

---

# **SpinW documentation Documentation**

***Release 0.1***

**Sandor Toth**

**Aug 16, 2017**



---

## Contents

---

<b>1</b>	<b>Introduction to SpinW</b>	<b>1</b>
<b>2</b>	<b>SW Class</b>	<b>3</b>
<b>3</b>	<b>Crystal structure</b>	<b>9</b>
<b>4</b>	<b>List of functions</b>	<b>11</b>
<b>5</b>	<b>Indices and tables</b>	<b>13</b>



---

## Introduction to SpinW

---

SpinW is a Matlab library that can solve spin Hamiltonians using mean field theory and linear spin wave theory. It is optimised for spin wave calculation on complex lattices that one often encounter experimentally.

### Model

In short SpinW can solve the following spin Hamiltonian using classical and quasi classical numerical methods:

$$\mathcal{H} = \sum_{i,j} \mathbf{s}_i^T \cdot J_{ij} \cdot \mathbf{s}_j + \sum_i \mathbf{s}_i^T \cdot A_i \cdot \mathbf{s}_i + \mu_B \mathbf{B}^T \cdot \sum_i \mathbf{g}_i \cdot \mathbf{s}_i$$

where  $\mathbf{s}_i$  are spin vector operators,  $J_{ij}$  are 3x3 exchange matrices describing coupling between spins,  $A_{ij}$  are 3x3 single ion anisotropy matrices,  $\mathbf{B}$  is external magnetic field,  $g_i$  are the g-tensors with 3x3 elements and  $\mu_B$  is the Bohr-magneton. The 3x3 matrices are necessary to include all possible anisotropic and antisymmetric interactions.

### Features

#### Crystal structures

- definition of crystal lattice with arbitrary unit cell, using symmetry operators
- definition of non-magnetic atoms and magnetic atoms with arbitrary moment size
- publication quality plotting of crystal structures (atoms, labels, axes, surrounding polyhedron, anisotropy ellipsoids, DM vector, etc.)

#### Magnetic structures

- definition of 1D, 2D and 3D magnetic structures
- representation of single-Q incommensurate structures using rotating coordinate system or complex basis vectors

- generation of magnetic structures on a magnetic supercell
- plotting of magnetic structures

## Magnetic interactions

- simple assignment of magnetic interactions to bonds based on bond length
- possible interactions: Heisenberg, Dzyaloshinskii-Moriya, anisotropic and general 3x3 exchange tensor
- arbitrary single ion anisotropy tensor (easy-plane, easy-axis, etc.)
- Zeeman energy in homogeneous magnetic field including arbitrary g-tensor
- calculation of symmetry allowed elements of the above tensors based on the crystallographic space group

## Magnetic ground state optimisation

- minimization of the classical energy assuming single-Q magnetic structure
- simulated annealing using the Metropolis algorithm on large magnetic supercells
- calculation of properties in thermodynamical equilibrium (heat capacity, magnetic susceptibility, etc.)
- magnetic structure factor calculation using FFT
- simulation of magnetic neutron diffraction and diffuse scattering

## Spin wave simulation

- solution for commensurate and single-Q incommensurate magnetic structures
- calculation of spin wave dispersion, spin-spin correlation functions
- calculation of neutron scattering cross section for unpolarized neutrons including the magnetic form factor
- calculation of polarized neutron scattering cross sections
- possibility to include different moment sizes for different magnetic atoms
- calculation of powder averaged spin wave spectrum

## Plotting of spin wave spectrum

- plotting of dispersions and correlation functions
- calculation and plotting of the convoluted spectra for direct comparison with inelastic neutron scattering
- full integration into Horace for plotting and comparison with time of flight neutron data, see <http://horace.isis.rl.ac.uk>

## Fitting spin wave spectra

- possible to fit any parameter of the Hamiltonian
- robust fitting, even when the number of simulated spin wave modes differs from the measured number of modes

## CHAPTER 2

---

### SW Class

---

To perform calculation using the SpinW library, we need to create an object (sw class type). It stores all the necessary parameters for the calculation (crystal structure, interactions, magnetic structure, etc.). In the object oriented programming dictionary, the data stored in an object, are called properties. Beside the data, the object also has assigned functions that perform different computations on the object data. These functions are called methods and they take the object as first input argument. To create an sw class object you can simply type:

```
modell = sw
```

### Properties

The output of the previous command shows all the data fields of modell. Each data field has an initial value and any of them can be modified directly:

```
modell.lattice.lat_const = [3 5 5];
```

The above command directly modifies the lattice parameters of the lattice. Modifying propoerties directly is quick and very flexible but prone to error. The most common mistake is that the new values are not the same data type as the original ones. For example the field that stores the lattice space group is integer type:

```
class(modell.lattice.sym)
```

Thus if we want to change it directly, we need an integer number:

```
modell.lattice.sym = int32(5);
```

This will change the crystal space group to 'C 2'. To avoid most common mistakes, there are several methods (functions) for modifying the above properties that also perform additional error checking and makes certain input conversions. For example all lattice related properties can be modified using the genlattice() function:

```
modell.genlattice('lat_const',[3 5 5],'sym','C 2','angled',[90 90 90])
```

The alternative usage of the above function is the following:

```
genlattice(model1, 'lat_const', [3 5 5], 'sym', 'C 2', 'angled', [90 90 90])
```

This reflects better the input argument structure. The first argument is the sw object 'model1'. After the first argument comes option name and value pairs. The first options is 'lat\_const' and the value it expects is a vector with 3 elements if the input vector has different length, the function throws an error. The second option is 'sym' that also accepts string input (name of the space group) that is automatically converted to the index of the space group and stored in model1:

```
model1.lattice.sym
```

The last option is 'angled' that requires a vector with three elements and defines the alpha, beta, gamma lattice angles in degree. This will be converted into radian and stored:

```
model1.lattice.angle
```

## List of properties

The sw object properties store all the information necessary for the spin wave calculation. There are eight public properties of the sw class each with several subfields:

- `sw.lattice`
- `sw.unit_cell`
- `sw.twin`
- `sw.matrix`
- `sw.single_ion`
- `sw.coupling`
- `sw.mag_str`
- `sw.unit`

### sw.lattice

The **lattice** field stores the lattice parameters and space group information. The subfields are:

**lat\_const** Lattice constants in a row vector with three elements ( $a$ ,  $b$  and  $c$ ) in units.

**angle**  $\alpha$ ,  $\beta$  and  $\gamma$  angles in a row vector with three elements in radian units.

**sym** Crystal space group, single integer (int32 type) that gives the line number in the symmetry.dat file. This file stores the generators of the space group and by default contains the 230 crystallographic space groups with standard settings.

See also `sw.genlattice()`, `sw.abc()`, `sw.basisvector()`, `sw.nosym()`.

### sw.unit\_cell

The **unit\_cell** field stores the information on the atoms in the crystallographic unit cell. Subfields are:

**r** Atomic positions in lattice units, matrix with dimensions of [3 nAtom], in lattice units.

**S** Spin (or angular momentum) quantum number of the atoms, row vector with nAtom number of elements. Non-magnetic atoms have  $S = 0$ .

**label** String label of the atoms in a cell with dimensions of [1 nAtom].

**color** Color of the atoms for plotting. Stored in a matrix with dimensions of [3 nAtom], every column is an RGB code (int32 numbers between 0-255).



See also `sw.addatom()`, `sw.atom()`, `sw.matom()`, `sw.newcell()`.

#### `sw.twin`

The **twin** field stores information on crystallographic twins. The subfields are:

**rotr** Rotation matrices in the xyz coordinate system for every twin, stored in a matrix with dimensions of [3 3 nTwin].

**vol** Relative volume of the twins, stored in a row vector with nTwin elements.

See also `sw.addtwin()`, `sw.twinq()`, `sw.ntwin()`.

#### `sw.matrix`

The **matrix** field stores 3x3 matrices that can be referenced in the magnetic Hamiltonian. The subfields are:

**mat** It stores the actual values of 3x3 matrices stacked along the third dimension with dimensions of [3 3 nMat].

**color** RGB color codes assigned for every matrix, stored in a matrix with dimensions of [3 nMatrix], each column is an [R;G;B] code with int32 numbers between 0 and 255.

**label** Label for every matrix, stored as strings in a cell with dimensions of [1 nCell].

See also `sw.addmatrix()`, `sw.getmatrix()`, `sw.setmatrix()`.

#### `sw.single_ion`

The **single\_ion** field stores the single ion expression of the Hamiltonian. The subfields are:

**aniso** Row vector contains nMagAtom integers, each integer assigns one of the matrices from the `sw.matrix` field to a magnetic atom in the `sw.matom()` list as a single ion anisotropy. Zero means no assigned anisotropy matrix.

**g** Row vector of nMagAtom integers, each integer assigns one of the matrices from the `sw.matrix` field to a magnetic atom in the `sw.matom()` list as a g-tensor.

**field** External magnetic field stored in a row vector with the 3 components in the xyz coordinate system, default unit is Tesla.

**T** Temperature, scalar, default unit is Kelvin.

See also `sw.addaniso()`, `sw.addg()`, `sw.getmatrix()`, `sw.setmatrix()`, `sw.intmatrix()`, `sw.field()`.

#### `sw.coupling`

The **coupling** field stores the list of bonds. Where each bond is defined between two magnetic atom. Each bond has a direction, it points from atom 1 to atom 2. In the subfiels every column defines a bond, the subfields are:

**dl** Translation vector between the unit cells of the two interacting spins, stored in a matrix of integer numbers with dimensions of [3 nCoupling].

**atom1** Stores the index of atom 1 for every bond pointing to the list of magnetic atoms in `sw.matom()` list, stored in a row vector with nCoupling elements.

**atom2** Index of atom 2 for each bond.

**mat\_idx** Integer indices selecting exchange matrices from `sw.matrix` field for every bond, stored in a matrix with dimensions of [3 nCoupling]. Maximum three matrices per bond can be assigned (zeros for no coupling).

**idx** An integer index for every bond, an increasing number with bond length. Symmetry equivalent bonds have the same index.

See also `sw.gencoupling()`, `sw.addcoupling()`.

#### `sw.mag_str`

The **mag\_str** field stores the magnetic structure. It can store single-Q structures or multi-Q structures on a

magnetic supercell. Strictly speaking it stores the expectation value of the spin of each magnetic atom. The magnetic moment directions are given by  $g_i \cdot \langle S_i \rangle$ . The subfields are:

**S** It stores the spin directions for every magnetic atom either in the crystallographic unit cell or in a magnetic supercell in a matrix with dimensions of [3 nMagExt]. Every column stores  $\langle S_x \rangle, \langle S_y \rangle, \langle S_z \rangle$  spin components in the xyz coordinate system. The number of spins in the supercell is:

```
nMagExt = nMagAtom*prod(mag_str.N_ext);
```

**k** Magnetic ordering wave vector stored in a row vector with 3 components in reciprocal lattice units. Default value is [0 0 0].

**n** Normal vector to the rotation of the moments in case of non-zero ordering wave vector (single-Q magnetic structures), row vector with three elements. The components are in the xyz coordinate system. Default value is [0 0 1].

**N\_ext** Dimensions of the magnetic supercell in lattice units stored in a row vector with three elements. Default value is [1 1 1] if the magnetic cell is identical to the crystallographic cell. The three elements extends the cell along the  $a, b, c$  axes.

See also `sw.genmagstr()`, `sw.optmagstr()`, `sw.anneal()`, `sw.nmagext()`, `sw.structfact()`, `sw.optmagsteep()`.

#### **sw.unit**

The **unit** field stores the conversion factors between energy, magnetic field and temperature units in the Hamiltonian. Defaults units are meV, Tesla and Kelvin for energy, magnetic field and temperature respectively. To use identical units for energy, magnetic field and temperature use 1 for each subfields. The subfields are:

**kB** Boltzmann constant, default value is 0.0862 meV/K.

**muB** Bohr magneton, default value is 0.0579 meV/T.

## Methods

In line with the above example the general argument structure of the method functions is one of the following:

```
function(obj, 'Option1', Value1, 'Option2', Value2, ...)  
function(obj, Value1, Value2, ...)
```

The first type of argument list is used for functions that require variable number of input parameters with default values. The second type of argument structure is used for functions that require maximum up to three fixed input parameter. Every method has help that can be called by one of the following methods:

- selecting the function name in the Editor/Command Window and pressing F1
- in the Command Window typing for example:

```
help sw.genlattice
```

This shows the help of the `genlattice()` function in the Command Window. To open the help in a separate window you need to write:

```
doc sw.genlattice
```

To unambiguously identify the functions it is useful to refer them as `sw.function()` this way matlab knows which function to select from several that has the same name. For example the `plot()` function is also defined for the `sw` class. However by writing:

```
help plot
```

we get the help for the standard Matlab plot function. To get what we want use:

```
help sw.plot
```

By the way this function is one of the most usefull ones. It can show effectively all information stored in the sw object by plotting crystal structure, couplings, magnetic structure etc. Calling it on an empty object shows only the unit cell:

```
plot(model1)
```

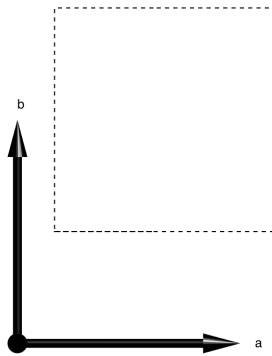


Fig. 2.1: *Plot of empty sw object.*

As you might noticed, there is an alternative calling of any method function: `obj.function(...)`, this is just equivalent to the previous argument structures.

## Copy

The sw class belong to the so called handle class. It means in short that the `model1` variable is just a pointer to the memory where the class is stored. Thus by doing the following:

```
model2 = model1;
```

will only duplicate the pointer but not the values stored in the sw object. Thus if I change something in `model1`, `model2` will change as well. To clone the object (the equivalent of the usual '=' operation in Matlab) the `sw.copy()` function is necessary:

```
model2 = copy(model1);
```



## CHAPTER 3

---

### Crystal structure

---



## List of functions

**class sw**

**sw.genlattice** (*obj*, 'option1', *value1*, 'option2', *value2*...)

Generates crystal lattice from given parameters.

**Input**

**obj** sw class object.

**Options**

**angled**  $\alpha, \beta, \gamma$  angles in degree, row vector with three elements.

**angle**  $\alpha, \beta, \gamma$  angles in radian, row vector with three elements.

**lat\_const**  $a, b, c$  lattice parameters, row vector with three elements.

**sym** Space group index, or space group name (string).

**Description**

Alternatively the lattice parameters can be given directly when the sw object is created using: `sw(inpStr)`, where struct contains the fields with initial parameters, e.g.:

```
inpStr.lattice.lat_const = [3 3 4];
```

The `sym` option points to the appropriate line in the `symmetry.dat` file, where every line defines a space group by its generators. The first 230 lines contains all crystallographic space groups with standard setting from the International Tables of Crystallography. Additional lines can be added to the `symmetry.dat` file using the `sw_addsym()` function. Every line in the `symmetry.dat` file can be referenced by either its line index or by its label (string).

If the `sym` option is 0, no symmetry will be used. The `gencoupling()` method will determine the equivalent bonds based on bond length.

**Output**

The `lattice` field will be changed based on the input, the lattice constants stored directly and the optional space group string is converted to the integer type index.

### Example

```
crystal = sw;  
crystal.genlattice('lat_const', [3 3 4], 'angled', [90 90 120], 'sym', 'P 6')  
crystal.genlattice('lat_const', [3 3 4], 'angled', [90 90 120], 'sym', 168)
```

The two lines are equivalent, both will create hexagonal lattice, with  $P 6$  space group.

TEST2.



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



## C

`coupling` (sw attribute), 5

## G

`genlattice()` (sw method), 11

## L

`lattice` (sw attribute), 4

## M

`mag_str` (sw attribute), 5

`matrix` (sw attribute), 5

## S

`single_ion` (sw attribute), 5

`sw` (built-in class), 11

## T

`twin` (sw attribute), 5

## U

`unit` (sw attribute), 6

`unit_cell` (sw attribute), 4