the 'pmz' basis is overwritten with 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** and **obj*a**. **a** can be a double, sym, or char class. For example, the expressions 2*Ix, sym(2)*Ix and '2'*Ix are equivalent.

  3. **v_row*obj** and **obj*v_col** to obtain the vectors **v_row*obj.M** and **obj.M*v_col**. **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**. If the **basis** properties of these objects are different, the same rules for the '*' operator are applied.

  2. **obj** + **a** and **a** + **obj**. **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**. If the **basis** properties of these objects are different, the same rules for the '*' operator are applied.

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

- **‘/’** operator can be used to calculate:

  obj/a. **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. **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 would like to construct $\rho = I_{1x}*\cos\theta + 2I_{1y}I_{2z}*\sin\theta$,

```
>> 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 **spin_no** is the number of spin types in the system, **sp_cell** is a cell array describing product operators (without the 2^{N_S-1} coefficients, e.g., 2 in $2I_zS_z$, 4 in $4I_xS_yK_z$), **coef_cell** is a cell array for the coefficients (the 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 the spin labels that will be used as the property **spin_label**.

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

$\rho = I_x + S_y$

```
>> 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, the spin types used for **sp_cell** should be selected from the first **spin_no** components of {'I' 'S' 'K' 'L' 'M'}. If a user wants to use a different set of the labels as a default, change the constant property **spin_label_cell_default** in the code.

$\rho = a*I_{1x} + b*I_{2y}$

```
>> 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 ('I', 'S', 'K', …) and two characters with a letter and a number ('I1', 'I2', 'I3', …). For example, 'SP' or 'I10' is not allowed to use as a label.

$\rho = I_x*\cos\theta + I_y*\sin\theta$

```
>> 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**.

$\rho = I_{1x}$ 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 3-spin system.

$\rho = I_{1x} + 4I_{1x} I_{2y} I_{3z}$

```
>> 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 $4I_{1x} I_{2y} I_{3z}$. $2^{N_S - 1}$ coefficients of product operators are calculated automatically and are stored in the property **Ncoef**.

$\rho = I^+S^-K_z$

```
>> 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 as the labels, 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.

$\rho = I^{\alpha}S^{\beta}$

```
>> 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 as the labels, 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., $\rho = I_x * I_y$.

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, for example, 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, '**coherence**' that shows populations and coherences in the matrix, and **logs** that keeps the record of the applied methods. 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 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

The method to apply a single pulse or simultaneous pulses is

obj = pulse(obj, sp_cell, ph_cell, q_cell) or

obj = obj.pulse(sp_cell, ph_cell, q_cell) ,

where **sp_cell** , **ph_cell** , and **q_cell** are cell arrays describing spin types to be manipulated, quadrature phases for pulses, and flip angles of pulses in radian, respectively. An element of **sp_cell** can be characters ('I', 'S', 'I1' or 'I2' etc. defined in **spin_label**) or the numbers describing the order of the spins in **spin_label** (1 for 'I', 2 for 'S' in {'I' 'S'} etc.). An element of **ph_cell** can be the characters such as 'x', 'X' or '-y' or the numbers 0, 1, 2 or 3 for x, y, -x or -y, respectively. An element of **q_cell** 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});
```

```
>> rho = PO(2, {'Iz' 'Sz'});
>> rho = pulse(rho, {'I' 'S'}, {'x' 'y'}, {pi/2 pi/2});
```

It applies a 90_x pulse to I and 90_y to S. Equivalently,

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

The wildcard character '*****' can be used for **sp_cell** to make an 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 90_x pulse to all I spins, the wildcard character can be used as

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

If applying 90_x pulses to both I and S spins,

```
>> rho = pulse(rho, {'*'},{'x'},{pi/2});
```

If applying a 90_x pulse to all I spins and a 180_y pulse to all S spins,

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

Pulses with Phase Shift

The method to apply a single pulse or simultaneous pulses with arbitrary phases is

obj = pulse_phshift(obj, sp_cell, ph_cell, q_cell) or

obj = obj. pulse_phshift(sp_cell, ph_cell, q_cell) .

The difference from **pulse()** is that **ph_cell** includes arbitrary phases in radian. An element of **ph_cell** can be a double or sym class.

2.2. Chemical Shift

The method to apply the chemical shift evolution to spins is

obj = cs(obj, sp_cell, q_cell) or

obj = obj. cs(sp_cell, q_cell)

where **sp_cell** and **q_cell** are cell arrays describing spin types to be affected and rotation angles in radian, respectively. The formats of **sp_cell** and **q_cell** are same as the ones used for **pulse()**. The wildcard character **'*'** also can be used for **sp_cell** same as **pulse()**.

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

or

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

2.3. *J*-coupling

The method to apply the *J*-coupling evolutions to spin pairs is

obj = jc(obj, sp_cell, q_cell) or

obj = obj.jc(sp_cell, q_cell) ,

where **sp_cell** and **q_cell** are cell arrays describing labels for the spin pairs corresponding to the *J*-coupling Hamiltonians and rotation angles in radian, respectively. An element of **sp_cell** 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});
```

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 = jc(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) ,

where **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 the internal, symbolic constant **Z** as their products, and they are used as input parameters of **cs()**. In the case above, the angles are `{G*Z*gH G*Z*gH G*Z*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 is obtained from the reference (Güntert, 2006).

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

obj = dephase(obj) or

obj = obj.dephase ().

This method is obtained from the reference (Güntert, 2006).

2.5. Notes for Applying PO Methods to PO Object

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 resulting **PO object**. For example,

```
>> PO.create({'I' 'S'})
>> pulse(xyz2pmz(Iz),{'I'},{'y'},{pi/2});% pmz basis
Pulse: I  90y
    Ip*1/2 + Im*1/2
>> pulse(xyz2pol(Iz),{'I'},{'y'},{pi/2});% pol basis
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 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 of the methods to be applied to the system is left to right (**pulse()** => **cs()** => **jc()**) but not right to left (**jc()** => **cs()** => **pulse()**).

3. Running Pulse Sequence

It is possible to construct a pulse sequence combining the **PO methods** above in addition to other **PO methods** as utilities (**4. Utilities** for details). In this program, a format for constructing a pulse sequence is provided with the dedicated **PO method**, **PO.run_PS()**. Many types of pulse sequences can be calculated using them.

[rho_cell, rho_detect_cell, rho_total, rho_obs, a0_M, rho_M] = PO.run_PS(fname)

runs a pulse sequence defined in a text file, **fname**. Note that it is not necessary for the text file to be a MATLAB script file (*.m). Any ASCII file can be loaded. The lines below show an example of the input file (the Hahn-echo experiment including the miscalibration of the 180 pulse).

```
% Para begin %
spin_label_cell = {'I'};
rho_ini = Iz;
% rho_ini = PO(1,{'Iz'},{1},spin_label_cell);
obs_cell = {'I'};
ph_cell{1} = [1,2,3,0];
ph_cell{2} = [0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3];
phRtab = [0,3,2,1,2,1,0,3];
phid = 1:16;
coef_cell = {};
disp_bin = 1;
% Para end %

% PS begin %
rho = rho.pulse({1},{ph1},{1/2*pi});% 90-pulse
rho = rho.cs({1},{oI*t});% Chemical shift evolution
rho = rho.pulse({1},{ph2},{pi+d});% 180+d-pulse, where d indicates the
miscalibration of 180-pulse
rho = rho.cs({1},{oI*t});% Chemical shift evolution
% PS end %
```

The parameters required for the pulse sequence should be described between the lines, '`% Para begin %`' and '`% Para end %`'.

spin_label_cell is a cell array for the spin labels. If it is not assigned, {'I' 'S' 'K' 'L' 'M'} is used (this creates a 5-spin system, and the calculation speed will get slow. Try to assign **spin_label_cell** explicitly.).

rho_ini is a **PO object** of the initial state. It should be described by **PO objects** or the constructor **PO()**.

obs_cell is a cell array defining the observed spins used in **observable()**. The wildcard character '*' can be used. If it is not assigned, {'*'} is used, meaning all spins are observed.

ph_cell is a cell array defining the phase tables. ph_cell{*n*} is a row vector corresponding to ph*n* in the pulse sequence. Describe ph_cell{1}, ph_cell{2}, …, for ph1, ph2, …, respectively.

phRtab is a row vector defining the receiver phase. Only the quadrature phase is accepted.

phid is a row vector defining the phase cycling steps. For a full phase cycling, it is not necessary to assign **phid**. If it is necessary to run particular sets of phase steps, assign this parameter. For example, if you like to check 2nd and 4th steps of the phase cycling, phid = [2 4];

coef_cell is a cell array to create additional symbolic coefficients not created by **PO.symcoef()**. If it is not assigned, {} is used.

disp_bin is a binary value to control the display of the pulse sequence on the Command Window (1 for ON, 0 for OFF). If it is not assigned, 1 is used.

The code is not sensitive to the order that the above parameters are described. If other parameters are required for the pulse sequence section, they should be described in this parameter section. If a parameter is created by another parameter, for example, a1 = 2 and a2 = 2*a1, they should be described in this order.

The section for the pulse sequence should be described between the lines, '`% PS begin %`' and '`% PS end %`'. Names of the input and output **PO objects** should be **rho**. The code is sensitive to the order of the methods described in this section.

The output parameters are explained below.

rho_cell is a cell array storing **PO objects** after the pulse sequence section for phase cycling steps. For example, if there are 8 steps of the phase cycling, **rho_cell** stores 8 **PO objects**.

In each phase cycle, the resulting **PO object** after the pulse sequence section is handled by **receiver()**. The obtained **PO object** from **receiver()** is stored in **rho_detect_cell**.

rho_total is a **PO object** that is the sum of **rho_detect_cell**.

rho_obs is a **PO object** that shows observable spins in **rho_total**, obtained from **rho_obs = observable(rho_total, obs_cell)**.

a0_M and **rho_M** are matrices combining **a0_V** and **rho_V** from **[a0_V,rho_V] = SigAmp(obj, obs_cell, phR),** where **obj** is a **PO object** after the pulse sequence section. The calculation cost of this method is a bit high. If it is not necessary to obtain them, exclude them from the output list.

4. Utilities

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

obj_pmz = xyz2pmz(obj_xyz) or
obj_pmz = obj_xyz.xyz2pmz()
and
obj_xyz = pmz2xyz(obj_pmz)
obj_xyz = obj_pmz.pmz2xyz()
are 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 the 'xyz' and 'pol' bases, and **pmz2pol()** and **pol2pmz()** are for the conversion between the 'pmz' and 'pol' bases.

obj_pol = PO.M2pol(M_in, spin_label_cell)

obj_pol, a **PO object** with the ‘pol’ basis, is created from a 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, c, d, f, gH, gC, gN, t1, t2, t3, t4, t, q, w, B, J, and G. It is possible to add other parameters by a cell array, **add_cell**.

dispPO(obj) or

obj.dispPO()

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

ID Product-Operator Coefficient

For example, in the case of $\rho = I_x*\cos\theta + I_y* \sin\theta$

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

dispPOtxt(obj) or

obj.dispPOtxt()

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

obj = receiver(obj, phR) or

obj = obj. receiver(phR)

rotates product operators (defined by **obj) -phR** around Z-axis where **phR** is a receiver phase in the quadrature phase (i.e., 'x', 'X', 0, '-y', '-Y', 3, etc.). This method is obtained from the reference (Güntert, 2006).

obj = observable(obj, sp_cell) or

obj = obj. observable (sp_cell)

selects observable terms of spin types defined by a cell array **sp_cell**. The format of **sp_cell** is same as the one used for **pulse()** accepting the wildcard character '*', e.g., {'I*'}. If **sp_cell** is not assigned, all spin types are considered for the selection. This method is obtained from the reference (Güntert, 2006).

[a0_V,rho_V] = SigAmp(obj, sp_cell, phR) or

[a0_V,rho_V] = obj.SigAmp(sp_cell, phR)

calculates a signal amplitude $a_{[-]}$ corresponding to a (-1) quantum coherence in the equation

$a_{[-]} = 2*i*\rho_{[-]}(0) *\exp(-i*\phi_{rec})$

in *Spin Dynamics* (2nd Ed.), p. 288 (eq. 11.48).

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

$[a_{[-\beta]} \quad a_{[-\alpha]} \quad a_{[\beta-]} \quad a_{[\alpha-]}] = 2*i*[\rho_{[-\beta]}(0) \quad \rho_{[-\alpha]}(0) \quad \rho_{[\beta-]}(0) \quad \rho_{[\alpha-]}(0)]*\exp(-i*\phi_{rec})$

as 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 the 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 **pulse()**.

phR is a quadrature receiver phase (e.g. 'x', 'y', 0, 1).

a0_V is a vector corresponding to $2*i*[\rho_{[-\beta\ldots]}(0)\ \rho_{[-\alpha\ldots]}(0)\ \rho_{[\beta-\ldots]}(0)\ \rho_{[\alpha-\ldots]}(0)\ \ldots] *\exp(-i*\phi_{rec})$.

rho_V is a vector describing which component of **a0_V** corresponds to which coherence in the density operator (e.g., 'ma' for '-α', 'bm' for 'β-').

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**.

phout = PO.phmod(phx, ii)

outputs a phase value, **phout**, from a phase table (vector), **phx**. If **ii** is smaller or equal to the length of **phx**, the 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 .

id_vec = findcoef (obj, coef_in_cell) or

id_vec = obj. findcoef (coef_in_cell)

finds particular terms that includes the coefficients defined in a cell array **coef_in_cell** and returns index numbers for these terms as a verctor **id_vec**. This function can be used with **delPO()** or **selPO()** to delete or select certain terms.

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 index number of each term can be found by **dispPO()** or can be obtained from **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.selPO(id_in)

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

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 **obj.coef** to a different expression.

obj = set_basis(obj, basis_in) or

obj = obj.set_basis (basis_in)

rewrites **obj** to a different basis **basis_in**.

output_cell = PO.v2cell(v, input_cell)

creates a cell array (**output_cell**) {v v v v v …} which has the same size of **input_cell**, which can be a cell array, vector, or matrix.

obj = UrhoUinv(obj, H, q) or

obj = obj.UrhoUinv (H, q)

calculates an evolution of a density operator ρ under a Hamiltonian *H* during a time t, 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* = ω_{nut}**I*$_x$, *H* = πJ_{IS}*2*I*$_z$*S*$_z$,) . **obj** and **H** are the PO objects corresponding to ρ and *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 a case *H*t = ω_I**I*$_z$*t + πJ_{IS}*2*I*$_z$*S*$_z$*t (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.

Technical Details

Design of Program

The code is written with the object-oriented programming (OOP) style. In manner of 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). The idea of OOP is described, for example,
https://www.mathworks.com/company/newsletters/articles/introduction-to-object-oriented-programming-in-matlab.html .

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^{Ns-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 2^{Ns-1} coefficient at the beginning of a product operator is stored in another property, **Ncoef**.

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

txt: This property stores a text output of the current system. It is automatically generated. As a default, the asterisk '*' is not displayed between spin operators and between the 2^{Ns-1} coefficient and a spin operator, e.g., 2IxSy*a. By changing a property called **asterisk_bin**, '*' is displayed, e.g., 2*Ix*Sy*a. This property can be changed by rewriting the code. The change of this property affects all **PO objects**. The former makes the **txt** and **logs** properties good looks while the later can be used to re-create a **PO object** from the text information stored in the **logs** property.
spin_label: This property defines the labels of the spin types in a system such as, I,S,K, … or I1, I2, I3, … etc.

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).

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'.

logs: This property stores logs of methods applied to the **PO object**. Note that operations with the '*', '+', '-' , '/' and '^' operators overwrite the previous logs to the current **obj.txt**.

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

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

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

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.

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

Merits to Use 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** property) 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]% Sum of three vectors. It should be [2 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*Ix should be 2 instead of 4 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

$\exp(-i\theta A)\, B \exp(i\theta A) = B\cos\theta + C\sin\theta$,

$\exp(-i\theta B)\, C \exp(i\theta B) = C\cos\theta + A\sin\theta$, and

$\exp(-i\theta C)\, A \exp(i\theta C) = A\cos\theta + B\sin\theta$

It is known that the spin angular momentum operators I_x, I_y, I_z are in cyclic commutation ($[I_z, I_x] = iI_y$) and the formula above can be used to describe an evolution of a density operator under a Hamiltonian. For example, a density operator $\rho(0) = I_x$ evolves under a chemical shift Hamiltonian $H = \omega I_z$ during a time period of t as $\rho(t) = \exp(-iHt)\,\rho(0)\,\exp(iHt) = \exp(-i\omega t I_z)\,I_x\,\exp(i\omega t I_z) = I_x\cos(\omega t) + I_y\sin(\omega t)$

Use of Master Table to Accelerate Calculation

			B		
			x	y	z
			1	2	3
A	x	1	**0**	**3**	**-2**
	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 $\exp(-i\theta A)\,B\,\exp(i\theta A) = B\cos\theta + C\sin\theta$ above and I_x, I_y and I_z. For example, in the case of $\exp(-i\theta I_z)\,I_x\,\exp(i\theta I_z) = I_x\cos\theta + I_y\sin\theta$, 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 $-C\sin\theta$ is used instead of $+C\sin\theta$. This is the basic idea of the calculation in the code. The calculation using the master table is much faster than the matrix calculation (e.g., `expm(-1i*q*Iz.M)*Ix.M*expm(1i*q*Iz.M)`).

When can Master Table be Used for Calculation?

If the two rules below are satisfied, an evolution of ρ under *H* can be calculated by 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. $H = 2I_zS_z$ but not like $4I_zS_zK_z$). Since the 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 Rule 1 and 2 but satisfies the cyclic commutation are the sets of $4I_xS_yK_z$, $4I_yS_zK_x$ and $4I_zS_xK_y$.

Rule 1 is obvious but how about Rule 2. Here is the analysis. Suppose $H = 2I_aS_b$ and $\rho = 8I_bS_bKL$ where *K*, *L* are spin operators that are different types each other in addition to *I* and *S*. Suppose $[I_a, I_b] = iI_c$ in the cyclic commutation. There are two spin-types matching between *H* and ρ, thus satisfying Rule 1, and only one of them (*I* spin) has different labels between *H* and ρ, thus satisfying Rule 2.

Then $[H, \rho] = 2I_aS_b\,8I_bS_bKL - 8I_bS_bKL\,2I_aS_b = 16\,I_aI_bS_b^2KL - 16I_bI_aS_b^2KL$

$= 16\,(I_aI_b - I_bI_a)S_b^2KL = 4(I_aI_b - I_bI_a)KL = i4I_cKL$. Note that $S_b^2 = 1/4E$ for spin-1/2.

$[4I_cKL, H] = 4I_cKL\ 2I_aS_b - 2I_aS_b\ 4I_cKL = 8I_cI_aS_bKL - 8I_aI_cS_bKL = 8(I_cI_a - I_aI_c)S_bKL = i8I_bS_bKL = i\rho$.

$[\rho, 4I_cKL] = 8I_bS_bKL\ 4I_cKL - 4I_cKL\ 8I_bS_bKL = 32I_bI_cS_bK^2L^2 - 32I_cI_bS_b\ K^2L^2 = 2(I_bI_c - I_cI_b)\ S_b = i2I_aS_b = iH$. Note that $K^2 = L^2 = 1/4E$ for spin-1/2.
Thus, H and ρ are in the cyclic commutation.

What if Rule 2 is not satisfied? In the case of $H = 2I_bS_b$ and $\rho = 8I_bS_bKL$,
$[H, \rho] = 2I_bS_b\ 8I_bS_bKL - 8I_bS_bKL\ 2I_bS_b = 16\ I_b^2S_b^2KL - 16I_b^2S_b^2KL = 0$
thus, H and ρ are not in the cyclic commutation.

In the case of $H = 2I_aS_b$ and $\rho = 8I_bS_cKL$, $[H, \rho]$ is calculated as 0 from $[2I_aS_b, 2I_bS_c] = 0$ that can be obtained from the matrix representation.
$[H, \rho] = 2I_aS_b\ 8I_bS_cKL - 8I_bS_cKL\ 2I_aS_b = 2I_aS_b\ 2I_bS_c\ 4KL - 2I_bS_c\ 4KL\ 2I_aS_b = 2I_aS_b\ 2I_bS_c\ 4KL - 2I_bS_c\ 2I_aS_b\ 4KL = [2I_aS_b, 2I_bS_c]*4KL = 0$ (See *Spin Dynamics* (2nd Ed.), p. 403, Eq. 15.24 and p. 407, Note 3).

Examples for these rules
$\rho = I_y$ 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

$\rho = S_y$ and $H = I_z$
ρ and H don't have a same type of spin (Rule 1: No) ➔ Master Table: No

$\rho = I_y$ 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

$\rho = 2I_zS_y$ 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

$\rho = 2I_zS_y$ and $H = 2I_zS_z$
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

$\rho = 2I_xS_x$ 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

$\rho = 2I_xS_x$ and $H = 2I_zS_z$

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

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_jzI_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 Two Rules in Programing Code

The evolution of ρ under *H* with the two rules above in addition to 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 = [h_1 h_2 h_3 …] corresponding to *A* and

rho_axis = [r_1 r_2 r_3 …] corresponding to *B*

to calculate a new product operator
axis_tmp = [a_1 a_2 a_3 …] corresponding to C.

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

- if both r_n and h_n are not 0, a_n takes an absolute value of the (h_n , r_n) component of the master table. If the value from the master table is negative, the sign of the coefficient is inverted.
- if r_n is not 0 but h_n is 0, a_n is same as r_n. This is for cases such as

  $\rho = 2I_zS_z$ ([r_1 r_2] = [3 3]) and $H = I_y$ ([h_1 h_2] = [2 0]) ➔ C: $2I_xS_z$ ([a_1 a_2] = [a_1 r_2] = [1 3])
- if r_n is 0 but h_n is not 0, a_n is same as h_n. This is for cases such as

  $\rho = I_x$ ([r_1 r_2] = [1 0]) and $H = 2I_zS_z$ ([h_1 h_2] = [3 3]) ➔ C: $2I_ySz$ ([a_1 a_2] = [a_1 h_2] = [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 **PO** 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. Conversely, 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

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

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.

Sørensen, O. W.; Eich, G. W., Levitt, M. H.; Bodenhausen, G.; Ernst, R. R., *Product Operator Formalism for the Description of NMR Pulse Experiments*, *Prog. NMR Spectros.* **1983**, *16*, 163-192.

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.