Spirit Documentation

Gideon Mueller and contributors

Jul 04, 2019

Contents

CHAPTER 1

SPIRIT

SPIN SIMULATION FRAMEWORK

Logo

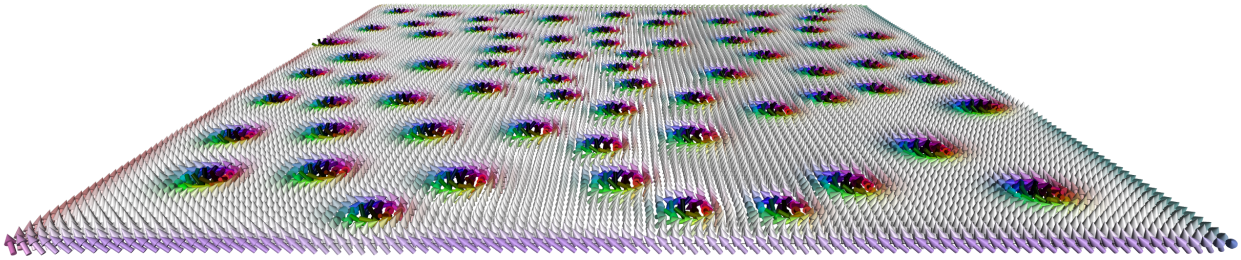
Core Library:

Python package: PyPI version

The code is released under MIT License. If you intend to *present and/or publish* scientific results or visualisations for which you used Spirit, please cite `G. P. Müller et al., Phys. Rev. B 99, 224414 (2019)` and read the docs/REFERENCE.md.

This is an open project and contributions and collaborations are always welcome!! See docs/CONTRIBUTING.md on how to contribute or write an email to g.mueller@fz-juelich.de For contributions and affiliations, see docs/CONTRIBUTORS.md.

Please note that a version of the *Spirit Web interface* is hosted by the Research Centre Jülich at http://juspin.de



Skyrmions

1.1 Contents

1.2 Introduction

1.2.1 A modern framework for magnetism science on clusters, desktops & laptops and even your Phone

Spirit is a **platform-independent** framework for spin dynamics, written in C++11. It combines the traditional cluster work, using using the command-line, with modern visualisation capabilites in order to maximize scientists' productivity.

> "It is unworthy of excellent men to lose hours like slaves in the labour of calculation which could safely be relegated to anyone else if machines were used."
>
> - Gottfried Wilhelm Leibniz

Our goal is to build such machines. The core library of the *Spirit* framework provides an **easy to use API**, which can be used from almost any programming language, and includes ready-to-use python bindings. A **powerful desktop user interface** is available, providing real-time visualisation and control of parameters.

1.2.2 *Physics Features*

- Atomistic Spin Lattice Heisenberg Model including also DMI and dipole-dipole
- **Spin Dynamics simulations** obeying the Landau-Lifschitz-Gilbert equation
- Direct **Energy minimisation** with different solvers
- **Minimum Energy Path calculations** for transitions between different spin configurations, using the GNEB method

1.2.3 *Highlights of the Framework*

- Cross-platform: everything can be built and run on Linux, OSX and Windows
- Standalone core library with C API which can be used from almost any programming language
- **Python package** making complex simulation workflows easy
- Desktop UI with powerful, live **3D visualisations** and direct control of most system parameters
- Modular backends including **parallelisation on GPU** (CUDA) and **CPU** (OpenMP)

1.2.4 *Documentation*

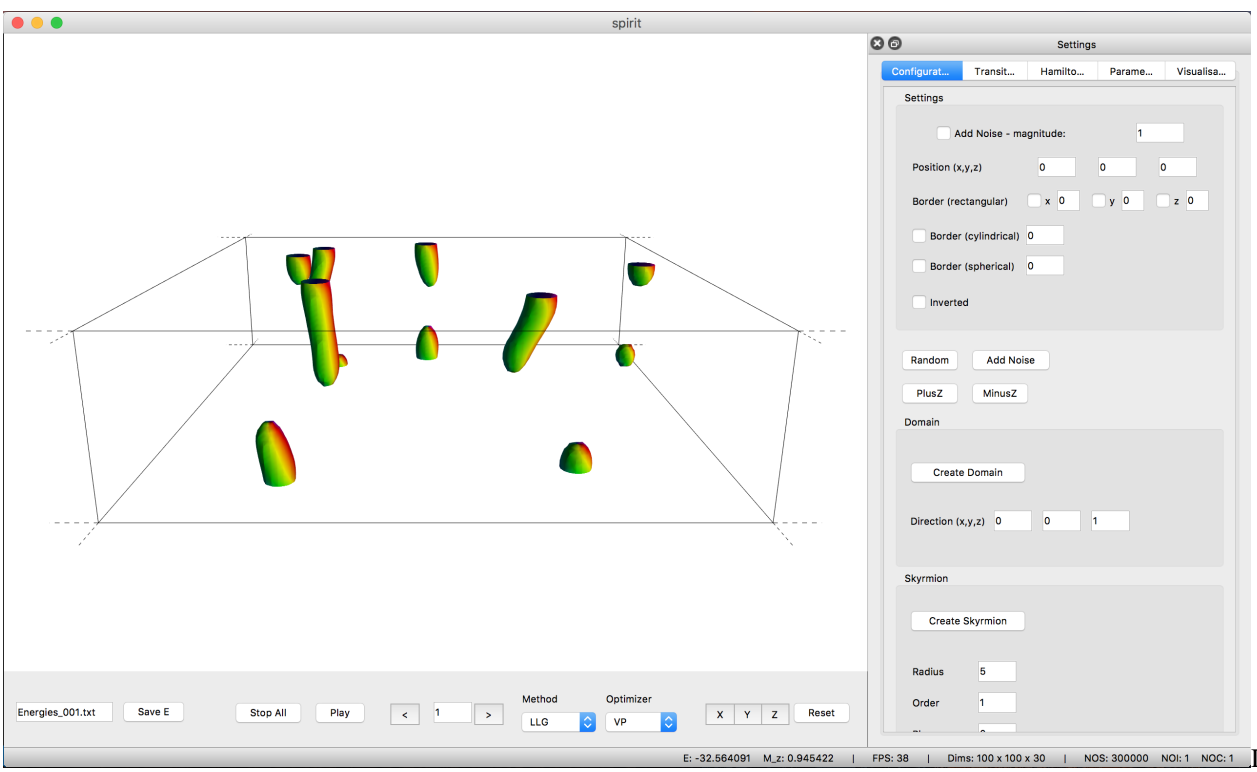
More details may be found at spirit-docs.readthedocs.io or in the Reference section including

- Unix/OSX build instructions
- Windows build instructions
- Input File Reference

There is also a Wiki, hosted by the Research Centre Jülich.

1.3 Getting started with the Desktop Interface

See the build instructions for Unix/OSX or Windows on how to get the desktop user interface.



Desktop UI with Isosurfaces in a thin layer

The user interface provides a powerful OpenGL visualisation window using the VFRendering library. It provides functionality to

- Control Calculations
- Locally insert Configurations (homogeneous, skyrmions, spin spiral, . . .)
- Generate homogeneous Transition Paths
- Change parameters of the Hamiltonian
- Change parameters of the Method and Solver
- Configure the Visualization (arrows, isosurfaces, lighting, . . .)

See the UI-QT Reference for the key bindings of the various features.

Unfortunately, distribution of binaries for the Desktop UI is not possible due to the restrictive license on QT-Charts.

1.4 Getting started with the Python Package

To install the *Spirit python package*, either build and install from source (Unix/OSX, Windows) or simply use

```
pip install spirit
```

With this package you have access to powerful Python APIs to run and control dynamics simulations or optimizations. This is especially useful for work on clusters, where you can now script your workflow, never having to re-compile when testing, debugging or adding features.

The most simple example of a **spin dynamics simulation** would be

```
from spirit import state, simulation
with state.State("input/input.cfg") as p_state:
    simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_SIB)
```

Where `SOLVER_SIB` denotes the semi-implicit method B and the starting configuration will be random.

To add some meaningful content, we can change the **initial configuration** by inserting a Skyrmion into a homogeneous background:

```
def skyrmion_on_homogeneous(p_state):
    from spirit import configuration
    configuration.plus_z(p_state)
    configuration.skyrmion(p_state, 5.0, phase=-90.0)
```

If we want to calculate a **minimum energy path** for a transition, we need to generate a sensible initial guess for the path and use the **GNEB method**. Let us consider the collapse of a skyrmion to the homogeneous state:

```
from spirit import state, chain, configuration, transition, simulation

### Copy the system and set chain length
chain.image_to_clipboard(p_state)
noi = 7
chain.set_length(p_state, noi)

### First image is homogeneous with a Skyrmion in the center
```

(continues on next page)

(continued from previous page)

```
configuration.plus_z(p_state, idx_image=0)
configuration.skyrmion(p_state, 5.0, phase=-90.0, idx_image=0)
simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_VP, idx_image=0)
### Last image is homogeneous
configuration.plus_z(p_state, idx_image=noi-1)
simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_VP, idx_image=noi-
↪1)

### Create transition of images between first and last
transition.homogeneous(p_state, 0, noi-1)

### GNEB calculation
simulation.start(p_state, simulation.METHOD_GNEB, simulation.SOLVER_VP)
```

where `SOLVER_VP` denotes a direct minimization with the velocity projection algorithm.

You may also use *Spirit* order to **extract quantitative data**, such as the energy.

```
def evaluate(p_state):
    from spirit import system, quantities
    M = quantities.get_magnetization(p_state)
    E = system.get_energy(p_state)
    return M, E
```

Obviously you may easily create significantly more complex workflows and use Python to e.g. pre- or post-process data or to distribute your work on a cluster and much more!

CHAPTER 2

Spirit Desktop UI

Logo

The cross-platform QT desktop user interface provides a productive tool for Spin simulations, providing powerful real-time visualisations and access to simulation parameters, as well as other very useful features.

See the build instructions for *Unix/OSX* and *Windows* for information on how to build the graphical user interface on your machine.

2.1 Physics features

Insert Configurations:

- White noise
- (Anti-) Skyrmions
- Domains
- Spin Spirals

You may manipulate the Hamiltonian as well as simulation parameters and your output file configuration:

You may start and stop simulation and directly interact with a running simulation.

- LLG Simulation: Dynamics and Minimization
- GNEB: create transitions and calculate minimum energy paths

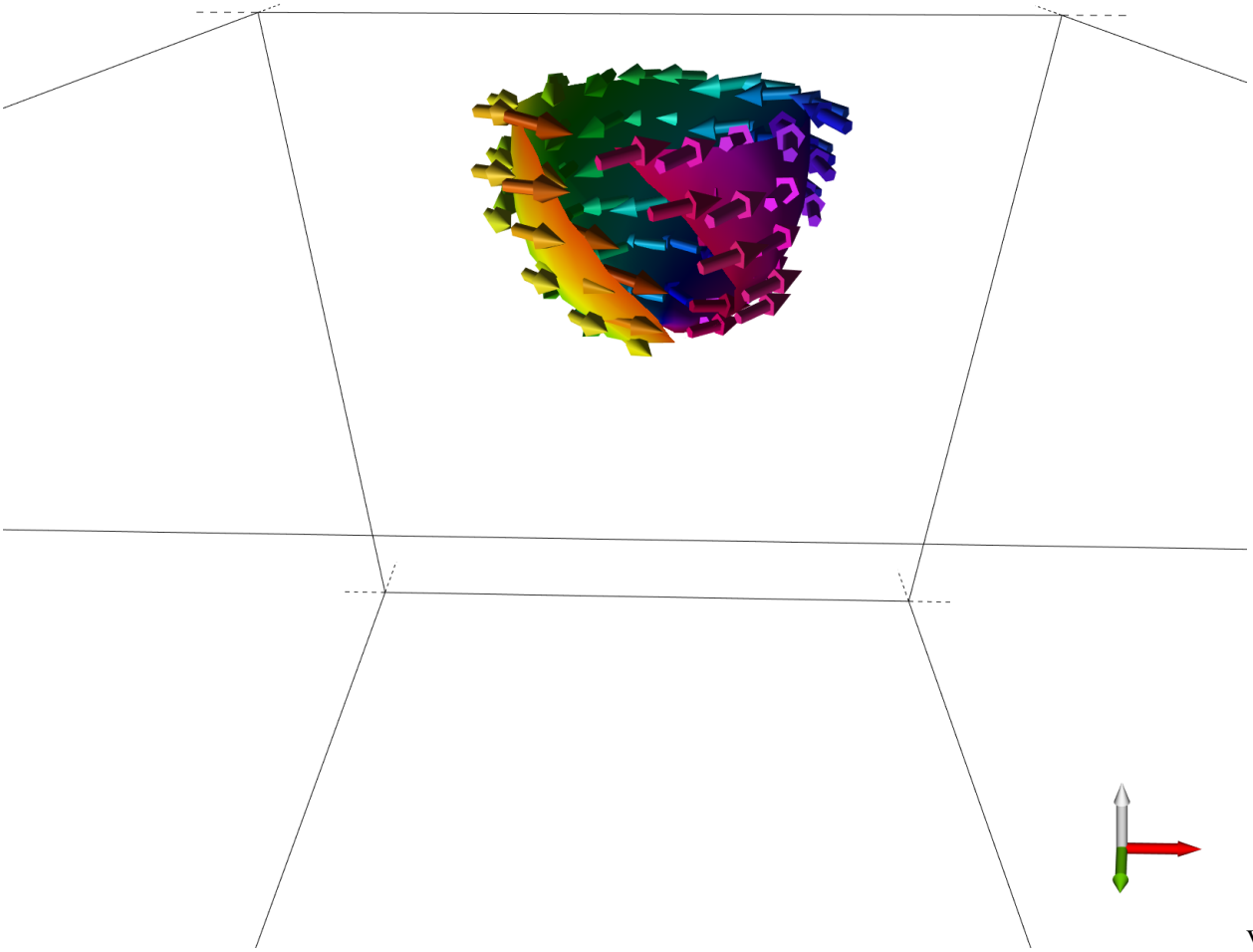
By copying and inserting spin systems and manipulating them you may create arbitrary transitions between different spin states to use them in GNEB calculations. Furthermore you can choose different images to be climbing or falling images during your calculation.

2.2 Real-time visualisation

This feature is most powerful for 3D systems but shows great use for the analysis of dynamical processes and understanding what is happening in your system during a simulation instead of post-processing your data.

- Arrows, Surface (2D/3D), Isosurfaces
- Spins or eff. field
- Every n'th arrow
- Spin sphere
- Directional & position filters
- Various colourmaps

You can also create quite complicate visualisations by combining these different features in order to visualise complex states in 3D systems:



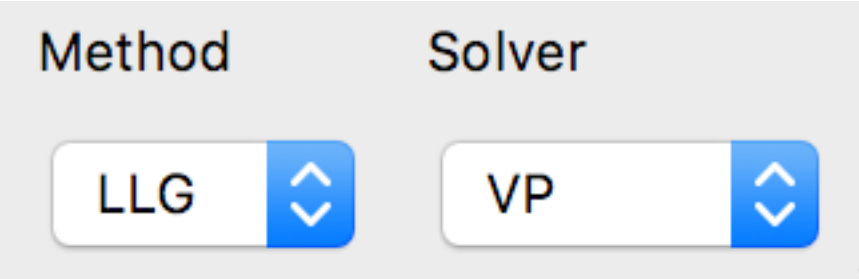
Visualisation of a complicated state

2.3 Additional features

- Drag mode: drag, copy, insert, change radius
- Screenshot
- Read configuration or chain
- Save configuration or chain

2.4 How to perform an energy minimisation

The most straightforward way of minimising the energy of a spin configuration is to use the LLG method and the velocity projection (VP) solver:

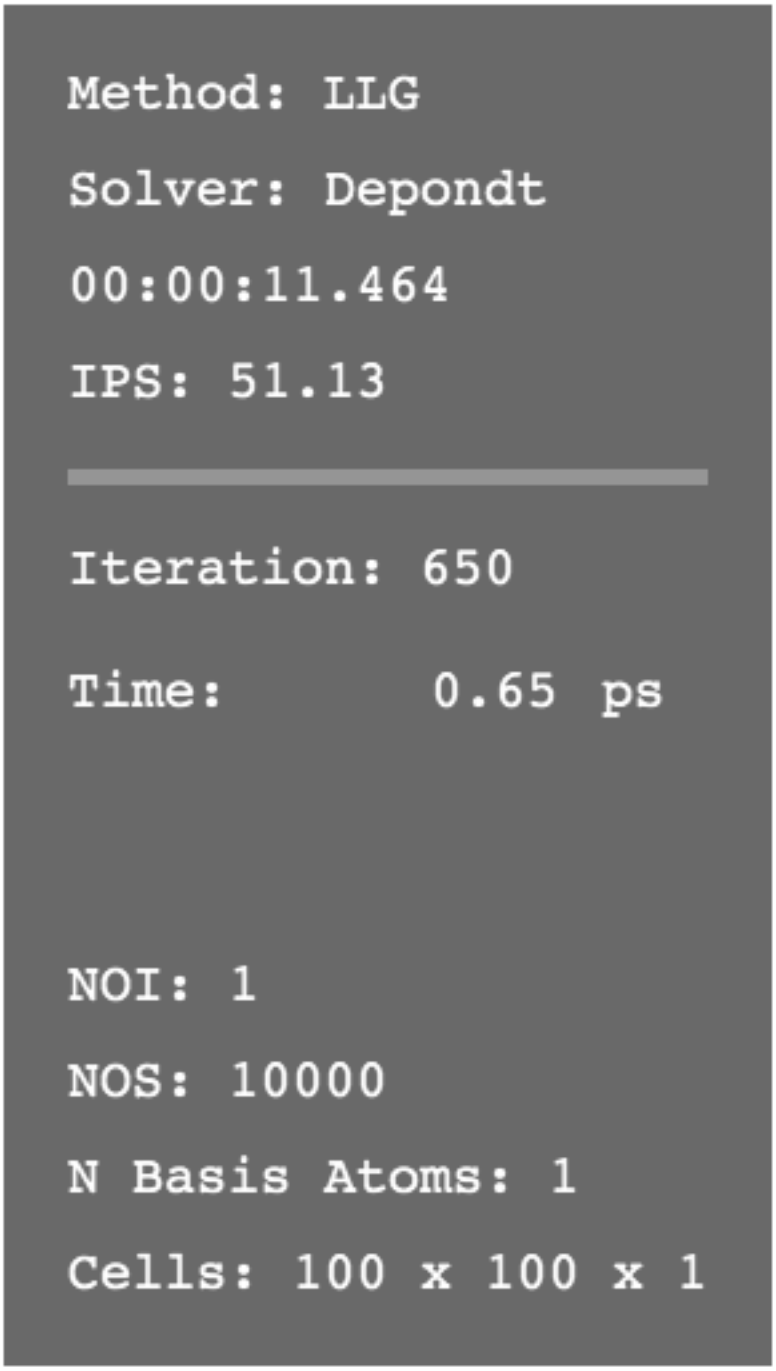


GUI controls

By pressing "start" or the space bar, the calculation is started.

2.5 How to perform an LLG dynamics calculation

To perform a dynamics simulation, use for example the Depondt solver. In this case, parameters such as temperature or spin current will have an effect and the passed time has physical meaning:

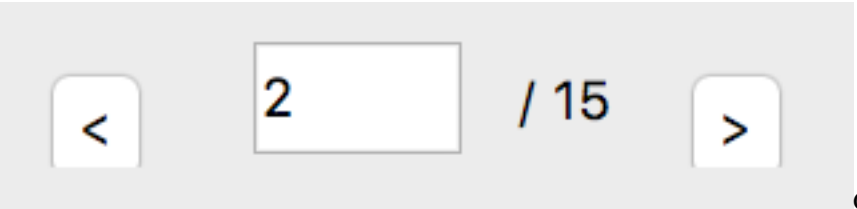


GUI info panel

2.6 How to perform a GNEB calculation

Select the GNEB method and the VP solver.

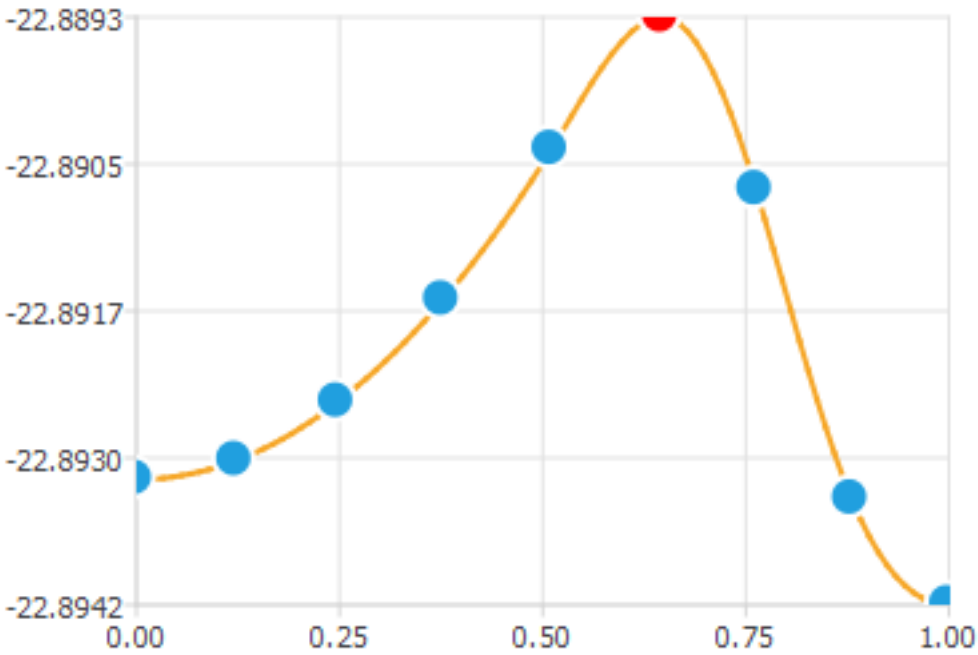
In order to perform a geodesic nudged elastic band (GNEB) calculation, you need to first create a chain of spin systems, in this context called "images". You can do this by pressing `ctrl+c` to "copy" the current image and then `ctrl+rightarrow` multiple times to insert the copy into the chain until the desired number of images is reached. The GUI will show the length of the chain:



GUI controls

You can use the buttons or the right and left arrow keys to switch between images.

A data plot is available to visualise your chain of spin systems. The interpolated energies become available when you run a GNEB calculation.



GNEB Transition Plot

2.7 Key bindings

Note that some of the keybindings may only work correctly on US keyboard layout.

2.7.1 UI Controls

2.7.2 Camera Controls

2.7.3 Control Simulations

2.7.4 Manipulate the current images

2.7.5 Visualisation

2.7.6 Manipulate the chain of images

Home

CHAPTER 3

SPIRIT INPUT FILES

The following sections will list and explain the input file keywords.

1. *General Settings and Log*
2. *Geometry*
3. *Heisenberg Hamiltonian*
4. *Gaussian Hamiltonian*
5. *Method Output*
6. *Method Parameters*
7. *Pinning*
8. *Disorder and Defects*

3.1 General Settings and Log

```
### Add a tag to output files (for timestamp use "<time>")
output_file_tag         some_tag
```

```
### Save input parameters on creation of State
log_input_save_initial  0
### Save input parameters on deletion of State
log_input_save_final    0

### Print log messages to the console
log_to_console    1
### Print messages up to (including) log_console_level
log_console_level 5

### Save the log as a file
```

(continues on next page)

(continued from previous page)

```
log_to_file    1
### Save messages up to (including) log_file_level
log_file_level 5
```

Except for `SEVERE` and `ERROR`, only log messages up to `log_console_level` will be printed and only messages up to `log_file_level` will be saved. If `log_to_file`, however is set to zero, no file is written at all.

3.2 Geometry

The Geometry of a spin system is specified in form of a bravais lattice and a basis cell of atoms. The number of basis cells along each principal direction of the basis can be specified. *Note:* the default basis is a single atom at (0,0,0).

3D simple cubic example:

```
### The bravais lattice type
bravais_lattice sc

### µSpin
mu_s 2.0

### Number of basis cells along principal
### directions (a b c)
n_basis_cells 100 100 10
```

If you have a nontrivial basis cell, note that you should specify `mu_s` for all atoms in your basis cell (see the next example).

2D honeycomb example:

```
### The bravais lattice type
bravais_lattice hex2d

### The basis cell in units of bravais vectors
### n            No of spins in the basis cell
### 1.x 1.y 1.z  position of spins within basis
### 2.x 2.y 2.z  cell in terms of bravais vectors
basis
2
0   0                       0
0.86602540378443864676 0.5 0

### µSpin
mu_s 2.0 1.0

### Number of basis cells along principal
### directions (a b c)
n_basis_cells 100 100 1
```

The bravais lattice can be one of the following:

Alternatively it can be input manually, either through vectors or as the bravais matrix:

```
### bravais_vectors or bravais_matrix
###   a.x a.y a.z       a.x b.x c.x
###   b.x b.y b.z       a.y b.y c.y
```

(continues on next page)

(continued from previous page)

```
###   c.x c.y c.z       a.z b.z c.z
bravais_vectors
1.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
```

A lattice constant can be used for scaling:

```
### Scaling constant
lattice_constant 1.0
```

Note that it scales the Bravais vectors and therefore the translations, atom positions in the basis cell and potentially – if you specified them in terms of the Bravais vectors – also the anisotropy and DM vectors.

Units:

The Bravais vectors (or matrix) are specified in Cartesian coordinates in units of Angstrom. The basis atoms are specified in units of the Bravais vectors.

The atomic moments `mu_s` are specified in units of the Bohr magneton `mu_B`.

3.3 Heisenberg Hamiltonian

To use a Heisenberg Hamiltonian, use either `heisenberg_neighbours` or `heisenberg_pairs` as input parameter after the `hamiltonian` keyword.

General Parameters:

```
### Hamiltonian Type (heisenberg_neighbours, heisenberg_pairs, gaussian)
hamiltonian              heisenberg_neighbours

### Boundary conditions (in a b c) = 0(open), 1(periodical)
boundary_conditions      1 1 0

### External magnetic field [T]
external_field_magnitude 25.0
external_field_normal    0.0 0.0 1.0

### Uniaxial anisotropy constant [meV]
anisotropy_magnitude     0.0
anisotropy_normal        0.0 0.0 1.0

### Dipole-dipole interaction caclulation method
### (none, fft, fmm, cutoff)
ddi_method               fft

### DDI number of periodic images (fft and fmm) in (a b c)
ddi_n_periodic_images    4 4 4

### DDI cutoff radius (if cutoff is used)
ddi_radius               0.0
```

Anisotropy: By specifying a number of anisotropy axes via `n_anisotropy`, one or more anisotropy axes can be set for the atoms in the basis cell. Specify columns via headers: an index `i` and an axis `Kx Ky Kz` or `Ka Kb Kc`, as well as optionally a magnitude `K`.

Dipole-Dipole Interaction: Via the keyword `ddi_method` the method employed to calculate the dipole-dipole interactions is specified.

```
  `none`   -  Dipole-Dipole interactions are neglected
  `fft`    -  Uses a fast convolution method to accelerate the calculation
  `cutoff` -  Lets only spins within a maximal distance of 'ddi_radius' interact
  `fmm`    -  Uses the Fast-Multipole-Method (NOT YET IMPLEMENTED!)
```

If the `cutoff`-method has been chosen the cutoff-radius can be specified via `ddi_radius`. *Note:* If `ddi_radius` < 0 a direct summation (i.e. brute force) over the whole system is performed. This is very inefficient and only encouraged for very small systems and/or unit-testing/debugging.

If the boundary conditions are periodic `ddi_n_periodic_images` specifies how many images are taken in the respective direction. *Note:* The images are appended on both sides (the edges get filled too) i.e. 1 0 0 -> one image in +a direction and one image in -a direction

Neighbour shells:

Using `hamiltonian heisenberg_neighbours`, pair-wise interactions are handled in terms of (isotropic) neighbour shells:

```
### Hamiltonian Type (heisenberg_neighbours, heisenberg_pairs, gaussian)
hamiltonian       heisenberg_neighbours

### Exchange: number of shells and constants [meV / unique pair]
n_shells_exchange 2
jij               10.0  1.0

### Chirality of DM vectors (+/-1=bloch, +/-2=neel)
dm_chirality      2
### DMI: number of shells and constants [meV / unique pair]
n_shells_dmi      2
dij                       6.0 0.5
```

Note that pair-wise interaction parameters always mean energy per unique pair (not per neighbour).

Specify Pairs:

Using `hamiltonian heisenberg_pairs`, you may input interactions explicitly, in form of unique pairs, giving you more granular control over the system and the ability to specify non-isotropic interactions:

```
### Hamiltonian Type (heisenberg_neighbours, heisenberg_pairs, gaussian)
hamiltonian       heisenberg_pairs

### Pairs
n_interaction_pairs 3
i j   da db dc    Jij   Dij  Dijx Dijy Dijz
0 0    1  0  0   10.0   6.0   1.0  0.0  0.0
0 0    0  1  0   10.0   6.0   0.0  1.0  0.0
0 0    0  0  1   10.0   6.0   0.0  0.0  1.0

### Quadruplets
n_interaction_quadruplets 1
i    j  da_j  db_j  dc_j    k  da_k  db_k  dc_k    l  da_l  db_l  dc_l    Q
0    0  1     0     0       0  0     1     0       0  0     0     1       3.0
```

Note that pair-wise interaction parameters always mean energy per unique pair (not per neighbour).

Pairs: Leaving out either exchange or DMI in the pairs is allowed and columns can be placed in arbitrary order. Note that instead of specifying the DM-vector as `Dijx Dijy Dijz`, you may specify it as `Dija Dijb Dijc` if you

prefer. You may also specify the magnitude separately as a column `Dij`, but note that if you do, the vector (e.g. `Dijx Dijy Dijz`) will be normalized.

Quadruplets: Columns for these may also be placed in arbitrary order.

Separate files: The anisotropy, pairs and quadruplets can be placed into separate files, you can use `anisotropy_from_file`, `pairs_from_file` and `quadruplets_from_file`.

If the headers for anisotropies, pairs or quadruplets are at the top of the respective file, it is not necessary to specify `n_anisotropy`, `n_interaction_pairs` or `n_interaction_quadruplets` respectively.

```
### Pairs
interaction_pairs_file       input/pairs.txt

### Quadruplets
interaction_quadruplets_file input/quadruplets.txt
```

Units:

The external field is specified in Tesla, while anisotropy is specified in meV. Pairwise interactions are specified in meV per unique pair, while quadruplets are specified in meV per unique quadruplet.

3.4 Gaussian Hamiltonian

Note that you select the Hamiltonian you use with the `hamiltonian gaussian` input option.

This is a testing Hamiltonian consisting of the superposition of gaussian potentials. It does not contain interactions.

```
hamiltonian gaussian

### Number of Gaussians
n_gaussians 2

### Gaussians
###   a is the amplitude, s is the width, c the center
###   the directions c you enter will be normalized
###   a1 s1 c1.x c1.y c1.z
###   ...
gaussians
 1    0.2   -1   0   0
 0.5  0.4    0   0  -1
```

3.5 Method Output

For `llg` and equivalently `mc` and `gneb`, you can specify which output you want your simulations to create. They share a few common output types, for example:

```
llg_output_any     1    # Write any output at all
llg_output_initial 1    # Save before the first iteration
llg_output_final   1    # Save after the last iteration
```

Note in the following that `step` means after each `N` iterations and denotes a separate file for each step, whereas `archive` denotes that results are appended to an archive file at each step.

The energy output files are in units of meV, and can be switched to meV per spin with `<method>_output_energy_divide_by_nspins`.

LLG:

```
llg_output_energy_step              0    # Save system energy at each step
llg_output_energy_archive           1    # Archive system energy at each step
llg_output_energy_spin_resolved     0    # Also save energies for each spin
llg_output_energy_divide_by_nspins  1    # Normalize energies with number of spins

llg_output_configuration_step       1    # Save spin configuration at each step
llg_output_configuration_archive    0    # Archive spin configuration at each step
```

MC:

```
mc_output_energy_step              0
mc_output_energy_archive           1
mc_output_energy_spin_resolved     0
mc_output_energy_divide_by_nspins  1

mc_output_configuration_step    1
mc_output_configuration_archive 0
```

GNEB:

```
gneb_output_energies_step              0 # Save energies of images in chain
gneb_output_energies_interpolated      1 # Also save interpolated energies
gneb_output_energies_divide_by_nspins  1 # Normalize energies with number of spins

gneb_output_chain_step 0    # Save the whole chain at each step
```

3.6 Method Parameters

Again, the different Methods share a few common parameters. On the example of the LLG Method:

```
### Maximum wall time for single simulation
### hh:mm:ss, where 0:0:0 is infinity
llg_max_walltime        0:0:0

### Force convergence parameter
llg_force_convergence   10e-9

### Number of iterations
llg_n_iterations        2000000
### Number of iterations after which to save
llg_n_iterations_log    2000
```

LLG:

```
### Seed for Random Number Generator
llg_seed            20006

### Damping [none]
llg_damping         0.3E+0

### Time step dt [ps]
llg_dt              1.0E-3
```

(continues on next page)

(continued from previous page)

```
### Temperature [K]
llg_temperature            0
llg_temperature_gradient_direction   1 0 0
llg_temperature_gradient_inclination 0.0

### Spin transfer torque parameter proportional to injected current density
llg_stt_magnitude   0.0
### Spin current polarisation normal vector
llg_stt_polarisation_normal        1.0 0.0 0.0
```

The time step `dt` is given in picoseconds. The temperature is given in Kelvin and the temperature gradient in Kelvin/Angstrom.

MC:

```
### Seed for Random Number Generator
mc_seed                    20006

### Temperature [K]
mc_temperature      0

### Acceptance ratio
mc_acceptance_ratio 0.5
```

GNEB:

```
### Constant for the spring force
gneb_spring_constant 1.0

### Number of energy interpolations between images
gneb_n_energy_interpolations 10
```

3.7 Pinning

Note that for this feature you need to build with `SPIRIT_ENABLE_PINNING` set to `ON` in cmake.

For each lattice direction `a` `b` and `c`, you have two choices for pinning. For example to pin `n` cells in the `a` direction, you can set both `pin_na_left` and `pin_na_right` to different values or set `pin_na` to set both to the same value. To set the direction of the pinned cells, you need to give the `pinning_cell` keyword and one vector for each basis atom.

You can for example do the following to create a U-shaped pinning in x-direction:

```
# Pin left side of the sample (2 rows)
pin_na_left 2
# Pin top and bottom sides (2 rows each)
pin_nb      2
# Pin the atoms to x-direction
pinning_cell
1 0 0
```

To specify individual pinned sites (overriding the above pinning settings), insert a list into your input. For example:

```
### Specify the number of pinned sites and then the sites (in terms of translations)␣
↪and directions
### i  da db dc  Sx Sy Sz
n_pinned 3
0  0 0 0  1.0 0.0 0.0
0  1 0 0  0.0 1.0 0.0
0  0 1 0  0.0 0.0 1.0
```

You may also place it into a separate file with the keyword `pinned_from_file`, e.g.

```
### Read pinned sites from a separate file
pinned_from_file input/pinned.txt
```

The file should either contain only the pinned sites or you need to specify `n_pinned` inside the file.

3.8 Disorder and Defects

Note that for this feature you need to build with `SPIRIT_ENABLE_DEFECTS` set to `ON` in cmake.

In order to specify disorder across the lattice, you can write for example a single atom basis with 50% chance of containing one of two atom types (0 or 1):

```
# iatom  atom_type  mu_s  concentration
atom_types 1
    0        1       2.0     0.5
```

Note that you have to also specify the magnetic moment, as this is now site- and atom type dependent.

A two-atom basis where

- the first atom is type 0
- the second atom is 70% type 1 and 30% type 2

```
# iatom  atom_type  mu_s  concentration
atom_types 2
    0        0       1.0      1
    1        1       2.5     0.7
    1        2       2.3     0.3
```

The total concentration on a site should not be more than `1`. If it is less than `1`, vacancies will appear.

To specify defects, be it vacancies or impurities, you may fix atom types for sites of the whole lattice by inserting a list into your input. For example:

```
### Atom types: type index 0..n or or vacancy (type < 0)
### Specify the number of defects and then the defects in terms of translations and␣
↪type
### i  da db dc  itype
n_defects 3
0  0 0 0  -1
0  1 0 0  -1
0  0 1 0  -1
```

You may also place it into a separate file with the keyword `defects_from_file`, e.g.

```
### Read defects from a separate file
defects_from_file input/defects.txt
```

The file should either contain only the defects or you need to specify `n_defects` inside the file.

Home

CHAPTER 4

Building Spirit on Unix/OSX

Binary packages are currently not provided! Therefore, you need to build the Spirit core library or the desktop user interface yourself.

The **Spirit** framework is designed to run across different platforms and uses `CMake` for its build process, which will generate the appropriate build scripts for each platform.

System-wide installation is not actively supported. While you can call `make install` after building, you may not achieve the desired results.

4.1 Core library

Requirements

- cmake >= 3.2
- compiler with C++11 support, e.g. gcc >= 5.1

Build

CMake is used to automatically generate makefiles.

```
# enter the top-level Spirit directory
$ cd spirit

# make a build directory and enter that
$ mkdir build
$ cd build

# Generate makefiles
$ cmake ..

# Build
$ make
```

Note that you can use the `-j` option of make to run the build in parallel.

To manually specify the build type (default is 'Release'), call `cmake --build . --config Release` instead of `make`.

4.2 Desktop GUI

By default, the desktop GUI will try to build. The corresponding CMake option is `SPIRIT_UI_CXX_USE_QT`.

4.2.1 Additional requirements

- Qt >= 5.7 (including qt-charts)
- OpenGL drivers >= 3.3

Necessary OpenGL drivers *should* be available through the regular drivers for any remotely modern graphics card.

4.3 Python package

The Python package is built by default. The corresponding CMake option is `SPIRIT_BUILD_FOR_PYTHON`. The package is then located at `core/python`. You can then

- make it locatable, e.g. by adding `path/to/spirit/core/python` to your `PYTHONPATH`
- `cd core/python` and `pip install -e . --user` to install it

Alternatively, the most recent release version can be installed from the official package, e.g. `pip install spirit --user`.

4.4 OpenMP backend

The OpenMP backend can be used to speed up calculations by using a multicore CPU.

At least version 4.5 of OpenMP needs to be supported by your compiler.

Build

You need to set the corresponding CMake variable, e.g. by calling

```
cd build
cmake -DSPIRIT_USE_OPENMP=ON ..
cd ..
```

4.5 CUDA backend

The CUDA backend can be used to speed up calculations by using a GPU.

Spirit uses unified memory. At least version 8 of the CUDA toolkit is required and the GPU needs compute capability 3.0 or higher!

If the GUI is used, compute capability 6.0 or higher is required! (see the CUDA programming guide: coherency)

Note that **the GUI cannot be used on the CUDA backend on OSX**! (see the CUDA programming guide: coherency and requirements)

Note: the precision of the core will be automatically set to `float` in order to avoid the performance cost of `double` precision operations on GPUs.

Build

You need to set the corresponding `SPIRIT_USE_CUDA` CMake variable, e.g. by calling

```
cd build
cmake -DSPIRIT_USE_CUDA=ON ..
cd ..
```

You may additionally need to

- pass the `CUDA_TOOLKIT_ROOT_DIR` to cmake or edit it in the root CMakeLists.txt
- select the appropriate arch for your GPU using the `SPIRIT_CUDA_ARCH` CMake variable

4.6 Web assembly library

Using emscripten, Spirit can be built as a Web assembly library, meaning that it can be used e.g. from within JavaScript.

Build

The CMake option you need to set to `ON` is called `SPIRIT_BUILD_FOR_JS`.

You need to have emscripten available, meaning you might need to source, e.g. `source /usr/local/bin/emsdkvars.sh`.

Then to build, call

```
cd build
cmake .. -DCMAKE_TOOLCHAIN_FILE=/usr/local/emsdk/emscripten/1.38.29/cmake/Modules/
↪Platform/Emscripten.cmake
make
cd ..
```

4.7 Further build configuration options

More options than described above are available, allowing for example to deactivate building the Python library or the unit tests.

To list all available build options, call

```
cd build
cmake -LH ..
```

The build options of Spirit all start with `SPIRIT_`.

CHAPTER 5

Building Spirit on Windows

Binary packages are currently not provided! Therefore, you need to build the Spirit core library or the desktop user interface yourself.

The **Spirit** framework is designed to run across different platforms and uses `CMake` for its build process, which will generate the appropriate build scripts for each platform.

5.1 Core library

The Visual Studio Version needs to be specified and it usually makes sense to specify 64bit, as it otherwise defaults to 32bit. The version number and year may be different for you, Win64 can be appended to any of them. Execute `cmake -G` to get a listing of the available generators.

```
# enter the top-level Spirit directory
$ cd spirit

# make a build directory and enter that
$ mkdir build
$ cd build

# Generate a solution file
$ cmake -G "Visual Studio 14 2015 Win64" ..

# Either open the .sln with Visual Studio, or run
$ cmake --build . --config Release
```

You can also open the CMake GUI and configure and generate the project solution there. The solution file can be opened and built using Visual Studio, which is especially useful for debugging.

5.2 Desktop GUI

By default, the desktop GUI will try to build. The corresponding CMake option is `SPIRIT_UI_CXX_USE_QT`.

Additional requirements

- Qt >= 5.7 (including qt-charts)
- OpenGL drivers >= 3.3

Necessary OpenGL drivers *should* be available through the regular drivers for any remotely modern graphics card.

Note that in order to build with Qt as a dependency on Windows, you may need to add `path/to/qt/qtbase/bin` to your PATH variable.

5.3 Python package

The Python package is built by default. The corresponding CMake option is `SPIRIT_BUILD_FOR_PYTHON`. The package is then located at `core/python`. You can then

- make it locatable, e.g. by adding `path/to/spirit/core/python` to your `PYTHONPATH`
- `cd core/python` and `pip install -e . --user` to install it

Alternatively, the most recent release version can be installed from the official package, e.g. `pip install spirit --user`.

5.4 OpenMP backend

Using OpenMP on Windows is not officially supported. While it is possible to use it, the build process is nontrivial.

5.5 CUDA backend

The CUDA backend can be used to speed up calculations by using a GPU.

At least version 8 of the CUDA toolkit is required and the GPU needs compute capability 3.0 or higher!

Note that **the GUI cannot be used on the CUDA backend on Windows**! (see the CUDA programming guide: coherency and requirements)

Note: the precision of the core will be automatically set to `float` in order to avoid the performance cost of `double` precision operations on GPUs.

Build

You need to set the corresponding `SPIRIT_USE_CUDA` CMake variable, e.g. by calling

```
cd build
cmake -DSPIRIT_USE_CUDA=ON ..
cd ..
```

or by setting the option in the CMake GUI and re-generating.

You may additionally need to

- manually set the host compiler ("C:/Program Files (x86)/. . . /bin/cl.exe)
- manually set the CUDA Toolkit directory in the CMake GUI or pass the `CUDA_TOOLKIT_ROOT_DIR` to cmake or edit it in the root CMakeLists.txt
- select the appropriate arch for your GPU using the `SPIRIT_CUDA_ARCH` CMake variable

- add the CUDA Toolkit directory to the Windows PATH, so that the libraries will be found when the code is executed

5.6 Web assembly library

Using emscripten, Spirit can be built as a Web assembly library, meaning that it can be used e.g. from within JavaScript.

The CMake option you need to set to `ON` is called `SPIRIT_BUILD_FOR_JS`.

The build process on Windows has not been tested by us and we do not officially support it.

5.7 Further build configuration options

More options than described above are available, allowing for example to deactivate building the Python library or the unit tests.

To list all available build options, call

```
cd build
cmake -LH ..
```

The build options of Spirit all start with `SPIRIT_`.

CHAPTER 6

Docker

On Linux, the Dockerfile can be used to run the GUI of Spirit. In general, it can be used to run the core library, e.g. using Python.

The process can be

```
sudo docker build -t spirit .
```

then

```
xhost +
sudo docker run -ti --rm --device=/dev/dri:/dev/dri -e DISPLAY=$DISPLAY -v /tmp/.X11-
↪unix:/tmp/.X11-unix spirit
```

which uses the users X11 sessions (assuming that uid and gid of the host are 1000).

CHAPTER 7

Usage

7.1 Energy minimisation

Energy minimisation of a spin system can be performed using the LLG method and the velocity projection (VP) solver:

```
#include <Spirit/Simulation.h>
#include <Spirit/State.h>
#include <memory>

auto state = std::shared_ptr<State>(State_Setup("input/input.cfg"), State_Delete);
Simulation_LLG_Start(state.get(), Solver_VP);
```

or using one of the dynamics solvers, using dissipative dynamics:

```
#include <Spirit/Parameters.h>
#include <Spirit/Simulation.h>
#include <Spirit/State.h>
#include <memory>

auto state = std::shared_ptr<State>(State_Setup("input/input.cfg"), State_Delete);
Parameters_LLG_Set_Direct_Minimization(state.get(), true);
Simulation_LLG_Start(state.get(), Solver_Depondt);
```

7.2 LLG method

To perform an LLG dynamics simulation:

```
#include <Spirit/Simulation.h>
#include <Spirit/State.h>
#include <memory>
```

(continues on next page)

(continued from previous page)

```
auto state = std::shared_ptr<State>(State_Setup("input/input.cfg"), State_Delete);
Simulation_LLG_Start(state.get(), Solver_Depondt);
```

Note that the velocity projection (VP) solver is not a dynamics solver.

7.3 GNEB method

The geodesic nudged elastic band method. See also the method paper.

This method operates on a transition between two spin configurations, discretised by "images" on a "chain". The procedure follows these steps:

1. set the number of images
2. set the initial and final spin configuration
3. create an initial guess for the transition path
4. run an initial GNEB relaxation
5. determine and set the suitable images on the chain to converge on extrema
6. run a full GNEB relaxation using climbing and falling images

```
#include <Spirit/Chain.h>
#include <Spirit/Configuration.h>
#include <Spirit/Simulation.h>
#include <Spirit/State.h>
#include <Spirit/Transition.h>
#include <memory>

int NOI = 7;

auto state = std::shared_ptr<State>(State_Setup("input/input.cfg"), State_Delete);

// Copy the first image and set chain length
Chain_Image_to_Clipboard(state.get());
Chain_Set_Length(state.get(), NOI);

// First image is homogeneous with a Skyrmion in the center
Configuration_Plus_Z(state.get(), 0);
Configuration_Skyrmion(state.get(), 5.0, phase=-90.0);
Simulation_LLG_Start(state.get(), Solver_VP);
// Last image is homogeneous
Configuration_Plus_Z(state.get(), NOI-1);
Simulation_LLG_Start(state.get(), simulation.SOLVER_VP, NOI-1);

// Create initial guess for transition: homogeneous rotation
Transition_Homogeneous(state.get(), 0, noi-1);

// Initial GNEB relaxation
Simulation_GNEB_Start(state.get(), Solver_VP, 5000);
// Automatically set climbing and falling images
Chain_Set_Image_Type_Automatically(state.get());
// Full GNEB relaxation
Simulation_GNEB_Start(state.get(), Solver_VP);
```

7.4 HTST

The harmonic transition state theory. See also the method paper.

The usage of this method is not yet documented.

7.5 MMF method

The minimum mode following method. See also the method paper.

The usage of this method is not yet documented.

CHAPTER 8

Full API reference

8.1 Chain

```
#include "Spirit/Chain.h"
```

A chain of spin systems can be used for example for

- calculating minimum energy paths using the GNEB method
- running multiple (e.g. LLG) calculations in parallel

8.1.1 Chain_Get_NOI

```
int Chain_Get_NOI(State * state, int idx_chain=-1)
```

Returns the number of images in the chain.

8.1.2 Change the active image

Chain_next_Image

```
bool Chain_next_Image(State * state, int idx_chain=-1)
```

Move to next image in the chain (change active_image).

Returns: success of the function

Chain_prev_Image

```
bool Chain_prev_Image(State * state, int idx_chain=-1)
```

Move to previous image in the chain (change active_image)

Returns: success of the function

Chain_Jump_To_Image

```
bool Chain_Jump_To_Image(State * state, int idx_image=-1, int idx_chain=-1)
```

Move to a specific image (change active_image)

Returns: success of the function

8.1.3 Insert/replace/delete images

Chain_Set_Length

```
void Chain_Set_Length(State * state, int n_images, int idx_chain=-1)
```

Set the number of images in the chain.

If it is currently less, a corresponding number of images will be appended. If it is currently more, a corresponding number of images will be deleted from the end.

Note: you need to have an image in the *clipboard*.

Chain_Image_to_Clipboard

```
void Chain_Image_to_Clipboard(State * state, int idx_image=-1, int idx_chain=-1)
```

Copy an image from the chain (default: active image).

You can later insert it anywhere in the chain.

Chain_Replace_Image

```
void Chain_Replace_Image(State * state, int idx_image=-1, int idx_chain=-1)
```

Replaces the specified image (default: active image).

Note: you need to have an image in the *clipboard*.

Chain_Insert_Image_Before

```
void Chain_Insert_Image_Before(State * state, int idx_image=-1, int idx_chain=-1)
```

Inserts an image in front of the specified image index (default: active image).

Note: you need to have an image in the *clipboard*.

Chain_Insert_Image_After

```
void Chain_Insert_Image_After(State * state, int idx_image=-1, int idx_chain=-1)
```

Inserts an image behind the specified image index (default: active image).

Note: you need to have an image in the *clipboard*.

Chain_Push_Back

```
void Chain_Push_Back(State * state, int idx_chain=-1)
```

Appends an image to the chain.

Note: you need to have an image in the *clipboard*.

Chain_Delete_Image

```
bool Chain_Delete_Image(State * state, int idx_image=-1, int idx_chain=-1)
```

Removes an image from the chain (default: active image).

Does nothing if the chain contains only one image.

Chain_Pop_Back

```
bool Chain_Pop_Back(State * state, int idx_chain=-1)
```

Removes the last image of the chain.

Does nothing if the chain contains only one image.

Returns: success of the operation

8.1.4 Calculate data

Chain_Get_Rx

```
void Chain_Get_Rx(State * state, float * Rx, int idx_chain=-1)
```

Fills an array with the reaction coordinate values of the images in the chain.

Chain_Get_Rx_Interpolated

```
void Chain_Get_Rx_Interpolated(State * state, float * Rx_interpolated, int idx_chain=-
↪1)
```

Fills an array with the interpolated reaction coordinate values along the chain.

Chain_Get_Energy

```
void Chain_Get_Energy(State * state, float * energy, int idx_chain=-1)
```

Fills an array with the energy values of the images in the chain.

Chain_Get_Energy_Interpolated

```
void Chain_Get_Energy_Interpolated(State * state, float * E_interpolated, int idx_
↪chain=-1)
```

Fills an array with the interpolated energy values along the chain.

Chain_Update_Data

```
void Chain_Update_Data(State * state, int idx_chain=-1)
```

Update Data, such as energy or reaction coordinate.

This is primarily used for the plotting in the GUI, but needs to be called e.g. before calling the automatic setting of GNEB image types.

Chain_Setup_Data

```
void Chain_Setup_Data(State * state, int idx_chain=-1)
```

You probably won't need this.

8.2 Configurations

```
#include "Spirit/Configurations.h"
```

Setting spin configurations for individual spin systems.

The position of the relative center and a set of conditions can be defined.

8.2.1 Clipboard

Configuration_To_Clipboard

```
void Configuration_To_Clipboard(State *state, int idx_image=-1, int idx_chain=-1)
```

Copies the current spin configuration to the clipboard

Configuration_From_Clipboard

```
void Configuration_From_Clipboard(State *state, const float position[3]=defaultPos,␣
↪const float r_cut_rectangular[3]=defaultRect, float r_cut_cylindrical=-1, float r_
↪cut_spherical=-1, bool inverted=false, int idx_image=-1, int idx_chain=-1)
```

(continues on next page)

(continued from previous page)

Pastes the clipboard spin configuration

Configuration_From_Clipboard_Shift

```
bool Configuration_From_Clipboard_Shift(State *state, const float shift[3], const␣
↪float position[3]=defaultPos, const float r_cut_rectangular[3]=defaultRect, float r_
↪cut_cylindrical=-1, float r_cut_spherical=-1, bool inverted = false, int idx_image=-
↪1, int idx_chain=-1)
```

Pastes the clipboard spin configuration

8.2.2 Nonlocalised

Configuration_Domain

```
void Configuration_Domain(State *state, const float direction[3], const float␣
↪position[3]=defaultPos, const float r_cut_rectangular[3]=defaultRect, float r_cut_
↪cylindrical=-1, float r_cut_spherical=-1, bool inverted=false, int idx_image=-1,␣
↪int idx_chain=-1)
```

Creates a homogeneous domain

Configuration_PlusZ

```
void Configuration_PlusZ(State *state, const float position[3]=defaultPos, const␣
↪float r_cut_rectangular[3]=defaultRect, float r_cut_cylindrical=-1, float r_cut_
↪spherical=-1, bool inverted=false, int idx_image=-1, int idx_chain=-1)
```

Points all spins in +z direction

Configuration_MinusZ

```
void Configuration_MinusZ(State *state, const float position[3]=defaultPos, const␣
↪float r_cut_rectangular[3]=defaultRect, float r_cut_cylindrical=-1, float r_cut_
↪spherical=-1, bool inverted=false, int idx_image=-1, int idx_chain=-1)
```

Points all spins in -z direction

Configuration_Random

```
void Configuration_Random(State *state, const float position[3]=defaultPos, const␣
↪float r_cut_rectangular[3]=defaultRect, float r_cut_cylindrical=-1, float r_cut_
↪spherical=-1, bool inverted=false, bool external=false, int idx_image=-1, int idx_
↪chain=-1)
```

Points all spins in random directions

Configuration_SpinSpiral

```
void Configuration_SpinSpiral(State *state, const char * direction_type, float q[3],␣
↪float axis[3], float theta, const float position[3]=defaultPos, const float r_cut_
↪rectangular[3]=defaultRect, float r_cut_cylindrical=-1, float r_cut_spherical=-1,␣
↪bool inverted=false, int idx_image=-1, int idx_chain=-1)
```

Spin spiral

Configuration_SpinSpiral_2q

```
void Configuration_SpinSpiral_2q(State *state, const char * direction_type, float␣
↪q1[3], float q2[3], float axis[3], float theta, const float position[3]=defaultPos,␣
↪const float r_cut_rectangular[3]=defaultRect, float r_cut_cylindrical=-1, float r_
↪cut_spherical=-1, bool inverted=false, int idx_image=-1, int idx_chain=-1)
```

2q spin spiral

8.2.3 Perturbations

Configuration_Add_Noise_Temperature

```
void Configuration_Add_Noise_Temperature(State *state, float temperature, const float␣
↪position[3]=defaultPos, const float r_cut_rectangular[3]=defaultRect, float r_cut_
↪cylindrical=-1, float r_cut_spherical=-1, bool inverted=false, int idx_image=-1,␣
↪int idx_chain=-1)
```

Adds some random noise scaled by temperature

Configuration_Displace_Eigenmode

```
void Configuration_Displace_Eigenmode(State *state, int idx_mode, int idx_image=-1,␣
↪int idx_chain=-1)
```

Calculate the eigenmodes of the system (Image)

8.2.4 Localised

Configuration_Skyrmion

```
void Configuration_Skyrmion(State *state, float r, float order, float phase, bool␣
↪upDown, bool achiral, bool rl, const float position[3]=defaultPos, const float r_
↪cut_rectangular[3]=defaultRect, float r_cut_cylindrical=-1, float r_cut_spherical=-
↪1, bool inverted=false, int idx_image=-1, int idx_chain=-1)
```

Create a skyrmion configuration

Configuration_Hopfion

```
void Configuration_Hopfion(State *state, float r, int order=1, const float␣
↪position[3]=defaultPos, const float r_cut_rectangular[3]=defaultRect, float r_cut_
↪cylindrical=-1, float r_cut_spherical=-1, bool inverted=false,  int idx_image=-1,␣
↪int idx_chain=-1)
```

Create a toroidal Hopfion

8.2.5 Pinning and atom types

This API can also be used to change the `pinned` state and the `atom type` of atoms in a spacial region, instead of using translation indices.

Configuration_Set_Pinned

```
void Configuration_Set_Pinned(State *state, bool pinned, const float␣
↪position[3]=defaultPos, const float r_cut_rectangular[3]=defaultRect, float r_cut_
↪cylindrical=-1, float r_cut_spherical=-1, bool inverted=false, int idx_image=-1,␣
↪int idx_chain=-1)
```

Pinning

Configuration_Set_Atom_Type

```
void Configuration_Set_Atom_Type(State *state, int type, const float␣
↪position[3]=defaultPos, const float r_cut_rectangular[3]=defaultRect, float r_cut_
↪cylindrical=-1, float r_cut_spherical=-1, bool inverted=false, int idx_image=-1,␣
↪int idx_chain=-1)
```

Atom types

8.3 Constants

```
#include "Spirit/Constants.h"
```

Physical constants in units compatible to what is used in Spirit.

8.3.1 Constants_mu_B

```
scalar Constants_mu_B()
```

The Bohr Magneton [meV / T]

8.3.2 Constants_mu_0

```
scalar Constants_mu_0()
```

The vacuum permeability [T^2 m^3 / meV]

8.3.3 Constants_k_B

```
scalar Constants_k_B()
```

The Boltzmann constant [meV / K]

8.3.4 Constants_hbar

```
scalar Constants_hbar()
```

Planck constant [meV*ps / rad]

8.3.5 Constants_mRy

```
scalar Constants_mRy()
```

Millirydberg [mRy / meV]

8.3.6 Constants_gamma

```
scalar Constants_gamma()
```

Gyromagnetic ratio of electron [rad / (s*T)]

8.3.7 Constants_g_e

```
scalar Constants_g_e()
```

Electron g-factor [unitless]

8.3.8 Constants_Pi

```
scalar Constants_Pi()
```

Pi [rad]

8.4 Geometry

```
#include "Spirit/Geometry.h"
```

This set of functions can be used to get information about the geometric setup of the system and to change it.

Note that it is not fully safe to change the geometry during a calculation, as this has not been so thoroughly tested.

8.4.1 Definition of Bravais lattice types

```
typedef enum
{
    Bravais_Lattice_Irregular   = 0,
    Bravais_Lattice_Rectilinear = 1,
    Bravais_Lattice_SC          = 2,
    Bravais_Lattice_Hex2D       = 3,
    Bravais_Lattice_Hex2D_60    = 4,
    Bravais_Lattice_Hex2D_120   = 5,
    Bravais_Lattice_HCP         = 6,
    Bravais_Lattice_BCC         = 7,
    Bravais_Lattice_FCC         = 8
} Bravais_Lattice_Type;
```

8.4.2 Setters

Geometry_Set_Bravais_Lattice_Type

```
void Geometry_Set_Bravais_Lattice_Type(State *state, Bravais_Lattice_Type lattice_
↪type)
```

Set the type of Bravais lattice. Can be e.g. “sc” or “bcc”.

Geometry_Set_N_Cells

```
void Geometry_Set_N_Cells(State * state, int n_cells[3])
```

Set the number of basis cells in the three translation directions.

Geometry_Set_Cell_Atoms

```
void Geometry_Set_Cell_Atoms(State *state, int n_atoms, float ** atoms)
```

Set the number and positions of atoms in a basis cell. Positions are in units of the bravais vectors (scaled by the lattice constant).

Geometry_Set_mu_s

```
void Geometry_Set_mu_s(State *state, float mu_s, int idx_image=-1, int idx_chain=-1)
```

Set the magnetic moments of basis cell atoms.

Geometry_Set_Cell_Atom_Types

```
void Geometry_Set_Cell_Atom_Types(State *state, int n_atoms, int * atom_types)
```

Set the types of the atoms in a basis cell.

Geometry_Set_Bravais_Vectors

```
void Geometry_Set_Bravais_Vectors(State *state, float ta[3], float tb[3], float tc[3])
```

Manually set the bravais vectors.

Geometry_Set_Lattice_Constant

```
void Geometry_Set_Lattice_Constant(State *state, float lattice_constant)
```

Set the overall lattice scaling constant.

8.4.3 Getters

Geometry_Get_NOS

```
int Geometry_Get_NOS(State * state)
```

Returns: the number of spins.

Geometry_Get_Positions

```
scalar * Geometry_Get_Positions(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns: pointer to positions of spins (array of length 3*NOS).

Geometry_Get_Atom_Types

```
int * Geometry_Get_Atom_Types(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns: pointer to atom types of lattice sites.

Geometry_Get_Bounds

```
void Geometry_Get_Bounds(State *state, float min[3], float max[3], int idx_image=-1,␣
↪int idx_chain=-1)
```

Get Bounds as array (x,y,z).

Geometry_Get_Center

```
void Geometry_Get_Center(State *state, float center[3], int idx_image=-1, int idx_
↪chain=-1)
```

Get Center as array (x,y,z).

Geometry_Get_Bravais_Lattice_Type

```
Bravais_Lattice_Type Geometry_Get_Bravais_Lattice_Type(State *state, int idx_image=-1,
↪ int idx_chain=-1)
```

Get bravais lattice type (see the `enum` defined above).

Geometry_Get_Bravais_Vectors

```
void Geometry_Get_Bravais_Vectors(State *state, float a[3], float b[3], float c[3],␣
↪int idx_image=-1, int idx_chain=-1)
```

Get bravais vectors ta, tb, tc.

Geometry_Get_Dimensionality

```
int Geometry_Get_Dimensionality(State * state, int idx_image=-1, int idx_chain=-1)
```

Retrieve dimensionality of the system (0, 1, 2, 3).

Geometry_Get_mu_s

```
void Geometry_Get_mu_s(State *state, float * mu_s, int idx_image=-1, int idx_chain=-1)
```

Get the magnetic moments of basis cell atoms.

Geometry_Get_N_Cells

```
void Geometry_Get_N_Cells(State *state, int n_cells[3], int idx_image=-1, int idx_
↪chain=-1)
```

Get number of basis cells in the three translation directions.

The basis cell

Geometry_Get_Cell_Bounds

```
void Geometry_Get_Cell_Bounds(State *state, float min[3], float max[3], int idx_
↪image=-1, int idx_chain=-1)
```

Get Cell Bounds as array (x,y,z).

Geometry_Get_N_Cell_Atoms

```
int Geometry_Get_N_Cell_Atoms(State *state, int idx_image=-1, int idx_chain=-1)
```

Get number of atoms in a basis cell.

Geometry_Get_Cell_Atoms

```
int Geometry_Get_Cell_Atoms(State *state, scalar ** atoms, int idx_image=-1, int idx_
↪chain=-1)
```

Get basis cell atom positions in units of the bravais vectors (scaled by the lattice constant).

Triangulation and tetrahedra

Geometry_Get_Triangulation

```
int Geometry_Get_Triangulation(State * state, const int **indices_ptr, int n_cell_
↪step=1, int idx_image=-1, int idx_chain=-1)
```

Get the 2D Delaunay triangulation. Returns the number of triangles and sets *indices_ptr to point to a list of index 3-tuples.

Returns: the number of triangles in the triangulation

Geometry_Get_Tetrahedra

```
int Geometry_Get_Tetrahedra(State * state, const int **indices_ptr, int n_cell_step=1,
↪ int idx_image=-1, int idx_chain=-1)
```

Get the 3D Delaunay triangulation. Returns the number of tetrahedrons and sets *indices_ptr to point to a list of index 4-tuples.

Returns: the number of tetrahedra

8.5 Hamiltonian

```
#include "Spirit/Hamiltonian.h"
```

This currently only provides an interface to the Heisenberg Hamiltonian.

8.5.1 DMI chirality

This means that

- Bloch chirality corresponds to DM vectors along bonds
- Neel chirality corresponds to DM vectors orthogonal to bonds

Neel chirality should therefore only be used in 2D systems.

SPIRIT_CHIRALITY_BLOCH

```
SPIRIT_CHIRALITY_BLOCH 1
```

Bloch chirality

SPIRIT_CHIRALITY_NEEL

```
SPIRIT_CHIRALITY_NEEL  2
```

Neel chirality

SPIRIT_CHIRALITY_BLOCH_INVERSE

```
SPIRIT_CHIRALITY_BLOCH_INVERSE -1
```

Bloch chirality, inverted DM vectors

SPIRIT_CHIRALITY_NEEL_INVERSE

```
SPIRIT_CHIRALITY_NEEL_INVERSE  -2
```

Neel chirality, inverted DM vectors

8.5.2 Dipole-Dipole method

SPIRIT_DDI_METHOD_NONE

```
SPIRIT_DDI_METHOD_NONE   0
```

Do not use dipolar interactions

SPIRIT_DDI_METHOD_FFT

```
SPIRIT_DDI_METHOD_FFT    1
```

Use fast Fourier transform (FFT) convolutions

SPIRIT_DDI_METHOD_FMM

```
SPIRIT_DDI_METHOD_FMM    2
```

Use the fast multipole method (FMM)

SPIRIT_DDI_METHOD_CUTOFF

```
SPIRIT_DDI_METHOD_CUTOFF 3
```

Use a direct summation with a cutoff radius

8.5.3 Setters

Hamiltonian_Set_Boundary_Conditions

```
void Hamiltonian_Set_Boundary_Conditions(State *state, const bool* periodical, int␣
↪idx_image=-1, int idx_chain=-1)
```

Set the boundary conditions along the translation directions [a, b, c]

Hamiltonian_Set_Field

```
void Hamiltonian_Set_Field(State *state, float magnitude, const float* normal, int␣
↪idx_image=-1, int idx_chain=-1)
```

Set the (homogeneous) external magnetic field [T]

Hamiltonian_Set_Anisotropy

```
void Hamiltonian_Set_Anisotropy(State *state, float magnitude, const float* normal,␣
↪int idx_image=-1, int idx_chain=-1)
```

Set a global uniaxial anisotropy [meV]

Hamiltonian_Set_Exchange

```
void Hamiltonian_Set_Exchange(State *state, int n_shells, const float* jij, int idx_
↪image=-1, int idx_chain=-1)
```

Set the exchange interaction in terms of neighbour shells [meV]

Hamiltonian_Set_DMI

```
void Hamiltonian_Set_DMI(State *state, int n_shells, const float * dij, int␣
↪chirality=SPIRIT_CHIRALITY_BLOCH, int idx_image=-1, int idx_chain=-1)
```

Set the Dzyaloshinskii-Moriya interaction in terms of neighbour shells [meV]

Hamiltonian_Set_DDI

```
void Hamiltonian_Set_DDI(State *state, int ddi_method, int n_periodic_images[3],␣
↪float cutoff_radius=0, int idx_image=-1, int idx_chain=-1)
```

Configure the dipole-dipole interaction

- `ddi_method`: see integers defined above
- `n_periodic_images`: how many repetition of the spin configuration to append along the translation directions [a, b, c], if periodical boundary conditions are used
- `cutoff_radius`: the distance at which to stop the direct summation, if used

8.5.4 Getters

Hamiltonian_Get_Name

```
const char * Hamiltonian_Get_Name(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns a string containing the name of the Hamiltonian in use

Hamiltonian_Get_Boundary_Conditions

```
void Hamiltonian_Get_Boundary_Conditions(State *state, bool * periodical, int idx_
↪image=-1, int idx_chain=-1)
```

Retrieves the boundary conditions

Hamiltonian_Get_Field

```
void Hamiltonian_Get_Field(State *state, float * magnitude, float * normal, int idx_
↪image=-1, int idx_chain=-1)
```

Retrieves the external magnetic field [T]

Hamiltonian_Get_Anisotropy

```
void Hamiltonian_Get_Anisotropy(State *state, float * magnitude, float * normal, int␣
↪idx_image=-1, int idx_chain=-1)
```

Retrieves the uniaxial anisotropy [meV]

Hamiltonian_Get_Exchange_Shells

```
void Hamiltonian_Get_Exchange_Shells(State *state, int * n_shells, float * jij, int␣
↪idx_image=-1, int idx_chain=-1)
```

Retrieves the exchange interaction in terms of neighbour shells.

Note: if the interactions were specified as pairs, this function will retrieve `n_shells=0`.

Hamiltonian_Get_Exchange_N_Pairs

```
int  Hamiltonian_Get_Exchange_N_Pairs(State *state, int idx_image=-1, int idx_chain=-
↪1)
```

Returns the number of exchange interaction pairs

Hamiltonian_Get_DMI_Shells

```
void Hamiltonian_Get_DMI_Shells(State *state, int * n_shells, float * dij, int *␣
↪chirality, int idx_image=-1, int idx_chain=-1)
```

Retrieves the Dzyaloshinskii-Moriya interaction in terms of neighbour shells.

Note: if the interactions were specified as pairs, this function will retrieve `n_shells=0`.

Hamiltonian_Get_DMI_N_Pairs

```
int  Hamiltonian_Get_DMI_N_Pairs(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the number of Dzyaloshinskii-Moriya interaction pairs

Hamiltonian_Get_DDI

```
void Hamiltonian_Get_DDI(State *state, int * ddi_method, int n_periodic_images[3],␣
↪float * cutoff_radius, int idx_image=-1, int idx_chain=-1)
```

Retrieves the dipole-dipole interaction configuration.

- `ddi_method`: see integers defined above
- `n_periodic_images`: how many repetition of the spin configuration to append along the translation directions [a, b, c], if periodical boundary conditions are used
- `cutoff_radius`: the distance at which to stop the direct summation, if used

8.6 HTST

```
#include "Spirit/HTST.h"
```

Harmonic transition state theory.

Note that `HTST_Calculate` needs to be called before using any of the getter functions.

8.6.1 HTST_Calculate

```
float HTST_Calculate(State * state, int idx_image_minimum, int idx_image_sp, int n_
↪eigenmodes_keep=0, int idx_chain=-1)
```

Calculates the HTST transition rate prefactor for the transition from a minimum over saddle point.

- `idx_image_minimum`: index of the local minimum in the chain
- `idx_image_sp`: index of the transition saddle point in the chain
- `n_eigenmodes_keep`: the number of energy eigenmodes to keep in memory (0 = none, negative value = all)

Note that the method assumes you gave it correct images, where the gradient is zero and which correspond to a minimum and a saddle point respectively.

8.6.2 HTST_Get_Info

```
void HTST_Get_Info( State * state, float * temperature_exponent, float * me, float *␣
↪Omega_0, float * s, float * volume_min, float * volume_sp, float * prefactor_
↪dynamical, float * prefactor, int idx_chain=-1 )
```

Retrieves a set of information from HTST:

- temperature_exponent: the exponent of the temperature-dependent prefactor
- me: sqrt(2pi k_B)^(N_0^M - N_0^SP)
- Omega_0: sqrt(prod(lambda^M) / prod(lambda^SP))
- s: sqrt(prod(a^2 / lambda^SP))
- volume_min: zero mode volume at the minimum
- volume_sp: zero mode volume at the saddle point
- prefactor_dynamical: the dynamical part of the rate prefactor
- prefactor: the total rate prefactor for the transition

8.6.3 HTST_Get_Eigenvalues_Min

```
void HTST_Get_Eigenvalues_Min( State * state, float * eigenvalues_min, int idx_chain=-
↪1 )
```

Fetches HTST information eigenvalues at the min (array of length 2*NOS)

8.6.4 HTST_Get_Eigenvectors_Min

```
void HTST_Get_Eigenvectors_Min( State * state, float * eigenvectors_min, int idx_
↪chain=-1 )
```

Fetches HTST eigenvectors at the minimum (array of length 2*NOS*NOS)

8.6.5 HTST_Get_Eigenvalues_SP

```
void HTST_Get_Eigenvalues_SP( State * state, float * eigenvalues_sp, int idx_chain=-1␣
↪)
```

Fetches HTST eigenvalues at the saddle point (array of length 2*NOS)

8.6.6 HTST_Get_Eigenvectors_SP

```
void HTST_Get_Eigenvectors_SP( State * state, float * eigenvectors_sp, int idx_chain=-
↪1 )
```

Fetches HTST eigenvectors at the saddle point (array of length 2*NOS*NOS)

8.6.7 HTST_Get_Velocities

```
void HTST_Get_Velocities( State * state, float * velocities, int idx_chain=-1 )
```

Fetches HTST information:

- velocities along the unstable mode (array of length 2*NOS)

8.7 I/O

```
#include "Spirit/IO.h"
```

TODO: give bool returns for these functions to indicate success?

8.7.1 Definition of file formats for vectorfields

Spirit uses the OOMMF vector field file format with some minor variations.

IO_Fileformat_OVF_bin

```
IO_Fileformat_OVF_bin   0
```

OVF binary format, using the precision of Spirit

IO_Fileformat_OVF_bin4

```
IO_Fileformat_OVF_bin4  1
```

OVF binary format, using single precision

IO_Fileformat_OVF_bin8

```
IO_Fileformat_OVF_bin8  2
```

OVF binary format, using double precision

IO_Fileformat_OVF_text

```
IO_Fileformat_OVF_text  3
```

OVF text format

IO_Fileformat_OVF_csv

```
IO_Fileformat_OVF_csv   4
```

OVF text format with comma-separated columns

8.7.2 Other

IO_System_From_Config

```
int IO_System_From_Config( State * state, const char * file, int idx_image=-1, int␣
↪idx_chain=-1 )
```

Initialise a spin system using a config file.

IO_Positions_Write

```
void IO_Positions_Write( State * state, const char *file, int format=IO_Fileformat_
↪OVF_bin, const char *comment = "-", int idx_image=-1, int idx_chain=-1 )
```

Write the spin positions as a vector field to file.

8.7.3 Spin configurations

IO_N_Images_In_File

```
int IO_N_Images_In_File( State * state, const char *file, int idx_image=-1, int idx_
↪chain=-1 )
```

Returns the number of images (i.e. OVF segments) in a given file.

IO_Image_Read

```
void IO_Image_Read( State *state, const char *file, int idx_image_infile=0, int idx_
↪image_inchain=-1, int idx_chain=-1 )
```

Reads a spin configuration from a file.

IO_Image_Write

```
void IO_Image_Write( State *state, const char *file, int format=IO_Fileformat_OVF_bin,
↪ const char *comment = "-", int idx_image=-1, int idx_chain=-1 )
```

Writes a spin configuration to file.

IO_Image_Append

```
void IO_Image_Append( State *state, const char *file, int format=IO_Fileformat_OVF_
↪bin, const char *comment = "-", int idx_image=-1, int idx_chain=-1 )
```

Appends a spin configuration to a file.

8.7.4 Chains

IO_Chain_Read

```
void IO_Chain_Read( State *state, const char *file, int start_image_infile=0, int end_
↪image_infile=-1, int insert_idx=0, int idx_chain=-1 )
```

Read a chain of spin configurations from a file.

If the current chain is not long enough to fit the file contents, systems will be appended accordingly.

IO_Chain_Write

```
void IO_Chain_Write( State *state, const char *file, int format=IO_Fileformat_OVF_
↪text, const char* comment = "-", int idx_chain=-1 )
```

Write the current chain of spin configurations to file

IO_Chain_Append

```
void IO_Chain_Append( State *state, const char *file, int format=IO_Fileformat_OVF_
↪text, const char* comment = "-", int idx_chain=-1 )
```

Append the current chain of spin configurations to a file

8.7.5 Neighbours

IO_Image_Write_Neighbours_Exchange

```
void IO_Image_Write_Neighbours_Exchange( State * state, const char * file, int idx_
↪image=-1, int idx_chain=-1 )
```

Save the exchange interactions

IO_Image_Write_Neighbours_DMI

```
void IO_Image_Write_Neighbours_DMI( State * state, const char * file, int idx_image=-
↪1, int idx_chain=-1 )
```

Save the DM interactions

8.7.6 Energies

IO_Image_Write_Energy_per_Spin

```
void IO_Image_Write_Energy_per_Spin( State *state, const char *file, int format, int␣
↪idx_image=-1, int idx_chain = -1 )
```

Save the spin-resolved energy contributions of a spin system

IO_Image_Write_Energy

```
void IO_Image_Write_Energy( State *state, const char *file, int idx_image=-1, int idx_
↪chain=-1 )
```

Save the Energy contributions of a spin system

IO_Chain_Write_Energies

```
void IO_Chain_Write_Energies( State *state, const char *file, int idx_chain = -1 )
```

Save the Energy contributions of a chain of spin systems

IO_Chain_Write_Energies_Interpolated

```
void IO_Chain_Write_Energies_Interpolated( State *state, const char *file, int idx_
↪chain = -1 )
```

Save the interpolated energies of a chain of spin systems

8.7.7 Eigenmodes

IO_Eigenmodes_Read

```
void IO_Eigenmodes_Read( State *state, const char *file, int idx_image_inchain=-1,␣
↪int idx_chain=-1 )
```

Read eigenmodes of a spin system from a file.

The file is expected to contain a chain of vector fields, which will each be interpreted as one eigenmode of the system.

IO_Eigenmodes_Write

```
void IO_Eigenmodes_Write( State *state, const char *file, int format=IO_Fileformat_
↪OVF_text, const char *comment = "-", int idx_image=-1, int idx_chain=-1 )
```

Write eigenmodes of a spin system to a file.

The file will contain a chain of vector fields corresponding to the eigenmodes. The eigenvalues of the respective modes will be written in a commment.

8.8 Logging

```
#include "Spirit/Log.h"
```

8.8.1 Definition of log levels and senders

Spirit_Log_Level

```
typedef enum { Log_Level_All       = 0, Log_Level_Severe    = 1, Log_Level_Error
↪= 2, Log_Level_Warning   = 3, Log_Level_Parameter = 4, Log_Level_Info      = 5, Log_
↪Level_Debug     = 6 } Spirit_Log_Level
```

Levels

Spirit_Log_Sender

```
typedef enum { Log_Sender_All  = 0, Log_Sender_IO   = 1, Log_Sender_GNEB = 2, Log_
↪Sender_LLG  = 3, Log_Sender_MC   = 4, Log_Sender_MMF  = 5, Log_Sender_EMA  = 6, Log_
↪Sender_API  = 7, Log_Sender_UI   = 8, Log_Sender_HTST = 9 } Spirit_Log_Sender
```

Senders

8.8.2 Logging functions

Log_Send

```
void Log_Send(State *state, Spirit_Log_Level level, Spirit_Log_Sender sender, const
↪char * message, int idx_image=-1, int idx_chain=-1)
```

Send a Log message

Log_Append

```
void Log_Append(State *state)
```

Append the Log to it's file

Log_Dump

```
void Log_Dump(State *state)
```

Dump the Log into it's file

Log_Get_N_Entries

```
int Log_Get_N_Entries(State *state)
```

Get the number of Log entries

Log_Get_N_Errors

```
int Log_Get_N_Errors(State *state)
```

Get the number of errors in the Log

Log_Get_N_Warnings

```
int Log_Get_N_Warnings(State *state)
```

Get the number of warnings in the Log

8.8.3 Set Log parameters

Log_Set_Output_File_Tag

```
void Log_Set_Output_File_Tag(State *state, const char * tag)
```

The tag in front of the log file

Log_Set_Output_Folder

```
void Log_Set_Output_Folder(State *state, const char * folder)
```

The output folder for the log file

Log_Set_Output_To_Console

```
void Log_Set_Output_To_Console(State *state, bool output, int level)
```

Whether to write log messages to the console and corresponding level

Log_Set_Output_To_File

```
void Log_Set_Output_To_File(State *state, bool output, int level)
```

Whether to write log messages to the log file and corresponding level

8.8.4 Get Log parameters

Log_Get_Output_File_Tag

```
const char * Log_Get_Output_File_Tag(State *state)
```

Returns the tag in front of the log file

Log_Get_Output_Folder

```
const char * Log_Get_Output_Folder(State *state)
```

Returns the output folder for the log file

Log_Get_Output_To_Console

```
bool Log_Get_Output_To_Console(State *state)
```

Returns whether to write log messages to the console

Log_Get_Output_Console_Level

```
int Log_Get_Output_Console_Level(State *state)
```

Returns the console logging level

Log_Get_Output_To_File

```
bool Log_Get_Output_To_File(State *state)
```

Returns whether to write log messages to the log file

Log_Get_Output_File_Level

```
int Log_Get_Output_File_Level(State *state)
```

Returns the file logging level

8.9 MC Parameters

```
#include "Spirit/Parameters_MC.h"
```

8.9.1 Set

Parameters_MC_Set_Output_Tag

```
void Parameters_MC_Set_Output_Tag(State *state, const char * tag, int idx_image=-1,
↪int idx_chain=-1)
```

Set the tag placed in front of output file names.

If the tag is “”, it will be the date-time of the creation of the state.

Parameters_MC_Set_Output_Folder

```
void Parameters_MC_Set_Output_Folder(State *state, const char * folder, int idx_
↪image=-1, int idx_chain=-1)
```

Set the folder, where output files are placed.

Parameters_MC_Set_Output_General

```
void Parameters_MC_Set_Output_General(State *state, bool any, bool initial, bool␣
↪final, int idx_image=-1, int idx_chain=-1)
```

Set whether to write any output files at all.

Parameters_MC_Set_Output_Energy

```
void Parameters_MC_Set_Output_Energy(State *state, bool energy_step, bool energy_
↪archive, bool energy_spin_resolved, bool energy_divide_by_nos, bool energy_add_
↪readability_lines, int idx_image=-1, int idx_chain=-1)
```

Set whether to write energy output files.

- `step`: whether to write a new file after each set of iterations
- `archive`: whether to append to an archive file after each set of iterations
- `spin_resolved`: whether to write a file containing the energy of each spin
- `divide_by_nos`: whether to divide energies by the number of spins
- `add_readability_lines`: whether to separate columns by lines

Parameters_MC_Set_Output_Configuration

```
void Parameters_MC_Set_Output_Configuration(State *state, bool configuration_step,␣
↪bool configuration_archive, int configuration_filetype=IO_Fileformat_OVF_text, int␣
↪idx_image=-1, int idx_chain=-1)
```

Set whether to write spin configuration output files.

- `step`: whether to write a new file after each set of iterations
- `archive`: whether to append to an archive file after each set of iterations
- `filetype`: the format in which the data is written

Parameters_MC_Set_N_Iterations

```
void Parameters_MC_Set_N_Iterations(State *state, int n_iterations, int n_iterations_
↪log, int idx_image=-1, int idx_chain=-1)
```

Set the number of iterations and how often to log and write output.

- `n_iterations`: the maximum number of iterations
- `n_iterations_log`: the number of iterations after which status is logged and output written

8.9.2 Set Parameters

Parameters_MC_Set_Temperature

```
void Parameters_MC_Set_Temperature(State *state, float T, int idx_image=-1, int idx_
↪chain=-1)
```

Set the (homogeneous) base temperature [K].

Parameters_MC_Set_Metropolis_Cone

```
void Parameters_MC_Set_Metropolis_Cone(State *state, bool cone, float cone_angle,␣
↪bool adaptive_cone, float target_acceptance_ratio, int idx_image=-1, int idx_chain=-
↪1)
```

Configure the Metropolis parameters.

- use_cone: whether to displace the spins within a cone (otherwise: on the entire unit sphere)
- cone_angle: the opening angle within which the spin is placed
- use_adaptive_cone: automatically adapt the cone angle to achieve the set acceptance ratio
- target_acceptance_ratio: target acceptance ratio for the adaptive cone algorithm

Parameters_MC_Set_Random_Sample

```
void Parameters_MC_Set_Random_Sample(State *state, bool random_sample, int idx_image=-
↪1, int idx_chain=-1)
```

Set whether spins should be sampled randomly or in sequence.

8.9.3 Get Output

Parameters_MC_Get_Output_Tag

```
const char * Parameters_MC_Get_Output_Tag(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the output file tag.

Parameters_MC_Get_Output_Folder

```
const char * Parameters_MC_Get_Output_Folder(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the output folder.

Parameters_MC_Get_Output_General

```
void Parameters_MC_Get_Output_General(State *state, bool * any, bool * initial, bool␣
↪* final, int idx_image=-1, int idx_chain=-1)
```

Retrieves whether to write any output at all.

Parameters_MC_Get_Output_Energy

```
void Parameters_MC_Get_Output_Energy(State *state, bool * energy_step, bool * energy_
↪archive, bool * energy_spin_resolved, bool * energy_divide_by_nos, bool * energy_
↪add_readability_lines, int idx_image=-1, int idx_chain=-1)
```

Retrieves the energy output settings.

Parameters_MC_Get_Output_Configuration

```
void Parameters_MC_Get_Output_Configuration(State *state, bool * configuration_step,␣
↪bool * configuration_archive, int * configuration_filetype, int idx_image=-1, int␣
↪idx_chain=-1)
```

Retrieves the spin configuration output settings.

Parameters_MC_Get_N_Iterations

```
void Parameters_MC_Get_N_Iterations(State *state, int * iterations, int * iterations_
↪log, int idx_image=-1, int idx_chain=-1)
```

Returns the maximum number of iterations and the step size.

8.9.4 Get Parameters

Parameters_MC_Get_Temperature

```
float Parameters_MC_Get_Temperature(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the global base temperature [K].

Parameters_MC_Get_Metropolis_Cone

```
void Parameters_MC_Get_Metropolis_Cone(State *state, bool * cone, float * cone_angle,␣
↪bool * adaptive_cone, float * target_acceptance_ratio, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the Metropolis algorithm configuration.

- whether the spins are displaced within a cone (otherwise: on the entire unit sphere)
- the opening angle within which the spin is placed
- whether the cone angle is automatically adapted to achieve the set acceptance ratio

- target acceptance ratio for the adaptive cone algorithm

Parameters_MC_Get_Random_Sample

```
bool Parameters_MC_Get_Random_Sample(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns whether spins should be sampled randomly or in sequence.

8.10 LLG Parameters

```
#include "Spirit/Parameters_LLG.h"
```

8.10.1 Set Output

Parameters_LLG_Set_Output_Tag

```
void Parameters_LLG_Set_Output_Tag(State *state, const char * tag, int idx_image=-1,␣
↪int idx_chain=-1)
```

Set the tag placed in front of output file names.

If the tag is “”, it will be the date-time of the creation of the state.

Parameters_LLG_Set_Output_Folder

```
void Parameters_LLG_Set_Output_Folder(State *state, const char * folder, int idx_
↪image=-1, int idx_chain=-1)
```

Set the folder, where output files are placed.

Parameters_LLG_Set_Output_General

```
void Parameters_LLG_Set_Output_General(State *state, bool any, bool initial, bool␣
↪final, int idx_image=-1, int idx_chain=-1)
```

Set whether to write any output files at all.

Parameters_LLG_Set_Output_Energy

```
void Parameters_LLG_Set_Output_Energy(State *state, bool energy_step, bool energy_
↪archive, bool energy_spin_resolved, bool energy_divide_by_nos, bool energy_add_
↪readability_lines, int idx_image=-1, int idx_chain=-1)
```

Set whether to write energy output files.

- `step`: whether to write a new file after each set of iterations
- `archive`: whether to append to an archive file after each set of iterations

- `spin_resolved`: whether to write a file containing the energy of each spin
- `divide_by_nos`: whether to divide energies by the number of spins
- `add_readability_lines`: whether to separate columns by lines

Parameters_LLG_Set_Output_Configuration

```
void Parameters_LLG_Set_Output_Configuration(State *state, bool configuration_step,␣
↪bool configuration_archive, int configuration_filetype=IO_Fileformat_OVF_text, int␣
↪idx_image=-1, int idx_chain=-1)
```

Set whether to write spin configuration output files.

- `step`: whether to write a new file after each set of iterations
- `archive`: whether to append to an archive file after each set of iterations
- `filetype`: the format in which the data is written

Parameters_LLG_Set_N_Iterations

```
void Parameters_LLG_Set_N_Iterations(State *state, int n_iterations, int n_iterations_
↪log, int idx_image=-1, int idx_chain=-1)
```

Set the number of iterations and how often to log and write output.

- `n_iterations`: the maximum number of iterations
- `n_iterations_log`: the number of iterations after which status is logged and output written

8.10.2 Set Parameters

Parameters_LLG_Set_Direct_Minimization

```
void Parameters_LLG_Set_Direct_Minimization(State *state, bool direct, int idx_image=-
↪1, int idx_chain=-1)
```

Set whether to minimise the energy without precession.

This only influences dynamics solvers, which will then perform pseudodynamics, simulating only the damping part of the LLG equation.

Parameters_LLG_Set_Convergence

```
void Parameters_LLG_Set_Convergence(State *state, float convergence, int idx_image=-1,
↪ int idx_chain=-1)
```

Set the convergence limit.

When the maximum absolute component value of the force drops below this value, the calculation is considered converged and will stop.

Parameters_LLG_Set_Time_Step

```
void Parameters_LLG_Set_Time_Step(State *state, float dt, int idx_image=-1, int idx_
↪chain=-1)
```

Set the time step [ps] for the calculation.

Parameters_LLG_Set_Damping

```
void Parameters_LLG_Set_Damping(State *state, float damping, int idx_image=-1, int␣
↪idx_chain=-1)
```

Set the Gilbert damping parameter [unitless].

Parameters_LLG_Set_STT

```
void Parameters_LLG_Set_STT(State *state, bool use_gradient, float magnitude, const␣
↪float normal[3], int idx_image=-1, int idx_chain=-1)
```

Set the spin current configuration.

- use_gradient: `True`: use the spatial gradient, `False`: monolayer approximation
- magnitude: current strength
- direction: current direction or polarisation direction, array of shape (3)

Parameters_LLG_Set_Temperature

```
void Parameters_LLG_Set_Temperature(State *state, float T, int idx_image=-1, int idx_
↪chain=-1)
```

Set the (homogeneous) base temperature [K].

Parameters_LLG_Set_Temperature_Gradient

```
void Parameters_LLG_Set_Temperature_Gradient(State *state, float inclination, const␣
↪float direction[3], int idx_image=-1, int idx_chain=-1)
```

Set an additional temperature gradient.

- gradient_inclination: inclination of the temperature gradient [K/a]
- gradient_direction: direction of the temperature gradient, array of shape (3)

8.10.3 Get Output

Parameters_LLG_Get_Output_Tag

```
const char * Parameters_LLG_Get_Output_Tag(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the output file tag.

Parameters_LLG_Get_Output_Folder

```
const char * Parameters_LLG_Get_Output_Folder(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the output folder.

Parameters_LLG_Get_Output_General

```
void Parameters_LLG_Get_Output_General(State *state, bool * any, bool * initial, bool␣
↪* final, int idx_image=-1, int idx_chain=-1)
```

Retrieves whether to write any output at all.

Parameters_LLG_Get_Output_Energy

```
void Parameters_LLG_Get_Output_Energy(State *state, bool * energy_step, bool * energy_
↪archive, bool * energy_spin_resolved, bool * energy_divide_by_nos, bool * energy_
↪add_readability_lines, int idx_image=-1, int idx_chain=-1)
```

Retrieves the energy output settings.

Parameters_LLG_Get_Output_Configuration

```
void Parameters_LLG_Get_Output_Configuration(State *state, bool * configuration_step,␣
↪bool * configuration_archive, int * configuration_filetype, int idx_image=-1, int␣
↪idx_chain=-1)
```

Retrieves the spin configuration output settings.

Parameters_LLG_Get_N_Iterations

```
void Parameters_LLG_Get_N_Iterations(State *state, int * iterations, int * iterations_
↪log, int idx_image=-1, int idx_chain=-1)
```

Returns the maximum number of iterations and the step size.

8.10.4 Get Parameters

Parameters_LLG_Get_Direct_Minimization

```
bool Parameters_LLG_Get_Direct_Minimization(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns whether only energy minimisation will be performed.

Parameters_LLG_Get_Convergence

```
float Parameters_LLG_Get_Convergence(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the convergence value.

Parameters_LLG_Get_Time_Step

```
float Parameters_LLG_Get_Time_Step(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the time step [ps].

Parameters_LLG_Get_Damping

```
float Parameters_LLG_Get_Damping(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the Gilbert damping parameter.

Parameters_LLG_Get_Temperature

```
float Parameters_LLG_Get_Temperature(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the global base temperature [K].

Parameters_LLG_Get_Temperature_Gradient

```
void Parameters_LLG_Get_Temperature_Gradient(State *state, float * direction, float␣
↪normal[3], int idx_image=-1, int idx_chain=-1)
```

Retrieves the temperature gradient.

- inclination of the temperature gradient [K/a]
- direction of the temperature gradient, array of shape (3)

Parameters_LLG_Get_STT

```
void Parameters_LLG_Get_STT(State *state, bool * use_gradient, float * magnitude,␣
↪float normal[3], int idx_image=-1, int idx_chain=-1)
```

Returns the spin current configuration.

- magnitude
- direction, array of shape (3)
- whether the spatial gradient is used

8.11 GNEB Parameters

```
#include "Spirit/Parameters_GNEB.h"
```

8.11.1 GNEB_IMAGE_NORMAL

```
GNEB_IMAGE_NORMAL     0
```

Regular GNEB image type.

8.11.2 GNEB_IMAGE_CLIMBING

```
GNEB_IMAGE_CLIMBING   1
```

Climbing GNEB image type. Climbing images move towards maxima along the path.

8.11.3 GNEB_IMAGE_FALLING

```
GNEB_IMAGE_FALLING    2
```

Falling GNEB image type. Falling images move towards the closest minima.

8.11.4 GNEB_IMAGE_STATIONARY

```
GNEB_IMAGE_STATIONARY 3
```

Stationary GNEB image type. Stationary images are not influenced during a GNEB calculation.

8.11.5 Set Output

Parameters_GNEB_Set_Output_Tag

```
void Parameters_GNEB_Set_Output_Tag(State *state, const char * tag, int idx_chain=-1)
```

Set the tag placed in front of output file names.

If the tag is “”, it will be the date-time of the creation of the state.

Parameters_GNEB_Set_Output_Folder

```
void Parameters_GNEB_Set_Output_Folder(State *state, const char * folder, int idx_
↪chain=-1)
```

Set the folder, where output files are placed.

Parameters_GNEB_Set_Output_General

```
void Parameters_GNEB_Set_Output_General(State *state, bool any, bool initial, bool␣
↪final, int idx_chain=-1)
```

Set whether to write any output files at all.

Parameters_GNEB_Set_Output_Energies

```
void Parameters_GNEB_Set_Output_Energies(State *state, bool step, bool interpolated,␣
↪bool divide_by_nos, bool add_readability_lines, int idx_chain=-1)
```

Set whether to write energy output files.

- `step`: whether to write a new file after each set of iterations
- `interpolated`: whether to write a file containing interpolated reaction coordinate and energy values
- `divide_by_nos`: whether to divide energies by the number of spins
- `add_readability_lines`: whether to separate columns by lines

Parameters_GNEB_Set_Output_Chain

```
void Parameters_GNEB_Set_Output_Chain(State *state, bool step, int filetype=IO_
↪Fileformat_OVF_text, int idx_chain=-1)
```

Set whether to write chain output files.

- `step`: whether to write a new file after each set of iterations
- `filetype`: the format in which the data is written

Parameters_GNEB_Set_N_Iterations

```
void Parameters_GNEB_Set_N_Iterations(State *state, int n_iterations, int n_
↪iterations_log, int idx_chain=-1)
```

Set the number of iterations and how often to log and write output.

- `n_iterations`: the maximum number of iterations
- `n_iterations_log`: the number of iterations after which status is logged and output written

8.11.6 Set Parameters

Parameters_GNEB_Set_Convergence

```
void Parameters_GNEB_Set_Convergence(State *state, float convergence, int idx_image=-
↪1, int idx_chain=-1)
```

Set the convergence limit.

When the maximum absolute component value of the force drops below this value, the calculation is considered converged and will stop.

Parameters_GNEB_Set_Spring_Constant

```
void Parameters_GNEB_Set_Spring_Constant(State *state, float spring_constant, int idx_
↪image=-1, int idx_chain=-1)
```

Set the spring force constant.

Parameters_GNEB_Set_Spring_Force_Ratio

```
void Parameters_GNEB_Set_Spring_Force_Ratio(State *state, float ratio, int idx_chain=-
↪1)
```

Set the ratio between energy and reaction coordinate.

Parameters_GNEB_Set_Path_Shortening_Constant

```
void Parameters_GNEB_Set_Path_Shortening_Constant(State *state, float path_shortening_
↪constant, int idx_chain=-1)
```

Set the path shortening constant.

Parameters_GNEB_Set_Climbing_Falling

```
void Parameters_GNEB_Set_Climbing_Falling(State *state, int image_type, int idx_
↪image=-1, int idx_chain=-1)
```

Set the GNEB image type (see the integers defined above).

Parameters_GNEB_Set_Image_Type_Automatically

```
void Parameters_GNEB_Set_Image_Type_Automatically(State *state, int idx_chain=-1)
```

Automatically set GNEB image types.

Minima along the path will be set to falling, maxima to climbing and the rest to regular.

Parameters_GNEB_Set_N_Energy_Interpolations

```
void Parameters_GNEB_Set_N_Energy_Interpolations(State *state, int n, int idx_chain=-
↪1)
```

Returns the maximum number of iterations and the step size.

8.11.7 Get Output

Parameters_GNEB_Get_Output_Tag

```
const char * Parameters_GNEB_Get_Output_Tag(State *state, int idx_chain=-1)
```

Returns the output file tag.

Parameters_GNEB_Get_Output_Folder

```
const char * Parameters_GNEB_Get_Output_Folder(State *state, int idx_chain=-1)
```

Returns the output folder.

Parameters_GNEB_Get_Output_General

```
void Parameters_GNEB_Get_Output_General(State *state, bool * any, bool * initial,␣
↪bool * final, int idx_chain=-1)
```

Retrieves whether to write any output at all.

Parameters_GNEB_Get_Output_Energies

```
void Parameters_GNEB_Get_Output_Energies(State *state, bool * step, bool *␣
↪interpolated, bool * divide_by_nos, bool * add_readability_lines, int idx_chain=-1)
```

Retrieves the energy output settings.

Parameters_GNEB_Get_Output_Chain

```
void Parameters_GNEB_Get_Output_Chain(State *state, bool * step, int * filetype, int␣
↪idx_chain=-1)
```

Retrieves the chain output settings.

Parameters_GNEB_Get_N_Iterations

```
void Parameters_GNEB_Get_N_Iterations(State *state, int * iterations, int *␣
↪iterations_log, int idx_chain=-1)
```

Returns the maximum number of iterations and the step size.

8.11.8 Get Parameters

Parameters_GNEB_Get_Convergence

```
float Parameters_GNEB_Get_Convergence(State *state, int idx_image=-1, int idx_chain=-
↪1)
```

Simulation Parameters

Parameters_GNEB_Get_Spring_Constant

```
float Parameters_GNEB_Get_Spring_Constant(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the spring force constant.

Parameters_GNEB_Get_Spring_Force_Ratio

```
float Parameters_GNEB_Get_Spring_Force_Ratio(State *state, int idx_chain=-1)
```

Returns the spring force cratio of energy to reaction coordinate.

Parameters_GNEB_Get_Path_Shortening_Constant

```
float Parameters_GNEB_Get_Path_Shortening_Constant(State *state, int idx_chain=-1)
```

Return the path shortening constant.

Parameters_GNEB_Get_Climbing_Falling

```
int Parameters_GNEB_Get_Climbing_Falling(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the integer of whether an image is regular, climbing, falling, or stationary.

The integers are defined above.

Parameters_GNEB_Get_N_Energy_Interpolations

```
int Parameters_GNEB_Get_N_Energy_Interpolations(State *state, int idx_chain=-1)
```

Returns the number of energy values interpolated between images.

8.12 EMA Parameters

```
#include "Spirit/Parameters_EMA.h"
```

This method, if needed, calculates modes (they can also be read in from a file) and perturbs the spin system periodically in the direction of the eigenmode.

8.12.1 Set

Parameters_EMA_Set_N_Modes

```
void Parameters_EMA_Set_N_Modes(State *state, int n_modes, int idx_image=-1, int idx_
↪chain=-1)
```

Set the number of modes to calculate or use.

Parameters_EMA_Set_N_Mode_Follow

```
void Parameters_EMA_Set_N_Mode_Follow(State *state, int n_mode_follow, int idx_image=-
↪1, int idx_chain=-1)
```

Set the index of the mode to use.

Parameters_EMA_Set_Frequency

```
void Parameters_EMA_Set_Frequency(State *state, float frequency, int idx_image=-1,␣
↪int idx_chain=-1)
```

Set the frequency with which the mode is applied.

Parameters_EMA_Set_Amplitude

```
void Parameters_EMA_Set_Amplitude(State *state, float amplitude, int idx_image=-1,␣
↪int idx_chain=-1)
```

Set the amplitude with which the mode is applied.

Parameters_EMA_Set_Snapshot

```
void Parameters_EMA_Set_Snapshot(State *state, bool snapshot, int idx_image=-1, int␣
↪idx_chain=-1)
```

Set whether to displace the system statically instead of periodically.

8.12.2 Get

Parameters_EMA_Get_N_Modes

```
int Parameters_EMA_Get_N_Modes(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the number of modes to calculate or use.

Parameters_EMA_Get_N_Mode_Follow

```
int Parameters_EMA_Get_N_Mode_Follow(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the index of the mode to use.

Parameters_EMA_Get_Frequency

```
float Parameters_EMA_Get_Frequency(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the frequency with which the mode is applied.

Parameters_EMA_Get_Amplitude

```
float Parameters_EMA_Get_Amplitude(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the amplitude with which the mode is applied.

Parameters_EMA_Get_Snapshot

```
bool Parameters_EMA_Get_Snapshot(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns whether to displace the system statically instead of periodically.

8.13 MMF Parameters

```
#include "Spirit/Parameters_MMF.h"
```

8.13.1 Set Output

Parameters_MMF_Set_Output_Tag

```
void Parameters_MMF_Set_Output_Tag(State *state, const char * tag, int idx_image=-1,
↪int idx_chain=-1)
```

Set the tag placed in front of output file names.

If the tag is “”, it will be the date-time of the creation of the state.

Parameters_MMF_Set_Output_Folder

```
void Parameters_MMF_Set_Output_Folder(State *state, const char * folder, int idx_
↪image=-1, int idx_chain=-1)
```

Set the folder, where output files are placed.

Parameters_MMF_Set_Output_General

```
void Parameters_MMF_Set_Output_General(State *state, bool any, bool initial, bool
↪final, int idx_image=-1, int idx_chain=-1)
```

Set whether to write any output files at all.

Parameters_MMF_Set_Output_Energy

```
void Parameters_MMF_Set_Output_Energy(State *state, bool step, bool archive, bool
↪spin_resolved, bool divide_by_nos, bool add_readability_lines, int idx_image=-1,
↪int idx_chain=-1)
```

Set whether to write energy output files.

- `step`: whether to write a new file after each set of iterations
- `archive`: whether to append to an archive file after each set of iterations
- `spin_resolved`: whether to write a file containing the energy of each spin
- `divide_by_nos`: whether to divide energies by the number of spins

- `add_readability_lines`: whether to separate columns by lines

Parameters_MMF_Set_Output_Configuration

```
void Parameters_MMF_Set_Output_Configuration(State *state, bool step, bool archive,
↪int filetype, int idx_image=-1, int idx_chain=-1)
```

Set whether to write spin configuration output files.

- `step`: whether to write a new file after each set of iterations
- `archive`: whether to append to an archive file after each set of iterations
- `filetype`: the format in which the data is written

Parameters_MMF_Set_N_Iterations

```
void Parameters_MMF_Set_N_Iterations(State *state, int n_iterations, int n_iterations_
↪log, int idx_image=-1, int idx_chain=-1)
```

Set the number of iterations and how often to log and write output.

- `n_iterations`: the maximum number of iterations
- `n_iterations_log`: the number of iterations after which status is logged and output written

8.13.2 Set Parameters

Parameters_MMF_Set_N_Modes

```
void Parameters_MMF_Set_N_Modes(State *state, int n_modes, int idx_image=-1, int idx_
↪chain=-1)
```

Set the number of modes to be calculated at each iteration.

Parameters_MMF_Set_N_Mode_Follow

```
void Parameters_MMF_Set_N_Mode_Follow(State *state, int n_mode_follow, int idx_image=-
↪1, int idx_chain=-1)
```

Set the index of the mode to follow.

8.13.3 Get Output

Parameters_MMF_Get_Output_Tag

```
const char * Parameters_MMF_Get_Output_Tag(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the output file tag.

Parameters_MMF_Get_Output_Folder

```
const char * Parameters_MMF_Get_Output_Folder(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the output folder.

Parameters_MMF_Get_Output_General

```
void Parameters_MMF_Get_Output_General(State *state, bool * any, bool * initial, bool␣
↪* final, int idx_image=-1, int idx_chain=-1)
```

Retrieves whether to write any output at all.

Parameters_MMF_Get_Output_Energy

```
void Parameters_MMF_Get_Output_Energy(State *state, bool * step, bool * archive, bool␣
↪* spin_resolved, bool * divide_by_nos, bool * add_readability_lines, int idx_image=-
↪1, int idx_chain=-1)
```

Retrieves the energy output settings.

Parameters_MMF_Get_Output_Configuration

```
void Parameters_MMF_Get_Output_Configuration(State *state, bool * step, bool *␣
↪archive, int * filetype, int idx_image=-1, int idx_chain=-1)
```

Retrieves the spin configuration output settings.

Parameters_MMF_Get_N_Iterations

```
void Parameters_MMF_Get_N_Iterations(State *state, int * iterations, int * iterations_
↪log, int idx_image=-1, int idx_chain=-1)
```

Returns the maximum number of iterations and the step size.

8.13.4 Get Parameters

Parameters_MMF_Get_N_Modes

```
int Parameters_MMF_Get_N_Modes(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the number of modes calculated at each iteration.

Parameters_MMF_Get_N_Mode_Follow

```
int Parameters_MMF_Get_N_Mode_Follow(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the index of the mode which to follow.

8.14 Quantities

```
#include "Spirit/Quantities.h"
```

8.14.1 Quantity_Get_Magnetization

```
void Quantity_Get_Magnetization(State * state, float m[3], int idx_image=-1, int idx_
↪chain=-1)
```

Total Magnetization

8.14.2 Quantity_Get_Topological_Charge

```
float Quantity_Get_Topological_Charge(State * state, int idx_image=-1, int idx_chain=-
↪1)
```

Topological Charge

8.14.3 Quantity_Get_Grad_Force_MinimumMode

```
void Quantity_Get_Grad_Force_MinimumMode(State * state, float * gradient, float *␣
↪eval, float * mode, float * forces, int idx_image=-1, int idx_chain=-1)
```

Minimum mode following information

8.15 Simulation

```
#include "Spirit/Simulation.h"
```

This API of Spirit is used to run and monitor iterative calculation methods.

If many iterations are called individually, one should use the single shot simulation functionality. It avoids the allocations etc. involved when a simulation is started and ended and behaves like a regular simulation, except that the iterations have to be triggered manually.

8.15.1 Definition of solvers

Note that the VP and NCG Solvers are only meant for direct minimization and not for dynamics.

Solver_VP

```
Solver_VP          0
```

`VP`: Verlet-like velocity projection

Solver_SIB

```
Solver_SIB          1
```

`SIB`: Verlet-like velocity projection

Solver_Depondt

```
Solver_Depondt      2
```

`Depondt`: Verlet-like velocity projection

Solver_Heun

```
Solver_Heun         3
```

`Heun`: Verlet-like velocity projection

Solver_RungeKutta4

```
Solver_RungeKutta4 4
```

`RK4`: Verlet-like velocity projection

8.15.2 Start or stop a simulation

PREFIX

```

```

Monte Carlo

Simulation_LLG_Start

```
void Simulation_LLG_Start(State *state, int solver_type, int n_iterations=-1, int n_
↪iterations_log=-1, bool singleshot=false, int idx_image=-1, int idx_chain=-1)
```

Landau-Lifshitz-Gilbert dynamics and energy minimisation

Simulation_GNEB_Start

```
void Simulation_GNEB_Start(State *state, int solver_type, int n_iterations=-1, int n_
↪iterations_log=-1, bool singleshot=false, int idx_chain=-1)
```

Geodesic nudged elastic band method

Simulation_MMF_Start

```
void Simulation_MMF_Start(State *state, int solver_type, int n_iterations=-1, int n_
↪iterations_log=-1, bool singleshot=false, int idx_image=-1, int idx_chain=-1)
```

Minimum mode following method

Simulation_EMA_Start

```
void Simulation_EMA_Start(State *state, int n_iterations=-1, int n_iterations_log=-1,␣
↪bool singleshot=false, int idx_image=-1, int idx_chain=-1)
```

Eigenmode analysis

Simulation_SingleShot

```
void Simulation_SingleShot(State *state, int idx_image=-1, int idx_chain=-1)
```

Single iteration of a Method

If `singleshot=true` was passed to `Simulation_..._Start` before, this will perform one iteration. Otherwise, nothing will happen.

Simulation_Stop

```
void Simulation_Stop(State *state, int idx_image=-1, int idx_chain=-1)
```

Stop a simulation running on an image or chain

Simulation_Stop_All

```
void Simulation_Stop_All(State *state)
```

Stop all simulations

8.15.3 Get information

Simulation_Get_MaxTorqueComponent

```
float Simulation_Get_MaxTorqueComponent(State * state, int idx_image=-1, int idx_
↪chain=-1)
```

Get maximum torque component.

If a MC, LLG, MMF or EMA simulation is running this returns the max. torque on the current image.

If a GNEB simulation is running this returns the max. torque on the current chain.

Simulation_Get_Chain_MaxTorqueComponents

```
void Simulation_Get_Chain_MaxTorqueComponents(State * state, float * torques, int idx_
↪chain=-1)
```

Get maximum torque components on the images of a chain.

Will only work if a GNEB simulation is running.

Simulation_Get_IterationsPerSecond

```
float Simulation_Get_IterationsPerSecond(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Returns the iterations per second (IPS).

If a MC, LLG, MMF or EMA simulation is running this returns the IPS on the current image.

If a GNEB simulation is running this returns the IPS on the current chain.

Simulation_Get_Iteration

```
int Simulation_Get_Iteration(State *state, int idx_image=-1, int idx_chain=-1)
```

Returns the number of iterations performed by the current simulation so far.

Simulation_Get_Time

```
float Simulation_Get_Time(State *state, int idx_image=-1, int idx_chain=-1)
```

Get time passed by the simulation [ps]

Returns:

- if an LLG simulation is running returns the cumulatively summed time steps `dt`
- otherwise returns 0

Simulation_Get_Wall_Time

```
int Simulation_Get_Wall_Time(State *state, int idx_image=-1, int idx_chain=-1)
```

Get number of miliseconds of wall time since the simulation was started

Simulation_Get_Solver_Name

```
const char * Simulation_Get_Solver_Name(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Get name of the currently used method.

If a MC, LLG, MMF or EMA simulation is running this returns the Solver name on the current image.

If a GNEB simulation is running this returns the Solver name on the current chain.

Simulation_Get_Method_Name

```
const char * Simulation_Get_Method_Name(State *state, int idx_image=-1, int idx_
↪chain=-1)
```

Get name of the currently used method.

If a MC, LLG, MMF or EMA simulation is running this returns the Method name on the current image.

If a GNEB simulation is running this returns the Method name on the current chain.

8.15.4 Whether a simulation is running

Simulation_Running_On_Image

```
bool Simulation_Running_On_Image(State *state, int idx_image=-1, int idx_chain=-1)
```

Check if a simulation is running on specific image of specific chain

Simulation_Running_On_Chain

```
bool Simulation_Running_On_Chain(State *state, int idx_chain=-1)
```

Check if a simulation is running across a specific chain

Simulation_Running_Anywhere_On_Chain

```
bool Simulation_Running_Anywhere_On_Chain(State *state, int idx_chain=-1)
```

Check if a simulation is running on any or all images of a chain

8.16 State

```
#include "Spirit/State.h"
```

To create a new state with one chain containing a single image, initialized by an input file, and run the most simple example of a **spin dynamics simulation**:

```
#import "Spirit/State.h"
#import "Spirit/Simulation.h"

const char * cfgfile = "input/input.cfg";  // Input file
State * p_state = State_Setup(cfgfile);    // State setup
Simulation_LLG_Start(p_state, Solver_SIB); // Start a LLG simulation using the SIB␣
↪solver
State_Delete(p_state)                      // State cleanup
```

8.16.1 State

```
struct State
```

The opaque state struct, containing all calculation data.

```
+-----------------------------+
| State                       |
| +-------------------------+ |
| | Chain                   | |
| | +-------------------+   | |
| | | 0th System  Image |   | |
| | +-------------------+   | |
| | +-------------------+   | |
| | | 1st System  Image |   | |
| | +-------------------+   | |
| |   .                     | |
| |   .                     | |
| |   .                     | |
| | +-------------------+   | |
| | | Nth System  Image |   | |
| | +-------------------+   | |
| +-------------------------+ |
+-----------------------------+
```

This is passed to and is operated on by the API functions.

A new state can be created with `State_Setup()`, where you can pass a config file specifying your initial system parameters. If you do not pass a config file, the implemented defaults are used. **Note that you currently cannot change the geometry of the systems in your state once they are initialized.**

8.16.2 State_Setup

```
State * State_Setup(const char * config_file = "", bool quiet = false)
```

Create the State and fill it with initial data.

- `config_file`: if a config file is given, it will be parsed for keywords specifying the initial values. Otherwise, defaults are used
- `quiet`: if `true`, the defaults are changed such that only very few messages will be printed to the console and no output files are written

8.16.3 State_Delete

```
void State_Delete(State * state)
```

Correctly deletes a State and frees the corresponding memory.

8.16.4 State_Update

```
void State_Update(State * state)
```

Update the state to hold current values.

8.16.5 State_To_Config

```
void State_To_Config(State * state, const char * config_file, const char * original_
↪config_file="")
```

Write a config file which should give the same state again when used in State_Setup (modulo the number of chains and images)

8.16.6 State_DateTime

```
const char * State_DateTime(State * state)
```

Returns a string containing the datetime tag (timepoint of creation) of this state.

Format: `yyyy-mm-dd_hh-mm-ss`

8.17 System

```
#include "Spirit/System.h"
```

Spin systems are often referred to as "images" throughout Spirit. The `idx_image` is used throughout the API to specify which system out of the chain a function should be applied to. `idx_image=-1` refers to the active image of the chain.

8.17.1 System_Get_Index

```
int System_Get_Index(State * state)
```

Returns the index of the currently active spin system in the chain.

8.17.2 System_Get_NOS

```
int System_Get_NOS(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns the number of spins (NOS) of a spin system.

8.17.3 System_Get_Spin_Directions

```
scalar * System_Get_Spin_Directions(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns a pointer to the spin orientations data.

The array is contiguous and of shape (NOS, 3).

8.17.4 System_Get_Effective_Field

```
scalar * System_Get_Effective_Field(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns a pointer to the effective field data.

The array is contiguous and of shape (NOS, 3).

8.17.5 System_Get_Eigenmode

```
scalar * System_Get_Eigenmode(State * state, int idx_mode, int idx_image=-1, int idx_
↪chain=-1)
```

Returns a pointer to the data of the N'th eigenmode of a spin system.

The array is contiguous and of shape (NOS, 3).

8.17.6 System_Get_Rx

```
float System_Get_Rx(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns the reaction coordinate of a system along the chain.

8.17.7 System_Get_Energy

```
float System_Get_Energy(State * state, int idx_image=-1, int idx_chain=-1)
```

Returns the energy of a spin system.

8.17.8 System_Get_Energy_Array

```
void System_Get_Energy_Array(State * state, float * energies, int idx_image=-1, int␣
↪idx_chain=-1)
```

Retrieves the energy contributions of a spin system.

8.17.9 System_Get_Eigenvalues

```
void System_Get_Eigenvalues(State * state, float * eigenvalues, int idx_image=-1, int␣
↪idx_chain=-1)
```

Retrieves the eigenvalues of a spin system

8.17.10 System_Print_Energy_Array

```
void System_Print_Energy_Array(State * state, int idx_image=-1, int idx_chain=-1)
```

Write the energy as formatted output to the console

8.17.11 System_Update_Data

```
void System_Update_Data(State * state, int idx_image=-1, int idx_chain=-1)
```

Update Data (primarily for plots)

8.17.12 System_Update_Eigenmodes

```
void System_Update_Eigenmodes(State *state, int idx_image=-1, int idx_chain=-1)
```

Update Eigenmodes (primarily for visualisation or saving)

8.18 Transitions

```
#include "Spirit/Transitions.h"
```

Setting transitions between spin configurations over a chain.

8.18.1 Transition_Homogeneous

```
void Transition_Homogeneous(State *state, int idx_1, int idx_2, int idx_chain=-1)
```

A linear interpolation between two spin configurations on a chain.

The spins are moved along great circles connecting the start and end points, making it the shortest possible connection path between the two configurations.

- `idx_1`: the index of the first image
- `idx_2`: the index of the second image. `idx_2 > idx_1` is required

8.18.2 Transition_Add_Noise_Temperature

```
void Transition_Add_Noise_Temperature(State *state, float temperature, int idx_1, int␣
↪idx_2, int idx_chain=-1)
```

Adds some stochastic noise to the transition between two images.

- `temperature`: a measure of the intensity of the noise
- `idx_1`: the index of the first image
- `idx_2`: the index of the second image. `idx_2 > idx_1` is required

CHAPTER 9

Usage

9.1 Energy minimisation

Energy minimisation of a spin system can be performed using the LLG method and the velocity projection (VP) solver:

```
from spirit import simulation, state

with state.State("input/input.cfg") as p_state:
    simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_VP)
```

or using one of the dynamics solvers, using dissipative dynamics:

```
from spirit import parameters, simulation, state

with state.State("input/input.cfg") as p_state:
    parameters.llg.set_direct_minimization(p_state, True)
    simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_DEPONDT)
```

9.2 LLG method

To perform an LLG dynamics simulation:

```
from spirit import simulation, state

with state.State("input/input.cfg") as p_state:
    simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_DEPONDT)
```

Note that the velocity projection (VP) solver is not a dynamics solver.

9.3 GNEB method

The geodesic nudged elastic band method. See also the method paper.

This method operates on a transition between two spin configurations, discretised by "images" on a "chain". The procedure follows these steps:

1. set the number of images
2. set the initial and final spin configuration
3. create an initial guess for the transition path
4. run an initial GNEB relaxation
5. determine and set the suitable images on the chain to converge on extrema
6. run a full GNEB relaxation using climbing and falling images

```
from spirit import state, chain, configuration, transition, simulation

noi = 7

with state.State("input/input.cfg") as p_state:
    ### Copy the first image and set chain length
    chain.image_to_clipboard(p_state)
    chain.set_length(p_state, noi)

    ### First image is homogeneous with a Skyrmion in the center
    configuration.plus_z(p_state, idx_image=0)
    configuration.skyrmion(p_state, 5.0, phase=-90.0, idx_image=0)
    simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_VP, idx_
↪image=0)
    ### Last image is homogeneous
    configuration.plus_z(p_state, idx_image=noi-1)
    simulation.start(p_state, simulation.METHOD_LLG, simulation.SOLVER_VP, idx_
↪image=noi-1)

    ### Create initial guess for transition: homogeneous rotation
    transition.homogeneous(p_state, 0, noi-1)

    ### Initial GNEB relaxation
    simulation.start(p_state, simulation.METHOD_GNEB, simulation.SOLVER_VP, n_
↪iterations=5000)
    ### Automatically set climbing and falling images
    chain.set_image_type_automatically(p_state)
    ### Full GNEB relaxation
    simulation.start(p_state, simulation.METHOD_GNEB, simulation.SOLVER_VP)
```

9.4 HTST

The harmonic transition state theory. See also the method paper.

The usage of this method is not yet documented.

9.5 MMF method

The minimum mode following method. See also the method paper.

The usage of this method is not yet documented.

CHAPTER 10

Full API reference

10.1 Chain

Manipulate the chain of spin systems (also called images), e.g. add, remove or change active image. Get information, such as number of images or energies and reaction coordinates.

`spirit.chain.`**`delete_image`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Removes the specified image from the chain.

Note that the active image might change.

If it is the last remaining image in the chain, no action is taken.

`spirit.chain.`**`get_energy`**(*p_state*, *idx_chain=-1*)
Returns an array of shape (NOI) containing the energies of the images.

`spirit.chain.`**`get_energy_interpolated`**(*p_state*, *idx_chain=-1*)
Returns an array containing the interpolated energy values along the chain.

The number of interpolated values between images can be set in the GNEB parameters.

`spirit.chain.`**`get_noi`**(*p_state*, *idx_chain=-1*)
Get number of images (NOI) in the chain.

`spirit.chain.`**`get_reaction_coordinate`**(*p_state*, *idx_chain=-1*)
Returns an array of shape (NOI) containing the reaction coordinates of the images.

`spirit.chain.`**`get_reaction_coordinate_interpolated`**(*p_state*, *idx_chain=-1*)
Returns an array containing the interpolated reaction coordinate values along the chain.

The number of interpolated values between images can be set in the GNEB parameters.

`spirit.chain.`**`image_to_clipboard`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Copies an image to the clipboard of Spirit. It can then be later e.g. inserted or appended.

`spirit.chain.`**`insert_image_after`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Inserts an image after the specified index.

Note that the active image might change.

If no image is in the clipboard, no action is taken.

`spirit.chain.insert_image_before`(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Inserts an image in front of the specified index.

Note that the active image might change.

If no image is in the clipboard, no action is taken.

`spirit.chain.jump_to_image`(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Set the index of the active image in the chain.

`spirit.chain.next_image`(*p_state*, *idx_chain=-1*)
Switch the active image index to the next highest in the chain.

`spirit.chain.pop_back`(*p_state*, *idx_chain=-1*)
Removes the last image from the chain.

Note that the active image might change.

If it is the last remaining image in the chain, no action is taken.

`spirit.chain.prev_image`(*p_state*, *idx_chain=-1*)
Switch the active image index to the next lowest in the chain.

`spirit.chain.push_back`(*p_state*, *idx_chain=-1*)
Appends an image to the chain.

If no image is in the clipboard, no action is taken.

`spirit.chain.replace_image`(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Replaces the image from the one in the clipboard.

If no image is in the clipboard, no action is taken.

`spirit.chain.set_length`(*p_state*, *n_images*, *idx_chain=-1*)
Set the number of images (NOI) in the chain.

If the chain is longer, the corresponding number of images is erased from the end.

If the chain is shorter, the corresponding number of images is appended.

Note that the active image might change.

If no image is in the clipboard, no action is taken.

`spirit.chain.setup_data`(*p_state*, *idx_chain=-1*)

`spirit.chain.update_data`(*p_state*, *idx_chain=-1*)
Updates various data of the chain, including:

- Energies of images
- Reaction coordinates of images
- Interpolated energy and reaction coordinate values

10.2 Configuration

Set various spin configurations, such as homogeneous domains, spirals or skyrmions.

`spirit.configuration.`**`add_noise`**(*p_state, temperature, pos=[0, 0, 0], border_rectangular=[-1, -1, -1], border_cylindrical=-1, border_spherical=-1, inverted=False, idx_image=-1, idx_chain=-1*)

Add temperature-scaled random noise to configuration.

`spirit.configuration.`**`domain`**(*p_state, dir, pos=[0, 0, 0], border_rectangular=[-1, -1, -1], border_cylindrical=-1, border_spherical=-1, inverted=False, idx_image=-1, idx_chain=-1*)

Set a domain (homogeneous) configuration.

`spirit.configuration.`**`hopfion`**(*p_state, radius, order=1, pos=[0, 0, 0], border_rectangular=[-1, -1, -1], border_cylindrical=-1, border_spherical=-1, inverted=False, idx_image=-1, idx_chain=-1*)

Set a Hopfion configuration.

- radius: the distance from the center to the center of the corresponding tubular isosurface
- order: TODO

In contrast to the skyrmion, it extends over the whole allowed space.

`spirit.configuration.`**`minus_z`**(*p_state, pos=[0, 0, 0], border_rectangular=[-1, -1, -1], border_cylindrical=-1, border_spherical=-1, inverted=False, idx_image=-1, idx_chain=-1*)

Set a -z (homogeneous) configuration.

`spirit.configuration.`**`plus_z`**(*p_state, pos=[0.0, 0.0, 0.0], border_rectangular=[-1.0, -1.0, -1.0], border_cylindrical=-1.0, border_spherical=-1.0, inverted=False, idx_image=-1, idx_chain=-1*)

Set a +z (homogeneous) configuration.

`spirit.configuration.`**`random`**(*p_state, pos=[0, 0, 0], border_rectangular=[-1, -1, -1], border_cylindrical=-1, border_spherical=-1, inverted=False, idx_image=-1, idx_chain=-1*)

Distribute all spins randomly on the unit sphere.

`spirit.configuration.`**`set_atom_type`**(*p_state, atom_type=0, pos=[0.0, 0.0, 0.0], border_rectangular=[-1.0, -1.0, -1.0], border_cylindrical=-1.0, border_spherical=-1.0, inverted=False, idx_image=-1, idx_chain=-1*)

Set the type of the atoms in the given region (default: 0).

This can be used e.g. to insert defects (-1).

`spirit.configuration.`**`set_pinned`**(*p_state, pinned, pos=[0.0, 0.0, 0.0], border_rectangular=[-1.0, -1.0, -1.0], border_cylindrical=-1.0, border_spherical=-1.0, inverted=False, idx_image=-1, idx_chain=-1*)

Set whether the spins within the given region are pinned or not.

If they are pinned, they are pinned to their current orientation.

`spirit.configuration.`**`skyrmion`**(*p_state, radius, order=1, phase=1, up_down=False, achiral=False, right_left=False, pos=[0, 0, 0], border_rectangular=[-1, -1, -1], border_cylindrical=-1, border_spherical=-1, inverted=False, idx_image=-1, idx_chain=-1*)

Set a skyrmion configuration.

- radius: the extent of the skyrmion, at which it points approximately upwards
- order: the number of twists along a circle cutting the skyrmion
- phase: 0 corresponds to a Neel skyrmion, -90 to a Bloch skyrmion
- up_down: if *True*, the z-orientation is inverted

- achiral: if *True*, the topological charge is inverted
- right_left: if *True*, the in-plane rotation is inverted

The skyrmion only extends up to *radius*, meaning that *border_cylindrical* is not usually necessary.

`spirit.configuration.`**`spin_spiral`**(*p_state, direction_type, q_vector, axis, theta, pos=[0, 0, 0], border_rectangular=[-1, -1, -1], border_cylindrical=-1, border_spherical=-1, inverted=False, idx_image=-1, idx_chain=-1*)

Set a spin spiral configuration.

TODO: document parameters - direction_type: - q_vector: - axis: - theta:

10.3 Constants

`spirit.constants.`**`g_e = 2.00231930436182`**
Electron g-factor [unitless]

`spirit.constants.`**`gamma = 0.1760859644`**
Gyromagnetic ratio of electron [rad / (ps*T)]

`spirit.constants.`**`hbar = 0.6582119514`**
Planck's constant [meV*ps / rad]

`spirit.constants.`**`k_B = 0.0861733035`**
The Boltzmann constant [meV / K]

`spirit.constants.`**`mRy = 0.07349864496711137`**
Millirydberg [mRy / meV]

`spirit.constants.`**`mu_0 = 2.0133545e-28`**
The vacuum permeability [T^2 m^3 / meV]

`spirit.constants.`**`mu_B = 0.057883817555`**
The Bohr Magneton [meV / T]

`spirit.constants.`**`pi = 3.141592653589793`**
Pi [rad]

10.4 Geometry

Change or get info on the current geometrical configuration, e.g. number of cells in the three crystal translation directions.

`spirit.geometry.`**`get_atom_types`**(*p_state, idx_image=-1, idx_chain=-1*)
Get the types of all atoms as a *numpy.array_view* of shape (NOS).

If e.g. disorder is activated, this allows to view and manipulate the types of individual atoms.

`spirit.geometry.`**`get_bounds`**(*p_state, idx_image=-1, idx_chain=-1*)
Get the bounds of the system in global coordinates.

Returns two arrays of shape (3) containing minimum and maximum bounds respectively.

`spirit.geometry.`**`get_bravais_lattice_type`**(*p_state, idx_image=-1, idx_chain=-1*)
Get the bravais lattice type corresponding to one of the integers defined above.

`spirit.geometry.`**`get_bravais_vectors`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Get the Bravais vectors.

Returns three arrays of shape (3).

`spirit.geometry.`**`get_center`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Get the center of the system in global coordinates.

Returns an array of shape (3).

`spirit.geometry.`**`get_dimensionality`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Get the dimensionality of the geometry.

`spirit.geometry.`**`get_n_cell_atoms`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Get the number of atoms in the basis cell.

`spirit.geometry.`**`get_n_cells`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Get the number of basis cells along the three bravais vectors.

Returns an array of shape (3).

`spirit.geometry.`**`get_positions`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns a *numpy.array_view* of shape (NOS, 3) with the components of each spins position.

Changing the contents of this array_view will have direct effect on the state and should not be done.

`spirit.geometry.`**`set_bravais_lattice_type`**(*p_state*, *lattice_type*, *idx_image=-1*, *idx_chain=-1*)
Set the bravais vectors to a pre-defined lattice type:

- sc: simple cubic
- bcc: body centered cubic
- fcc: face centered cubic
- hex2d: hexagonal (120deg)
- hed2d120: hexagonal (120deg)
- hex2d60: hexagonal (60deg)

`spirit.geometry.`**`set_bravais_vectors`**(*p_state, ta=[1.0, 0.0, 0.0], tb=[0.0, 1.0, 0.0], tc=[0.0, 0.0, 1.0], idx_image=-1, idx_chain=-1*)
Manually specify the bravais vectors.

`spirit.geometry.`**`set_cell_atom_types`**(*p_state*, *atom_types*, *idx_image=-1*, *idx_chain=-1*)
Set the atom types of the atoms in the basis cell.

`spirit.geometry.`**`set_lattice_constant`**(*p_state*, *lattice_constant*, *idx_image=-1*, *idx_chain=-1*)
Set the global lattice scaling constant.

`spirit.geometry.`**`set_mu_s`**(*p_state*, *mu_s*, *idx_image=-1*, *idx_chain=-1*)
Set the magnetic moment of all atoms.

`spirit.geometry.`**`set_n_cells`**(*p_state, n_cells=[1, 1, 1], idx_image=-1, idx_chain=-1*)
Set the number of basis cells along the three bravais vectors.

10.5 Hamiltonian

Set the parameters of the Heisenberg Hamiltonian, such as external field or exchange interaction.

spirit.hamiltonian.**CHIRALITY_BLOCH = 1**
DMI Bloch chirality type for neighbour shells

spirit.hamiltonian.**CHIRALITY_BLOCH_INVERSE = -1**
DMI Bloch chirality type for neighbour shells with opposite sign

spirit.hamiltonian.**CHIRALITY_NEEL = 2**
DMI Neel chirality type for neighbour shells

spirit.hamiltonian.**CHIRALITY_NEEL_INVERSE = -2**
DMI Neel chirality type for neighbour shells with opposite sign

spirit.hamiltonian.**DDI_METHOD_CUTOFF = 3**
Dipole-dipole interaction: use a direct summation with a cutoff radius

spirit.hamiltonian.**DDI_METHOD_FFT = 1**
Dipole-dipole interaction: use FFT convolutions

spirit.hamiltonian.**DDI_METHOD_FMM = 2**
Dipole-dipole interaction: use a fast multipole method (FMM)

spirit.hamiltonian.**DDI_METHOD_NONE = 0**
Dipole-dipole interaction: do not calculate

spirit.hamiltonian.**get_boundary_conditions**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns an array of shape (3) containing the boundary conditions in the three translation directions [a, b, c] of the lattice.

spirit.hamiltonian.**get_ddi**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the cutoff radius of the DDI.

spirit.hamiltonian.**get_field**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the magnitude and an array of shape (3) containing the direction of the external magnetic field.

spirit.hamiltonian.**get_name**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns a string containing the name of the Hamiltonian currently in use.

spirit.hamiltonian.**set_anisotropy**(*p_state*, *magnitude*, *direction*, *idx_image=-1*, *idx_chain=-1*)
Set the (homogeneous) magnetocrystalline anisotropy.

spirit.hamiltonian.**set_boundary_conditions**(*p_state*, *boundaries*, *idx_image=-1*, *idx_chain=-1*)
Set the boundary conditions along the translation directions [a, b, c].

0 = open, 1 = periodical

spirit.hamiltonian.**set_ddi**(*p_state, ddi_method, n_periodic_images=[4, 4, 4], radius=0.0, idx_image=-1, idx_chain=-1*)
Set the dipolar interaction calculation method.

- ddi_method – one of the integers defined above
- n_periodic_images – the number of periodical images in the three translation directions, taken into account when boundaries in the corresponding direction are periodical
- radius – the cutoff radius for the direct summation method

spirit.hamiltonian.**set_dmi**(*p_state*, *n_shells*, *D_ij*, *chirality=1*, *idx_image=-1*, *idx_chain=-1*)
Set the Dzyaloshinskii-Moriya interaction in terms of neighbour shells.

spirit.hamiltonian.**set_exchange**(*p_state*, *n_shells*, *J_ij*, *idx_image=-1*, *idx_chain=-1*)
Set the Exchange interaction in terms of neighbour shells.

`spirit.hamiltonian.set_field`(*p_state*, *magnitude*, *direction*, *idx_image=-1*, *idx_chain=-1*)
Set the (homogeneous) external magnetic field.

10.6 HTST

Harmonic transition state theory.

Note that *calculate_prefactor* needs to be called before using any of the getter functions.

`spirit.htst.calculate`(*p_state*, *idx_image_minimum*, *idx_image_sp*, *idx_chain=-1*)
Performs an HTST calculation and returns rate prefactor.

Note: this function must be called before any of the getters.

`spirit.htst.get_eigenvalues_min`(*p_state*, *idx_chain=-1*)
Returns the eigenvalues at the minimum. Shape (2*nos)

`spirit.htst.get_eigenvalues_sp`(*p_state*, *idx_chain=-1*)
Returns the eigenvalues at the saddle point. Shape (2*nos)

`spirit.htst.get_eigenvectors_min`(*p_state*, *idx_chain=-1*)
Returns the eigenvectors at the minimum. Shape (2*nos*nos)

`spirit.htst.get_eigenvectors_sp`(*p_state*, *idx_chain=-1*)
Returns the eigenvectors at the saddle point. Shape (2*nos*nos)

`spirit.htst.get_info`(*p_state*, *idx_chain=-1*)
Returns a set of HTST information:

- the exponent of the temperature-dependence
- *me*
- *Omega_0*
- *s*
- zero mode volume at the minimum
- zero mode volume at the saddle point
- dynamical prefactor
- full rate prefactor (without temperature dependent part)

`spirit.htst.get_velocities`(*p_state*, *idx_chain=-1*)
Returns the velocities perpendicular to the dividing surface. Shape (2*nos)

10.7 I/O

Read and write spin configurations, chains or eigenmodes. Vectorfields are generally written in the OOMMF vector field (OVF) file format.

Note that, when reading an image or chain from file, the file will automatically be tested for an OVF header. If it cannot be identified as OVF, it will be tried to be read as three plain text columns (Sx Sy Sz).

Note also, IO is still being re-written and only OVF will be supported as output format.

`spirit.io.FILEFORMAT_OVF_BIN = 0`
OVF binary format corresponding to the precision with which the code was compiled

spirit.io.**FILEFORMAT_OVF_BIN4 = 1**
OVF binary format with precision 4

spirit.io.**FILEFORMAT_OVF_BIN8 = 2**
OVF binary format with precision 8

spirit.io.**FILEFORMAT_OVF_CSV = 4**
OVF text format with comma-separated columns

spirit.io.**FILEFORMAT_OVF_TEXT = 3**
OVF text format

spirit.io.**chain_append**(*p_state*, *filename*, *fileformat=3*, *comment=''*, *idx_chain=-1*)
Append a chain of images to a given file.

If the file does not exist, it is created.

Arguments: p_state – state pointer filename – the name of the file to append to

Keyword arguments: fileformat – the format in which to write the data (default: OVF text) comment – a comment string to be inserted in the header (default: empty)

spirit.io.**chain_read**(*p_state*, *filename*, *starting_image=0*, *ending_image=0*, *insert_idx=-1*, *idx_chain=-1*)
Attempt to read a chain of images from a given file.

Arguments: p_state – state pointer filename – the name of the file to read

Keyword arguments: starting_image – the index within the file at which to start reading (default: 0) ending_image – the index within the file at which to stop reading (default: 0) insert_idx – the index within the chain at which to start placing the images (default: active image)

Images of the chain will be overwritten with what is read from the file. If the chain is not long enough for the number of images to be read, it is automatically set to the right length.

spirit.io.**chain_write**(*p_state*, *filename*, *fileformat=3*, *comment=''*, *idx_chain=-1*)
Write a chain of images to a file.

Arguments: p_state – state pointer filename – the name of the file to write

Keyword arguments: fileformat – the format in which to write the data (default: OVF text) comment – a comment string to be inserted in the header (default: empty)

spirit.io.**eigenmodes_read**(*p_state*, *filename*, *fileformat=3*, *idx_image_inchain=-1*, *idx_chain=-1*)
Read the vector fields from a file as a set of eigenmodes for the spin system.

spirit.io.**eigenmodes_write**(*p_state*, *filename*, *fileformat=3*, *comment=' '*, *idx_image=-1*, *idx_chain=-1*)
Write the eigenmodes of a spin system to file, if they have been already calculated.

spirit.io.**image_append**(*p_state*, *filename*, *fileformat=3*, *comment=''*, *idx_image=-1*, *idx_chain=-1*)
Append an image of the chain to a file, incrementing the segment count.

If the file does not exist, it is created.

Arguments: p_state – state pointer filename – the name of the file to append to

Keyword arguments: fileformat – the format in which to write the data (default: OVF text) comment – a comment string to be inserted in the header (default: empty) idx_image – the index of the image to be written to the file (default: active image)

spirit.io.**image_read**(*p_state*, *filename*, *idx_image_infile=0*, *idx_image_inchain=-1*, *idx_chain=-1*)
Attempt to read a spin configuration from a file into an image of the chain.

Arguments: p_state – state pointer filename – the name of the file to read

Keyword arguments: idx_image_infile – the index of the image in the file which should be read in (default: 0) idx_image_inchain – the index of the image in the chain into which the data should be read (default: active image)

`spirit.io.image_write`(*p_state*, *filename*, *fileformat=3*, *comment=”*, *idx_image=-1*, *idx_chain=-1*)
Write an image of the chain to a file.

Arguments: p_state – state pointer filename – the name of the file to write

Keyword arguments: fileformat – the format in which to write the data (default: OVF text) comment – a comment string to be inserted in the header (default: empty) idx_image – the index of the image to be written to the file (default: active image)

`spirit.io.n_images_in_file`(*p_state*, *filename*, *idx_image_inchain=-1*, *idx_chain=-1*)
Returns the number of segments or images in a given file.

Arguments: p_state – state pointer filename – the name of the file to check

10.8 Log

`spirit.log.append`(*p_state*)
Force the appending of new messages to the log file.

`spirit.log.get_n_entries`(*p_state*)
Returns the number of Log entries.

`spirit.log.get_n_errors`(*p_state*)
Returns the number of error messages that have been logged.

`spirit.log.get_n_warnings`(*p_state*)
Returns the number of warning messages that have been logged.

`spirit.log.get_output_console_level`(*p_state*)
Returns the level up to which the Log is output to the console.

The return value will be one of the integers defined above.

`spirit.log.get_output_file_level`(*p_state*)
Returns the level up to which the Log is output to a file.

The return value will be one of the integers defined above.

`spirit.log.get_output_to_console`(*p_state*)
Returns a bool indicating whether the Log is output to the console.

`spirit.log.get_output_to_file`(*p_state*)
Returns a bool indicating whether the Log is output to a file.

`spirit.log.send`(*p_state*, *level*, *sender*, *message*, *idx_image=-1*, *idx_chain=-1*)
Add a message to the log.

- level: see integers defined above. The message may be printed to the console and/or written to the log file, depending on the current log parameters
- sender: see integers defined above. Used to distinguish context
- message: a string which to log
- idx_image: can be used to specify to which image the message relates (default: active image)

`spirit.log.set_output_file_tag(p_state, tag)`
Set the tagging string which is placed in front of the log file.

If "<time>" is used for the tag, it will be the time of creation of the state.

`spirit.log.set_output_folder(p_state, tag)`
Set the output folder in which to place the log file.

`spirit.log.set_output_to_console(p_state, output, level)`
Set whether the Log is output to the console and the level up to which messages are logged.

`spirit.log.set_output_to_file(p_state, output, level)`
Set whether the Log is output to a file and the level up to which messages are logged.

10.9 spirit.parameters

10.9.1 Monte Carlo (MC)

`spirit.parameters.mc.get_iterations(p_state, idx_image=-1, idx_chain=-1)`
Returns the maximum number of iterations and the step size.

`spirit.parameters.mc.get_metropolis_cone(p_state, idx_image=-1, idx_chain=-1)`
Returns the Metropolis algorithm configuration.

- whether the spins are displaced within a cone (otherwise: on the entire unit sphere)
- the opening angle within which the spin is placed
- whether the cone angle is automatically adapted to achieve the set acceptance ratio
- target acceptance ratio for the adaptive cone algorithm

`spirit.parameters.mc.get_temperature(p_state, idx_image=-1, idx_chain=-1)`
Returns the global base temperature [K].

`spirit.parameters.mc.set_iterations(p_state, n_iterations, n_iterations_log, idx_image=-1, idx_chain=-1)`
Set the number of iterations and how often to log and write output.

- n_iterations: the maximum number of iterations
- n_iterations_log: the number of iterations after which status is logged and output written

`spirit.parameters.mc.set_metropolis_cone(p_state, use_cone=True, cone_angle=40, use_adaptive_cone=True, target_acceptance_ratio=0.5, idx_image=-1, idx_chain=-1)`
Configure the Metropolis parameters.

- use_cone: whether to displace the spins within a cone (otherwise: on the entire unit sphere)
- cone_angle: the opening angle within which the spin is placed
- use_adaptive_cone: automatically adapt the cone angle to achieve the set acceptance ratio
- target_acceptance_ratio: target acceptance ratio for the adaptive cone algorithm

`spirit.parameters.mc.set_output_configuration(p_state, step=False, archive=True, filetype=3, idx_image=-1, idx_chain=-1)`
Set whether to write spin configuration output files.

- step: whether to write a new file after each set of iterations

- archive: whether to append to an archive file after each set of iterations
- filetype: the format in which the data is written

`spirit.parameters.mc.set_output_energy`(*p_state*, *step=False*, *archive=True*, *spin_resolved=False*, *divide_by_nos=True*, *add_readability_lines=True*, *idx_image=-1*, *idx_chain=-1*)

Set whether to write energy output files.

- step: whether to write a new file after each set of iterations
- archive: whether to append to an archive file after each set of iterations
- spin_resolved: whether to write a file containing the energy of each spin
- divide_by_nos: whether to divide energies by the number of spins
- add_readability_lines: whether to separate columns by lines

`spirit.parameters.mc.set_output_folder`(*p_state*, *folder*, *idx_image=-1*, *idx_chain=-1*)

Set the folder, where output files are placed.

`spirit.parameters.mc.set_output_general`(*p_state*, *any=True*, *initial=False*, *final=False*, *idx_image=-1*, *idx_chain=-1*)

Set whether to write any output files at all.

`spirit.parameters.mc.set_output_tag`(*p_state*, *tag*, *idx_image=-1*, *idx_chain=-1*)

Set the tag placed in front of output file names.

If the tag is "<time>", it will be the date-time of the creation of the state.

`spirit.parameters.mc.set_temperature`(*p_state*, *temperature*, *idx_image=-1*, *idx_chain=-1*)

Set the global base temperature [K].

10.9.2 Landau-Lifshitz-Gilbert (LLG)

`spirit.parameters.llg.get_convergence`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the convergence value.

`spirit.parameters.llg.get_damping`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the Gilbert damping parameter.

`spirit.parameters.llg.get_direct_minimization`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns whether only energy minimisation will be performed.

`spirit.parameters.llg.get_iterations`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the maximum number of iterations and the step size.

`spirit.parameters.llg.get_stt`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the spin current configuration.

- magnitude
- direction, array of shape (3)
- whether the spatial gradient is used

`spirit.parameters.llg.get_temperature`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the temperature configuration.

- global base temperature [K]
- inclination of the temperature gradient [K/a]

- direction of the temperature gradient, array of shape (3)

`spirit.parameters.llg.get_timestep(p_state, idx_image=-1, idx_chain=-1)`
Returns the time step [ps].

`spirit.parameters.llg.set_convergence(p_state, convergence, idx_image=-1, idx_chain=-1)`
Set the convergence limit.

When the maximum absolute component value of the force drops below this value, the calculation is considered converged and will stop.

`spirit.parameters.llg.set_damping(p_state, damping, idx_image=-1, idx_chain=-1)`
Set the Gilbert damping parameter [unitless].

`spirit.parameters.llg.set_direct_minimization(p_state, use_minimization, idx_image=-1, idx_chain=-1)`
Set whether to minimise the energy without precession.

This only influences dynamics solvers, which will then perform pseudodynamics, simulating only the damping part of the LLG equation.

`spirit.parameters.llg.set_iterations(p_state, n_iterations, n_iterations_log, idx_image=-1, idx_chain=-1)`
Set the number of iterations and how often to log and write output.

- n_iterations: the maximum number of iterations
- n_iterations_log: the number of iterations after which status is logged and output written

`spirit.parameters.llg.set_output_configuration(p_state, step=False, archive=True, filetype=3, idx_image=-1, idx_chain=-1)`
Set whether to write spin configuration output files.

- step: whether to write a new file after each set of iterations
- archive: whether to append to an archive file after each set of iterations
- filetype: the format in which the data is written

`spirit.parameters.llg.set_output_energy(p_state, step=False, archive=True, spin_resolved=False, divide_by_nos=True, add_readability_lines=True, idx_image=-1, idx_chain=-1)`
Set whether to write energy output files.

- step: whether to write a new file after each set of iterations
- archive: whether to append to an archive file after each set of iterations
- spin_resolved: whether to write a file containing the energy of each spin
- divide_by_nos: whether to divide energies by the number of spins
- add_readability_lines: whether to separate columns by lines

`spirit.parameters.llg.set_output_folder(p_state, folder, idx_image=-1, idx_chain=-1)`
Set the folder, where output files are placed.

`spirit.parameters.llg.set_output_general(p_state, any=True, initial=False, final=False, idx_image=-1, idx_chain=-1)`
Set whether to write any output files at all.

`spirit.parameters.llg.set_output_tag(p_state, tag, idx_image=-1, idx_chain=-1)`
Set the tag placed in front of output file names.

If the tag is "<time>", it will be the date-time of the creation of the state.

`spirit.parameters.llg.set_stt`(*p_state*, *use_gradient*, *magnitude*, *direction*, *idx_image=-1*, *idx_chain=-1*)

Set the spin current configuration.

- use_gradient: *True*: use the spatial gradient, *False*: monolayer approximation
- magnitude: current strength
- direction: current direction or polarisation direction, array of shape (3)

`spirit.parameters.llg.set_temperature`(*p_state, temperature, gradient_inclination=0, gradient_direction=[1.0, 0.0, 0.0], idx_image=-1, idx_chain=-1*)

Set the temperature configuration.

- temperature: homogeneous base temperature [K]
- gradient_inclination: inclination of the temperature gradient [K/a]
- gradient_direction: direction of the temperature gradient, array of shape (3)

`spirit.parameters.llg.set_timestep`(*p_state*, *dt*, *idx_image=-1*, *idx_chain=-1*)

Set the time step [ps] for the calculation.

10.9.3 Geodesic nudged elastic band (GNEB)

`spirit.parameters.gneb.IMAGE_CLIMBING = 1`

Climbing GNEB image type. Climbing images move towards maxima along the path.

`spirit.parameters.gneb.IMAGE_FALLING = 2`

Falling GNEB image type. Falling images move towards the closest minima.

`spirit.parameters.gneb.IMAGE_NORMAL = 0`

Regular GNEB image type.

`spirit.parameters.gneb.IMAGE_STATIONARY = 3`

Stationary GNEB image type. Stationary images are not influenced during a GNEB calculation.

`spirit.parameters.gneb.get_climbing_falling`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the integer of whether an image is regular, climbing, falling, or stationary.

The integers are defined above.

`spirit.parameters.gneb.get_convergence`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the convergence value.

`spirit.parameters.gneb.get_iterations`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the maximum number of iterations and the step size.

`spirit.parameters.gneb.get_n_energy_interpolations`(*p_state*, *idx_chain=-1*)

Returns the number of energy values interpolated between images.

`spirit.parameters.gneb.get_path_shortening_constant`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Return the path shortening constant.

`spirit.parameters.gneb.get_spring_force`(*p_state*, *idx_image=-1*, *idx_chain=-1*)

Returns the spring force constant and Ratio of energy to reaction coordinate.

`spirit.parameters.gneb.set_climbing_falling`(*p_state*, *image_type*, *idx_image=-1*, *idx_chain=-1*)

Set the GNEB image type (see the integers defined above).

`spirit.parameters.gneb.`**`set_convergence`**(*p_state*, *convergence*, *idx_image=-1*, *idx_chain=-1*)

Set the convergence limit.

When the maximum absolute component value of the force drops below this value, the calculation is considered converged and will stop.

`spirit.parameters.gneb.`**`set_image_type_automatically`**(*p_state*, *idx_chain=-1*)

Automatically set GNEB image types.

Minima along the path will be set to falling, maxima to climbing and the rest to regular.

`spirit.parameters.gneb.`**`set_iterations`**(*p_state*, *n_iterations*, *n_iterations_log*, *idx_image=-1*, *idx_chain=-1*)

Set the number of iterations and how often to log and write output.

- n_iterations: the maximum number of iterations
- n_iterations_log: the number of iterations after which status is logged and output written

`spirit.parameters.gneb.`**`set_output_chain`**(*p_state*, *step=False*, *filetype=3*, *idx_image=-1*, *idx_chain=-1*)

Set whether to write chain output files.

- step: whether to write a new file after each set of iterations
- filetype: the format in which the data is written

`spirit.parameters.gneb.`**`set_output_energies`**(*p_state*, *step=True*, *interpolated=True*, *divide_by_nos=True*, *add_readability_lines=True*, *idx_image=-1*, *idx_chain=-1*)

Set whether to write energy output files.

- step: whether to write a new file after each set of iterations
- interpolated: whether to write a file containing interpolated reaction coordinate and energy values
- divide_by_nos: whether to divide energies by the number of spins
- add_readability_lines: whether to separate columns by lines

`spirit.parameters.gneb.`**`set_output_folder`**(*p_state*, *folder*, *idx_image=-1*, *idx_chain=-1*)

Set the folder, where output files are placed.

`spirit.parameters.gneb.`**`set_output_general`**(*p_state*, *any=True*, *initial=False*, *final=False*, *idx_image=-1*, *idx_chain=-1*)

Set whether to write any output files at all.

`spirit.parameters.gneb.`**`set_output_tag`**(*p_state*, *tag*, *idx_image=-1*, *idx_chain=-1*)

Set the tag placed in front of output file names.

If the tag is "<time>", it will be the date-time of the creation of the state.

`spirit.parameters.gneb.`**`set_path_shortening_constant`**(*p_state*, *shortening_constant*, *idx_image=-1*, *idx_chain=-1*)

Set the path shortening constant.

`spirit.parameters.gneb.`**`set_spring_force`**(*p_state*, *spring_constant=1*, *ratio=0*, *idx_image=-1*, *idx_chain=-1*)

Set the spring force constant and the ratio between energy and reaction coordinate.

10.9.4 Eigenmode analysis (EMA)

This method, if needed, calculates modes (they can also be read in from a file) and perturbs the spin system periodically in the direction of the eigenmode.

`spirit.parameters.ema.`**`get_n_mode_follow`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the index of the mode to use.

`spirit.parameters.ema.`**`get_n_modes`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the number of modes to calculate or use.

`spirit.parameters.ema.`**`set_n_mode_follow`**(*p_state*, *n_mode*, *idx_image=-1*, *idx_chain=-1*)
Set the index of the mode to use.

`spirit.parameters.ema.`**`set_n_modes`**(*p_state*, *n_modes*, *idx_image=-1*, *idx_chain=-1*)
Set the number of modes to calculate or use.

10.9.5 Minimum mode following (MMF)

`spirit.parameters.mmf.`**`get_iterations`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the maximum number of iterations and the step size.

`spirit.parameters.mmf.`**`get_n_mode_follow`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the index of the mode which to follow.

`spirit.parameters.mmf.`**`get_n_modes`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the number of modes calculated at each iteration.

`spirit.parameters.mmf.`**`set_iterations`**(*p_state*, *n_iterations*, *n_iterations_log*, *idx_image=-1*, *idx_chain=-1*)
Set the number of iterations and how often to log and write output.

- n_iterations: the maximum number of iterations
- n_iterations_log: the number of iterations after which status is logged and output written

`spirit.parameters.mmf.`**`set_n_mode_follow`**(*p_state*, *n_mode*, *idx_image=-1*, *idx_chain=-1*)
Set the index of the mode to follow.

`spirit.parameters.mmf.`**`set_n_modes`**(*p_state*, *n_modes*, *idx_image=-1*, *idx_chain=-1*)
Set the number of modes to be calculated at each iteration.

`spirit.parameters.mmf.`**`set_output_configuration`**(*p_state*, *step=False*, *archive=True*, *filetype=3*, *idx_image=-1*, *idx_chain=-1*)
Set whether to write spin configuration output files.

- step: whether to write a new file after each set of iterations
- archive: whether to append to an archive file after each set of iterations
- filetype: the format in which the data is written

`spirit.parameters.mmf.`**`set_output_energy`**(*p_state*, *step=False*, *archive=True*, *spin_resolved=False*, *divide_by_nos=True*, *add_readability_lines=True*, *idx_image=-1*, *idx_chain=-1*)
Set whether to write energy output files.

- step: whether to write a new file after each set of iterations
- archive: whether to append to an archive file after each set of iterations
- spin_resolved: whether to write a file containing the energy of each spin

- divide_by_nos: whether to divide energies by the number of spins
- add_readability_lines: whether to separate columns by lines

`spirit.parameters.mmf.`**`set_output_folder`**(*p_state*, *folder*, *idx_image=-1*, *idx_chain=-1*)
Set the folder, where output files are placed.

`spirit.parameters.mmf.`**`set_output_general`**(*p_state*, *any=True*, *initial=False*, *final=False*, *idx_image=-1*, *idx_chain=-1*)
Set whether to write any output files at all.

`spirit.parameters.mmf.`**`set_output_tag`**(*p_state*, *tag*, *idx_image=-1*, *idx_chain=-1*)
Set the tag placed in front of output file names.

If the tag is "<time>", it will be the date-time of the creation of the state.

10.10 Quantities

`spirit.quantities.`**`get_magnetization`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Calculates and returns the average magnetization of the system as an array of shape (3).

`spirit.quantities.`**`get_mmf_info`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns a set of MMF information, meant mostly for testing or debugging.

- numpy.array_view of shape (NOS, 3) of the energy gradient
- the lowest eigenvalue
- numpy.array_view of shape (NOS, 3) of the eigenmode
- numpy.array_view of shape (NOS, 3) of the force

`spirit.quantities.`**`get_topological_charge`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Calculates and returns the total topological charge of 2D systems.

Note that the charge can take unphysical non-integer values for open boundaries, because it is not well-defined in this case.

Returns 0 for systems of other dimensionality.

10.11 Simulation

This module of Spirit is used to run and monitor iterative calculation methods.

If many iterations are called individually, one should use the single shot simulation functionality. It avoids the allocations etc. involved when a simulation is started and ended and behaves like a regular simulation, except that the iterations have to be triggered manually.

Note that the VP and NCG Solvers are only meant for direct minimization and not for dynamics.

`spirit.simulation.`**`METHOD_EMA = 4`**
Eigenmode analysis.

Applies eigenmodes to the spins of a system. Depending on parameters, this can be used to calculate the change of a spin configuration through such a mode or to get a "dynamical" chain of images corresponding to the movement of the system under the mode.

`spirit.simulation.`**`METHOD_GNEB = 2`**
Geodesic nudged elastic band.

Runs on the entire chain.

As this is a minimisation method, the dynamical solvers perform worse than those designed for minimisation.

`spirit.simulation.`**`METHOD_LLG = 1`**
Landau-Lifshitz-Gilbert.

Can be either a dynamical simulation or an energy minimisation. Note: the VP solver can *only* minimise.

`spirit.simulation.`**`METHOD_MC = 0`**
Monte Carlo.

Standard implementation.

`spirit.simulation.`**`METHOD_MMF = 3`**
Minimum mode following.

As this is a minimisation method, the dynamical solvers perform worse than those designed for minimisation.

`spirit.simulation.`**`SOLVER_DEPONDT = 2`**
Depondt's Heun-like method.

`spirit.simulation.`**`SOLVER_HEUN = 3`**
Heun's method.

`spirit.simulation.`**`SOLVER_RK4 = 4`**
4th order Runge-Kutta method.

`spirit.simulation.`**`SOLVER_SIB = 1`**
Semi-implicit midpoint method B.

`spirit.simulation.`**`SOLVER_VP = 0`**
Verlet-like velocity projection method.

`spirit.simulation.`**`get_iterations_per_second`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the current estimation of the number of iterations per second.

`spirit.simulation.`**`running_anywhere_on_chain`**(*p_state*, *idx_chain=-1*)
Check if any simulation running on any image of - or the entire - chain.

`spirit.simulation.`**`running_on_chain`**(*p_state*, *idx_chain=-1*)
Check if a simulation is running across a specific chain.

`spirit.simulation.`**`running_on_image`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Check if a simulation is running on a specific image.

`spirit.simulation.`**`single_shot`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Perform a single iteration.

In order to use this, a single shot simulation must be running on the corresponding image or chain.

`spirit.simulation.`**`start`**(*p_state*, *method_type*, *solver_type=None*, *n_iterations=-1*, *n_iterations_log=-1*, *single_shot=False*, *idx_image=-1*, *idx_chain=-1*)
Start any kind of iterative calculation method.

- method_type: one of the integers defined above
- solver_type: only used for LLG, GNEB and MMF methods (default: None)
- n_iterations: the maximum number of iterations that will be performed (default: take from parameters)

- n_iterations_log: the number of iterations after which to log the status and write output (default: take from parameters)
- single_shot: if set to *True*, iterations have to be triggered individually
- idx_image: the image on which to run the calculation (default: active image). Not used for GNEB

`spirit.simulation.stop(p_state, idx_image=-1, idx_chain=-1)`
Stop the simulation running on an image or chain.

`spirit.simulation.stop_all(p_state)`
Stop all simulations running anywhere.

10.12 State

The state contains the chain of spin systems (also called images).

To create a new state with containing a single image and run a simple example of a **spin dynamics simulation**:

or call setup and delete manually:

You can pass an input file specifying your initial system parameters. If you do not pass an input file, the implemented defaults are used.

class `spirit.state.State(configfile=", quiet=False)`
Bases: `object`

Wrapper Class for a Spirit State.

Can be used as *with spirit.state.State() as p_state:*

__dict__ = mappingproxy({'__module__': 'spirit.state', '__doc__': 'Wrapper Class for

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

__init__(*configfile=", quiet=False*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'spirit.state'

__weakref__
list of weak references to the object (if defined)

`spirit.state.date_time(p_state)`
Returns a string containing the date-time of the creation of the state.

`spirit.state.delete(p_state)`

`spirit.state.setup(configfile=", quiet=False)`

`spirit.state.to_config(p_state, filename, comment=")`
Write a config (input) file corresponding to the current parameters etc. of the state.

10.13 System

`spirit.system.get_effective_field(p_state, idx_image=-1, idx_chain=-1)`

`spirit.system.get_eigenmode(p_state, idx_mode, idx_image=-1, idx_chain=-1)`

`spirit.system.get_eigenvalues(p_state, idx_image=-1, idx_chain=-1)`

`spirit.system.`**`get_energy`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Calculates and returns the energy of the system.

`spirit.system.`**`get_index`**(*p_state*)
Returns the index of the currently active image.

`spirit.system.`**`get_nos`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns the number of spins (NOS).

`spirit.system.`**`get_spin_directions`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Returns an *numpy.array_view* of shape (NOS, 3) with the components of each spins orientation vector.

Changing the contents of this array_view will have direct effect on calculations etc.

`spirit.system.`**`print_energy_array`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Print the energy array of the state to the console.

`spirit.system.`**`update_data`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
TODO: document when this needs to be called.

`spirit.system.`**`update_eigenmodes`**(*p_state*, *idx_image=-1*, *idx_chain=-1*)
Calculates eigenmodes of a system according to EMA parameters. This needs to be called or eigenmodes need to be read in before they can be used by other functions (e.g. writing them to a file).

10.14 Transition

`spirit.transition.`**`add_noise`**(*p_state*, *temperature*, *idx_1*, *idx_2*, *idx_chain=-1*)
Add some temperature-scaled noise to a transition between two images of a chain.

`spirit.transition.`**`homogeneous`**(*p_state*, *idx_1*, *idx_2*, *idx_chain=-1*)
Generate homogeneous transition between two images of a chain.

CHAPTER 11

Contributing

Contributions are always welcome! See also the current *list of contributors*.

1. Fork this repository
2. Check out the develop branch: `git checkout develop`
3. Create your feature branch: `git checkout -b feature-something`
4. Commit your changes: `git commit -am 'Add some feature'`
5. Push to the branch: `git push origin feature-something`
6. Submit a pull request

Please keep your pull requests *feature-specific* and limit yourself to one feature per feature branch. Remember to pull updates from this repository before opening a new feature branch.

If you are unsure where to add you feature into the code, please do not hesitate to contact us.

There is no strict coding guideline, but please try to match your code style to the code you edited or to the style in the respective module.

11.1 *Branches*

We aim to adhere to the "git flow" branching model: http://nvie.com/posts/a-successful-git-branching-model/

> Release versions (`master` branch) are tagged `major.minor.patch`, starting at `1.0.0`

Download the latest stable version from https://github.com/spirit-code/spirit/releases

The develop branch contains the latest updates, but is generally less consistently tested than the releases.

CHAPTER 12

Contributors

12.1 Gideon P. Müller

- RWTH Aachen
- University of Iceland
- PGI-1/IAS-1 at Forschungszentrum Jülich

General code design and project setup (including CMake). Implementation of the core library and user interfaces, most notably:

- GNEB and MMF methods
- Velocity projection solver
- CUDA and OpenMP parallelizations of backend
- C API and Python bindings
- C++ QT GUI and initial OpenGL code
- Unit tests and continuous integration

email: g.mueller@fz-juelich.de

(Oct. 2014 - ongoing)

12.2 Moritz Sallermann

- RWTH Aachen
- PGI-1/IAS-1 at Forschungszentrum Jülich

Implementation of the dipole-dipole interaction using FFT convolutions.

email: m.sallermann@fz-juelich.de

(Apr. 2015 - Sept. 2016)

12.3 Markus Hoffmann

- RWTH Aachen
- PGI-1/IAS-1 at Forschungszentrum Jülich

Bug-reports, feedback on code features and general help designing some of the functionality, user interface and input file format.

email: m.hoffmann@fz-juelich.de

(Jun. 2016 - ongoing)

12.4 Nikolai S. Kiselev

- PGI-1/IAS-1 at Forschungszentrum Jülich

Scientific advice, general help and feedback, initial (Fortran90) implementations of:

- isotropic Heisenberg Hamiltonian
- Neighbour calculations
- SIB solver
- Monte Carlo methods

email: n.kiselev@fz-juelich.de

(2007 - ongoing)

12.5 Florian Rhiem

- Scientific IT-Systems, PGI/JCNS at Forschungszentrum Jülich

Implementation of C++ OpenGL code (VFRendering library), as well as JavaScript Web UI and WebGL code. Code design improvements, including the C API and CMake.

(Jan. 2016 - ongoing)

12.6 Pavel F. Bessarab

- University of Iceland

Help with the initial GNEB implementation. Initial (Fortran90) implementation of the HTST method.

email: bessarab@hi.is

(Apr. 2015 - ongoing)

12.7 Daniel Schürhoff

- RWTH Aachen
- PGI-1/IAS-1 at Forschungszentrum Jülich

Implementation of the initial core library, notably translating from Fortran90 to C++ and addition of STT to the SIB solver. Work on QT GUI and Python bindings.

(Oct. 2015 - Sept. 2016)

12.8 Stefanos Mavros

- RWTH Aachen

Work on unit testing and documentation, implementation of the Depondt solver. Also some general code design and IO improvements.

(Apr. 2017 - Oct. 2018)

12.9 Constantin Disselkamp

- RWTH Aachen

Implementation and testing of gradient approximation of spin transfer torque.

(Apr. 2017 - Jul. 2017)

12.10 Filipp N. R. Rybakov

- Various Universities

Designs and ideas for the user interface and other code features, such as isosurfaces and colormaps for 3D systems. Some help and ideas related to code performance and CUDA.

(Jan. 2016 - ongoing)

12.11 Ingo Heimbach

- Scientific IT-Systems, PGI/JCNS at Forschungszentrum Jülich

Implementation of the initial OpenGL code. Code design suggestions and other general help.

(Jan. 2016 - ongoing)

12.12 Mathias Redies, Maximilian Merte, Rene Suckert

- RWTH Aachen

Initial CUDA implementation and tests. Code optimizations, suggestions and feedback.

(Sept. 2016 - Dec. 2016)

12.13 David Bauer

- RWTH Aachen
- PGI-1/IAS-1 at Forschungszentrum Jülich

Initial (Fortran90) implementations of the isotropic Heisenberg Hamiltonian, Neighbour calculations and the SIB solver.

(Oct. 2007 - Sept. 2008)

12.14 Graph

You may also take a look at the contributors graph.

CHAPTER 13

Reference

The **Spirit** framework is a scientific project. If you present and/or publish scientific results or visualisations that used Spirit, you should add a reference.

13.1 The Framework

If you used Spirit to produce scientific results or when referring to it as a scientific project, please cite the paper.

```
\bibitem{mueller_spirit_2019}{
    G. P. Müller, M. Hoffmann, C. Disselkamp, D. Schürhoff, S. Mavros, M. Sallermann,␣
↪N. S. Kiselev, H. Jónsson, S. Blügel.
    "Spirit: Multifunctional framework for atomistic spin simulations."
    Phys. Rev. B \textbf{99}, 224414 (2019)
}
```

When referring to code of this framework please add a reference to our GitHub page. You may use e.g. the following TeX code:

```
\bibitem{spirit}
{Spirit spin simulation framework} (see spirit-code.github.io)
```

13.2 Specific Methods

The following need only be cited if used.

Depondt Solver

This Heun-like method for solving the LLG equation including the stochastic term has been published by Depondt et al.: http://iopscience.iop.org/0953-8984/21/33/336005 You may use e.g. the following TeX code:

```
\bibitem{Depondt}
Ph. Depondt et al. \textit{J. Phys. Condens. Matter} \textbf{21}, 336005 (2009).
```

SIB Solver

This stable method for solving the LLG equation efficiently and including the stochastic term has been published by Mentink et al.: http://iopscience.iop.org/0953-8984/22/17/176001 You may use e.g. the following TeX code:

```
\bibitem{SIB}
J. H. Mentink et al. \textit{J. Phys. Condens. Matter} \textbf{22}, 176001 (2010).
```

VP Solver

This intuitive direct minimization routine has been published as supplementary material by Bessarab et al.: http://www.sciencedirect.com/science/article/pii/S0010465515002696 You may use e.g. the following TeX code:

```
\bibitem{VP}
P. F. Bessarab et al. \textit{Comp. Phys. Comm.} \textbf{196}, 335 (2015).
```

GNEB Method

This specialized nudged elastic band method for calculating transition paths of spin systems has been published by Bessarab et al.: http://www.sciencedirect.com/science/article/pii/S0010465515002696 You may use e.g. the following TeX code:

```
\bibitem{GNEB}
P. F. Bessarab et al. \textit{Comp. Phys. Comm.} \textbf{196}, 335 (2015).
```

HTST

The harmonic transition state theory for calculating transition rates of spin systems has been published by Bessarab et al.: https://link.aps.org/doi/10.1103/PhysRevB.85.184409 You may use e.g. the following TeX code:

```
\bibitem{GNEB}
P. F. Bessarab et al. \textit{Comp. Phys. Comm.} \textbf{196}, 335 (2015).
```

MMF Method

The mode following method, intended for saddle point searches, has been published by Müller et al.: https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.121.197202 You may use e.g. the following TeX code:

```
\bibitem{MMF}
G. P. Müller et al. Phys. Rev. Lett. 121, 197202 (2018).
```

Home

CHAPTER 14

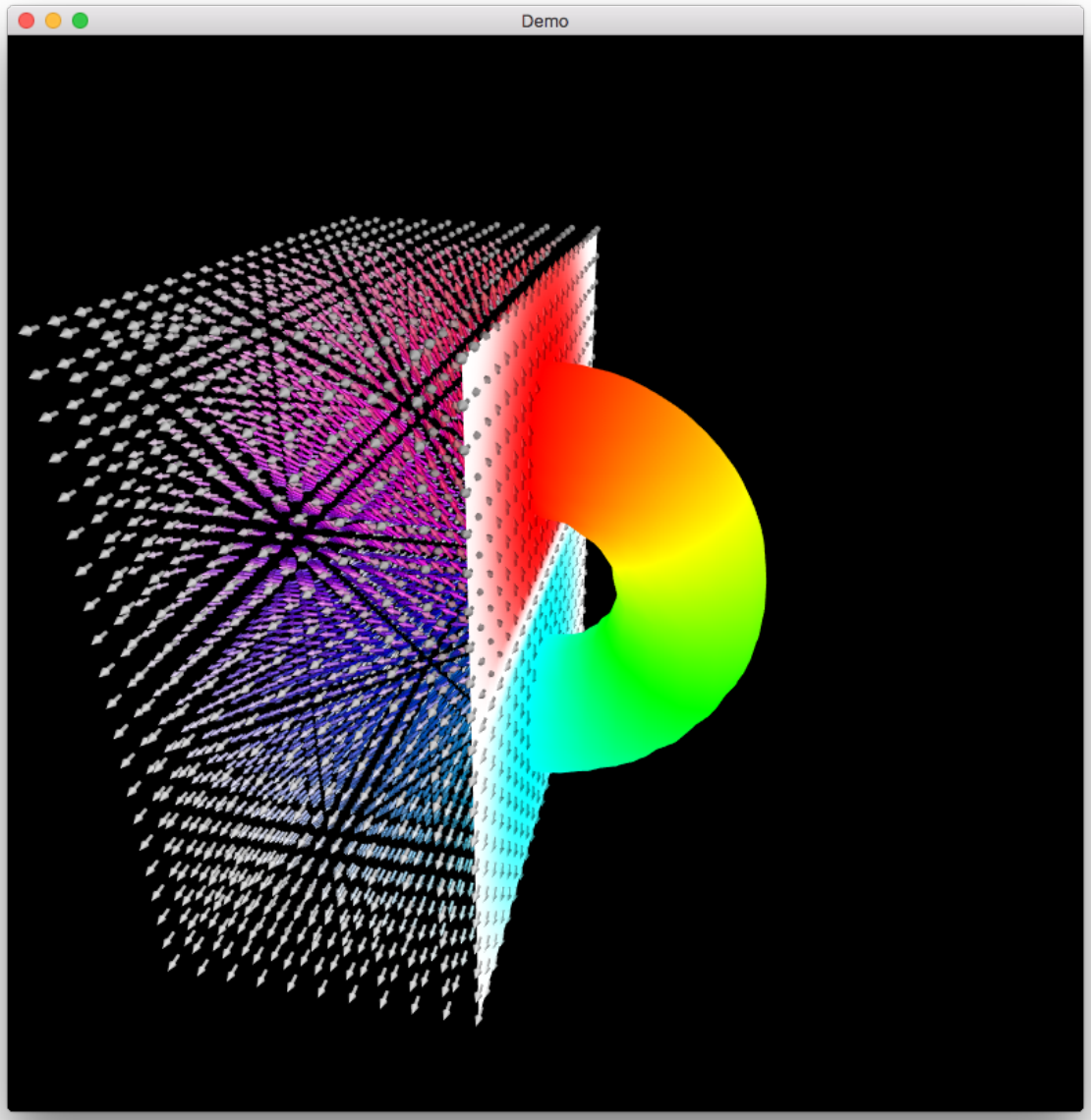
Included Dependencies

14.1 Vector Field Rendering

libvfrendering is a C++ library for rendering vectorfields using OpenGL. Originally developed for spirit and based on WegGLSpins.js, it has an extendable architecture and currently offers renderer implementations for:

- glyph-based vector field representations as arrows
- colormapped surface and isosurface rendering
- mapping vector directions onto a sphere

The library is still very much a work-in-progress, so its API is not yet stable and there are still several features missing that will be added in later releases. If you miss a feature or have another idea on how to improve libvfrendering, please open an issue or pull request!



Demo

14.1.1 Getting Started

To use **libvfrendering**, you need to perform the following steps:

1. Include <VFRendering/View.hxx>
2. Create a *VFRendering::Geometry*
3. Read or calculate the vector directions
4. Pass geometry and directions to a *VFRendering::View*
5. Draw the view in an existing OpenGL context

1. Include <VFRendering/View.hxx>

When using **libvfrendering**, you will mostly interact with View objects, so it should be enough to `#include <VFRendering/View.hxx>`.

2. Create a VFRendering::Geometry

The **geometry describes the positions** on which you evaluated the vector field and how they might form a grid (optional, e.g. for isosurface and surface rendering). You can pass the positions directly to the constructor or call one of the class' static methods.

As an example, this is how you could create a simple, cartesian 30x30x30 geometry, with coordinates between -1 and 1:

```
auto geometry = VFRendering::Geometry::cartesianGeometry(
    {30, 30, 30},
    {-1.0, -1.0, -1.0},
    {1.0, 1.0, 1.0}
);
```

3. Read or calculate the vector directions

This step highly depends on your use case. The **directions are stored as a `std::vector<glm::vec3>`**, so they can be created in a simple loop:

```
std::vector<glm::vec3> directions;
for (int iz = 0; iz < 10; iz++) {
    for (int iy = 0; iy < 10; iy++) {
        for (int ix = 0; ix < 10; ix++) {
            // calculate direction for ix, iy, iz
            directions.push_back(glm::normalize({ix-4.5, iy-4.5, iz-4.5}));
        }
    }
}
```

As shown here, the directions should be in **C order** when using the `VFRendering::Geometry` static methods. If you do not know glm, think of a `glm::vec3` as a struct containing three floats x, y and z.

4. Create a VFRendering::VectorField

This class simply contains geometry and directions.

```
VFRendering::VectorField vf(geometry, directions);
```

To update the VectorField data, use `VectorField::update`. If the directions changed but the geometry is the same, you can use the `VectorField::updateVectors` method or `VectorField::updateGeometry` vice versa.

5. Create a VFRendering::View and a Renderer

The view object is what you will interact most with. It provides an interface to the various renderers and includes functions for handling mouse input.

You can **create a new view** and then **initialize the renderer(s)** (as an example, we use the `VFRendering::ArrowRenderer`):

```
VFRendering::View view;
auto arrow_renderer_ptr = std::make_shared<VFRendering::ArrowRenderer>(view, vf);
view.renderers( {{ arrow_renderer_ptr, {0, 0, 1, 1} }} );
```

5. Draw the view in an existing OpenGL context

To actually see something, you need to create an OpenGL context using a toolkit of your choice, e.g. Qt or GLFW. After creating the context, pass the framebuffer size to the **setFramebufferSize method**. You can then call the **draw method** of the view to render the vector field, either in a loop or only when you update the data.

```
view.draw();
```

For a complete example, including an interactive camera, see demo.cxx.

14.1.2 Python Package

The Python package has bindings which correspond directly to the C++ class and function names. To use **pyVFRendering**, you need to perform the following steps:

1. `import pyVFRendering as vfr`
2. Create a `vfr.Geometry`
3. Read or calculate the vector directions
4. Pass geometry and directions to a `vfr.View`
5. Draw the view in an existing OpenGL context

1. import

In order to `import pyVFRendering as vfr`, you can either `pip install pyVFRendering` or download and build it yourself.

You can build with `python3 setup.py build`, which will generate a library somewhere in your `build` subfolder, which you can `import` in python. Note that you may need to add the folder to your `PYTHONPATH`.

2. Create a pyVFRendering.Geometry

As above:

```
geometry = vfr.Geometry.cartesianGeometry(
    (30, 30, 30),        # number of lattice points
    (-1.0, -1.0, -1.0), # lower bound
    (1.0, 1.0, 1.0) )    # upper bound
```

3. Read or calculate the vector directions

This step highly depends on your use case. Example:

```
directions = []
for iz in range(n_cells[2]):
    for iy in range(n_cells[1]):
        for ix in range(n_cells[0]):
            # calculate direction for ix, iy, iz
            directions.append( [ix-4.5, iy-4.5, iz-4.5] )
```

4. Pass geometry and directions to a pyVFRendering.View

You can **create a new view** and then **pass the geometry and directions by calling the update method**:

```
view = vfr.View()
view.update(geometry, directions)
```

If the directions changed but the geometry is the same, you can use the **updateVectors method**.

5. Draw the view in an existing OpenGL context

To actually see something, you need to create an OpenGL context using a framework of your choice, e.g. Qt or GLFW. After creating the context, pass the framebuffer size to the **setFramebufferSize method**. You can then call the **draw method** of the view to render the vector field, either in a loop or only when you update the data.

```
view.setFramebufferSize(width*self.window().devicePixelRatio(), height*self.window().
↪devicePixelRatio())
view.draw()
```

For a complete example, including an interactive camera, see demo.py.

14.1.3 Renderers

libvfrendering offers several types of renderers, which all inherit from `VFRendering::RendererBase`. The most relevant are the `VectorFieldRenderers`:

- VFRendering::ArrowRenderer, which renders the vectors as colored arrows
- VFRendering::SphereRenderer, which renders the vectors as colored spheres
- VFRendering::SurfaceRenderer, which renders the surface of the geometry using a colormap
- VFRendering::IsosurfaceRenderer, which renders an isosurface of the vectorfield using a colormap
- VFRendering::VectorSphereRenderer, which renders the vectors as dots on a sphere, with the position of each dot representing the direction of the vector

In addition to these, there also the following renderers which do not require a `VectorField`:

- VFRendering::CombinedRenderer, which can be used to create a combination of several renderers, like an isosurface rendering with arrows
- VFRendering::BoundingBoxRenderer, which is used for rendering bounding boxes around the geometry rendered by an VFRendering::ArrorRenderer, VFRendering::SurfaceRenderer or VFRendering::IsosurfaceRenderer
- VFRendering::CoordinateSystemRenderer, which is used for rendering a coordinate system, with the axes colored by using the colormap

To control what renderers are used, you can use `VFRendering::View::renderers`, where you can pass it a `std::vectors` of `std::pairs` of renderers as `std::shared_ptr<VFRendering::RendererBase>` (i.e. shared pointers) and viewports as `glm::vec4`.

14.1.4 Options

To modify the way the vector field is rendered, **libvfrendering** offers a variety of options. To set these, you can create an **VFRendering::Options** object.

As an example, to adjust the vertical field of view, you would do the following:

```
VFRendering::Options options;
options.set<VFRendering::View::Option::VERTICAL_FIELD_OF_VIEW>(30);
view.updateOptions(options);
```

If you want to set only one option, you can also use **View::setOption**:

```
view.setOption<VFRendering::View::Option::VERTICAL_FIELD_OF_VIEW>(30);
```

If you want to set an option for an individual Renderer, you can use the methods **RendererBase::updateOptions** and **RendererBase::setOption** in the same way.

Whether this way of setting options should be replaced by getters/setters will be evaluated as the API becomes more stable.

Currently, the following options are available:

14.1.5 ToDo

- A **EGS plugin** for combining **libvfrendering** with existing **EGS** plugins.
- Methods for reading geometry and directions from data files
- Documentation

See the issues for further information and adding your own requests.

14.2 OVF Parser Library

Simple API for powerful OOMMF Vector Field file parsing

OVF format specification

Build Status Build status

Python package: PyPI version

14.2.1 How to use

For usage examples, take a look into the test folders: test, python/test or fortran/test.

Except for opening a file or initializing a segment, all functions return status codes (generally `OVF_OK`, `OVF_INVALID` or `OVF_ERROR`). When the return code is not `OVF_OK`, you can take a look into the latest message, which should tell you what the problem was (`const char * ovf_latest_message(struct ovf_file *)` in the C API).

In C/C++ and Fortran, before writing a segment, make sure the `ovf_segment` you pass in is initialized, i.e. you already called either `ovf_read_segment_header` or `ovf_segment_create`.

C/C++

Opening and closing:

- `struct ovf_file *myfile = ovf_open("myfilename.ovf")` to open a file
- `myfile->found` to check if the file exists on disk
- `myfile->is_ovf` to check if the file contains an OVF header
- `myfile->n_segments` to check the number of segments the file should contain
- `ovf_close(myfile);` to close the file and free resources

Reading from a file:

- `struct ovf_segment *segment = ovf_segment_create()` to initialize a new segment and get the pointer
- `ovf_read_segment_header(myfile, index, segment)` to read the header into the segment struct
- create float data array of appropriate size…
- `ovf_read_segment_data_4(myfile, index, segment, data)` to read the segment data into your float array
- setting `segment->N` before reading allows partial reading of large data segments

Writing and appending to a file:

- `struct ovf_segment *segment = ovf_segment_create()` to initialize a new segment and get the pointer
- `segment->n_cells[0] = ...` etc to set data dimensions, title and description, etc.
- `ovf_write_segment_4(myfile, segment, data, OVF_FORMAT_TEXT)` to write a file containing the segment header and data
- `ovf_append_segment_4(myfile, segment, data, OVF_FORMAT_TEXT)` to append the segment header and data to the file

Python

To install the *ovf python package*, either build and install from source or simply use

```
pip install ovf
```

To use `ovf` from Python, e.g.

```
from ovf import ovf
import numpy as np

data = np.zeros((2, 2, 1, 3), dtype='f')
data[0,1,0,:] = [3.0, 2.0, 1.0]

with ovf.ovf_file("out.ovf") as ovf_file:

    # Write one segment
```

(continues on next page)

(continued from previous page)

```
    segment = ovf.ovf_segment(n_cells=[2,2,1])
    if ovf_file.write_segment(segment, data) != -1:
        print("write_segment failed: ", ovf_file.get_latest_message())

    # Add a second segment to the same file
    data[0,1,0,:] = [4.0, 5.0, 6.0]
    if ovf_file.append_segment(segment, data) != -1:
        print("append_segment failed: ", ovf_file.get_latest_message())
```

Fortran

The Fortran bindings are written in object-oriented style for ease of use. Writing a file, for example:

```
type(ovf_file)      :: file
type(ovf_segment)   :: segment
integer             :: success
real(kind=4), allocatable :: array_4(:,:)
real(kind=8), allocatable :: array_8(:,:)

! Initialize segment
call segment%initialize()

! Write a file
call file%open_file("fortran/test/testfile_f.ovf")
segment%N_Cells = [ 2, 2, 1 ]
segment%N = product(segment%N_Cells)

allocate( array_4(3, segment%N) )
array_4 = 0
array_4(:,1) = [ 6.0, 7.0, 8.0 ]
array_4(:,2) = [ 5.0, 4.0, 3.0 ]

success = file%write_segment(segment, array_4, OVF_FORMAT_TEXT)
if ( success == OVF_OK) then
    write (*,*) "test write_segment succeeded."
    ! write (*,*) "n_cells = ", segment%N_Cells
    ! write (*,*) "n_total = ", segment%N
else
    write (*,*) "test write_segment did not work. Message: ", file%latest_message
    STOP 1
endif
```

For more information on how to generate modern Fortran bindings, see also https://github.com/MRedies/Interfacing-Fortran

14.2.2 How to embed it into your project

TODO…

14.2.3 Build

On Unix systems

Usually:

```
mkdir build
cd build
cmake ..
make
```

On Windows

One possibility:

- open the folder in the CMake GUI
- generate the VS project
- open the resulting project in VS and build it

CMake Options

The following options are `ON` by default. If you want to switch them off, just pass `-D<OPTION>=OFF` to CMake, e.g. `-DOVF_BUILD_FORTRAN_BINDINGS=OFF`.

- `OVF_BUILD_PYTHON_BINDINGS`
- `OVF_BUILD_FORTRAN_BINDINGS`
- `OVF_BUILD_TEST`

On Windows, you can also set these from the CMake GUI.

Create and install the Python package

Instead of `pip`-installing it, you can e.g. build everything and then install the package locally, where the `-e` flag will let you change/update the package without having to re-install it.

```
cd python
pip install -e .
```

Build without CMake

The following is an example of how to manually build the C library and link it with bindings into a corresponding Fortran executable, using gcc.

C library:

```
g++ -DFMT_HEADER_ONLY -Iinclude -fPIC -std=c++11 -c src/ovf.cpp -o ovf.cpp.o

# static
ar qc libovf_static.a ovf.cpp.o
ranlib libovf_static.a

# shared
g++ -fPIC -shared -lc++ ovf.cpp.o -o libovf_shared.so
```

C/C++ test executable:

```
g++ -Iinclude -Itest -std=c++11 -c test/main.cpp -o main.cpp.o
g++ -Iinclude -Itest -std=c++11 -c test/simple.cpp -o simple.cpp.o

# link static lib
g++ -lc++ libovf_static.a main.cpp.o simple.cpp.o -o test_cpp_simple

# link shared lib
g++ libovf_shared.so main.cpp.o simple.cpp.o -o test_cpp_simple
```

Fortran library:

```
gfortran -fPIC -c fortran/ovf.f90 -o ovf.f90.o

ar qc libovf_fortran.a libovf_static.a ovf.f90.o
ranlib libovf_fortran.a
```

Fortran test executable

```
gfortran -c fortran/test/simple.f90 -o simple.f90.o
gfortran -lc++ libovf_fortran.a simple.f90.o -o test_fortran_simple
```

When linking statically, you can also link the object file `ovf.cpp.o` instead of `libovf_static.a`.

Note: depending on compiler and/or system, you may need `-lstdc++` instead of `-lc++`.

14.2.4 File format v2.0 specification

This specification is written according to the NIST user guide for OOMMF and has been implemented, but not tested or verified against OOMMF.

Note: The OVF 2.0 format is a modification to the OVF 1.0 format that also supports fields across three spatial dimensions but having values of arbitrary (but fixed) dimension. The following is a full specification of the 2.0 format.

General

- An OVF file has an ASCII header and trailer, and data blocks that may be either ASCII or binary.
- All non-data lines begin with a `#` character
- Comments start with `##` and are ignored by the parser. A comment continues until the end of the line.
- There is no line continuation character
- Lines starting with a `#` but containing only whitespace are ignored
- Lines starting with a `#` but containing an unknown keyword are are an error

After an overall header, the file consists of segment blocks, each composed of a segment header, data block and trailer.

- The field domain (i.e., the spatial extent) lies across three dimensions, with units typically expressed in meters or nanometers
- The field can be of any arbitrary dimension `N > 0` (This dimension, however, is fixed within each segment).

Header

- The first line of an OVF 2.0 file must be `# OOMMF OVF 2.0`
- The header should also contain the number of segments, specified as e.g. `# Segment count:  000001`
- Zero-padding of the segment count is not specified

Segments

Segment Header

- Each block begins with a `# Begin:  <block type>` line, and ends with a corresponding `# End: <block type>` line
- A non-empty non-comment line consists of a keyword and a value:
  - A keyword consists of all characters after the initial `#` up to the first colon (`:`) character. Case is ignored, and all whitespace is removed
  - Unknown keywords are errors
  - The value consists of all characters after the first colon (`:`) up to a comment (`##`) or line ending
- The order of keywords is not specified
- None of the keywords have default values, so all are required unless stated otherwise

Everything inside the `Header` block should be either comments or one of the following file keyword lines

- `title`: long file name or title
- `desc` (optional): description line, use as many as desired
- `meshunit`: fundamental mesh spatial unit. The comment marker `##` is not allowed in this line. Example value: `nm`
- `valueunits`: should be a (Tcl) list of value units. The comment marker `##` is not allowed in this line. Example value: `"kA/m"`. The length of the list should be one of
  - `N`: each element denotes the units for the corresponding dimension index
  - `1`: the single element is applied to all dimension indexes
- `valuelabels`: This should be a `N`-item (Tcl) list of value labels, one for each value dimension. The labels identify the quantity in each dimension. For example, in an energy density file, `N` would be `1`, valueunits could be `"J/m3"`, and valuelabels might be `"Exchange energy density"`
- `valuedim` (integer): specifies an integer value, `N`, which is the dimensionality of the field. `N >= 1`
- `xmin`, `ymin`, `zmin`, `xmax`, `ymax`, `zmax`: six separate lines, specifying the bounding box for the mesh, in units of `meshunit`
- `meshtype`: grid structure; one of
  - `rectangular`: Requires also
    - `xbase`, `ybase`, `zbase`: three separate lines, denoting the origin (i.e. the position of the first point in the data section), in units of `meshunit`
    - `xstepsize`, `ystepsize`, `zstepsize`: three separate lines, specifying the distance between adjacent grid points, in units of `meshunit`
    - `xnodes`, `ynodes`, `znodes` (integers): three separate lines, specifying the number of nodes along each axis.

  - `irregular`: Requires also
    * `pointcount` (integer): number of data sample points/locations, i.e., nodes. For irregular grids only

Segment Data

- The data block start is marked by a line of the form `# Begin: data <representation>` (and therefore closed by `# End: data <representation>`), where `<representation>` is one of
  - `text`
  - `binary 4`
  - `binary 8`
- In the Data block, for regular meshes each record consists of `N` values, where `N` is the value dimension as specified by the `valuedim` record in the Segment Header. For irregular meshes, each record consists of `N + 3` values, where the first three values are the x , y and z components of the node position.
- It is common convention for the `text` data to be in `N` columns, separated by whitespace
- Data ordering is generally with the x index incremented first, then the y index, and the z index last

For binary data:

- The binary representations are IEEE 754 standardized floating point numbers in little endian (LSB) order. To ensure that the byte order is correct, and to provide a partial check that the file hasn't been sent through a non 8-bit clean channel, the first data value is fixed to `1234567.0` for 4-byte mode, corresponding to the LSB hex byte sequence `38 B4 96 49`, and `123456789012345.0` for 8-byte mode, corresponding to the LSB hex byte sequence `40 DE 77 83 21 12 DC 42`
- The data immediately follows the check value
- The first character after the last data value should be a newline

Extensions made by this library

These extensions are mainly to help with data for atomistic systems.

- The segment count is padded to 6 digits with zeros (this is so that segments can be appended and the count incremented without having to re-write the entire file)
- Lines starting with a `#` but containing an unknown keyword are ignored.
- `##` is always a comment and is allowed in all keyword lines, including `meshunit` and `valueunits`
- All keywords have default values, so none are required
- `csv` is also a valid ASCII data representation and corresponds to comma-separated columns of `text` type

Current limitations of this library

- naming of variables in structs/classes is inconsistent with the file format specifications
- not all defaults in the segment are guaranteed to be sensible
- `valueunits` and `valuelabels` are written and parsed, but not checked for dimensionality or content in either
- `min` and `max` values are not checked to make sure they are sensible bounds
- `irregular` mesh type is not supported properly, as positions are not accounted for in read or write

Example

An example OVF 2.0 file for an irregular mesh with N = 2:

```
# OOMMF OVF 2.0
#
# Segment count: 1
#
# Begin: Segment
# Begin: Header
#
# Title: Long file name or title goes here
#
# Desc: Optional description line 1.
# Desc: Optional description line 2.
# Desc: ...
#
## Fundamental mesh measurement unit.  Treated as a label:
# meshunit: nm
#
# meshtype: irregular
# pointcount: 5      ## Number of nodes in mesh
#
# xmin:    0.    ## Corner points defining mesh bounding box in
# ymin:    0.    ## 'meshunit'.  Floating point values.
# zmin:    0.
# xmax:   10.
# ymax:    5.
# zmax:    1.
#
# valuedim: 2    ## Value dimension
#
## Fundamental field value units, treated as labels (i.e., unparsed).
## In general, there should be one label for each value dimension.
# valueunits:  J/m^3  A/m
# valuelabels: "Zeeman energy density"  "Anisotropy field"
#
# End: Header
#
## Each data records consists of N+3 values: the (x,y,z) node
## location, followed by the N value components.  In this example,
## N+3 = 5, the two value components are in units of J/m^3 and A/m,
## corresponding to Zeeman energy density and a magneto-crystalline
## anisotropy field, respectively.
#
# Begin: data text
0.5 0.5 0.5  500.  4e4
9.5 0.5 0.5  300.  5e3
0.5 4.5 0.5  400.  4e4
9.5 4.5 0.5  200.  5e3
5.0 2.5 0.5  350.  2.1e4
# End: data text
# End: segment
```

Comparison to OVF 1.0

- The first line reads `# OOMMF OVF 2.0` for both regular and irregular meshes.

- In the segment header block
  - the keywords `valuemultiplier`, `boundary`, `ValueRangeMaxMag` and `ValueRangeMinMag` of the OVF 1.0 format are not supported.
  - the new keyword `valuedim` is required. This must specify an integer value, `N`, bigger or equal to one.
  - the new `valueunits` keyword replaces the `valueunit` keyword of OVF 1.0, which is not allowed in OVF 2.0 files.
  - the new `valuelabels` keyword is required.
- In the segment data block
  - The node ordering is the same as for the OVF 1.0 format.
  - For data blocks using text representation with `N = 3`, the data block in OVF 1.0 and OVF 2.0 files are exactly the same. Another common case is `N = 1`, which represents scalar fields, such as energy density (in say, `J/m3`)

14.3 Eigen

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.

For more information go to http://eigen.tuxfamily.org/.

For ***pull request*** please only use the official repository at https://bitbucket.org/eigen/eigen.

For ***bug reports*** and ***feature requests*** go to http://eigen.tuxfamily.org/bz.

14.4 Spectra

Build Status

Spectra stands for **Sp**arse **E**igenvalue **C**omputation **T**oolkit as a **R**edesigned **A**RPACK. It is a C++ library for large scale eigenvalue problems, built on top of Eigen, an open source linear algebra library.

Spectra is implemented as a header-only C++ library, whose only dependency, **Eigen**, is also header-only. Hence **Spectra** can be easily embedded in C++ projects that require calculating eigenvalues of large matrices.

14.4.1 Relation to ARPACK

ARPACK is a software written in FORTRAN for solving large scale eigenvalue problems. The development of **Spectra** is much inspired by ARPACK, and as the whole name indicates, **Spectra** is a redesign of the ARPACK library using C++ language.

In fact, **Spectra** is based on the algorithms described in the ARPACK Users' Guide, but it does not use the ARPACK code, and it is **NOT** a clone of ARPACK for C++. In short, **Spectra** implements the major algorithms in ARPACK, but **Spectra** provides a completely different interface, and it does not depend on ARPACK.

14.4.2 Common Usage

Spectra is designed to calculate a specified number (`k`) of eigenvalues of a large square matrix (`A`). Usually `k` is much less than the size of matrix (`n`), so that only a few eigenvalues and eigenvectors are computed, which in general is

more efficient than calculating the whole spectral decomposition. Users can choose eigenvalue selection rules to pick up the eigenvalues of interest, such as the largest `k` eigenvalues, or eigenvalues with largest real parts, etc.

To use the eigen solvers in this library, the user does not need to directly provide the whole matrix, but instead, the algorithm only requires certain operations defined on `A`, and in the basic setting, it is simply the matrix-vector multiplication. Therefore, if the matrix-vector product `A * x` can be computed efficiently, which is the case when `A` is sparse, **Spectra** will be very powerful for large scale eigenvalue problems.

There are two major steps to use the **Spectra** library:

1. Define a class that implements a certain matrix operation, for example the matrix-vector multiplication `y = A * x` or the shift-solve operation `y = inv(A - σ * I) * x`. **Spectra** has defined a number of helper classes to quickly create such operations from a matrix object. See the documentation of DenseGenMatProd, DenseSymShiftSolve, etc.
2. Create an object of one of the eigen solver classes, for example SymEigsSolver for symmetric matrices, and GenEigsSolver for general matrices. Member functions of this object can then be called to conduct the computation and retrieve the eigenvalues and/or eigenvectors.

Below is a list of the available eigen solvers in **Spectra**:

- SymEigsSolver: For real symmetric matrices
- GenEigsSolver: For general real matrices
- SymEigsShiftSolver: For real symmetric matrices using the shift-and-invert mode
- GenEigsRealShiftSolver: For general real matrices using the shift-and-invert mode, with a real-valued shift
- GenEigsComplexShiftSolver: For general real matrices using the shift-and-invert mode, with a complex-valued shift
- SymGEigsSolver: For generalized eigen solver for real symmetric matrices

14.4.3 Examples

Below is an example that demonstrates the use of the eigen solver for symmetric matrices.

```
#include <Eigen/Core>
#include <SymEigsSolver.h>  // Also includes <MatOp/DenseSymMatProd.h>
#include <iostream>

using namespace Spectra;

int main()
{
    // We are going to calculate the eigenvalues of M
    Eigen::MatrixXd A = Eigen::MatrixXd::Random(10, 10);
    Eigen::MatrixXd M = A + A.transpose();

    // Construct matrix operation object using the wrapper class DenseGenMatProd
    DenseSymMatProd<double> op(M);

    // Construct eigen solver object, requesting the largest three eigenvalues
    SymEigsSolver< double, LARGEST_ALGE, DenseSymMatProd<double> > eigs(&op, 3, 6);

    // Initialize and compute
    eigs.init();
    int nconv = eigs.compute();
```

(continues on next page)

(continued from previous page)

```
    // Retrieve results
    Eigen::VectorXd evalues;
    if(eigs.info() == SUCCESSFUL)
        evalues = eigs.eigenvalues();

    std::cout << "Eigenvalues found:\n" << evalues << std::endl;

    return 0;
}
```

Sparse matrix is supported via the `SparseGenMatProd` class.

```
#include <Eigen/Core>
#include <Eigen/SparseCore>
#include <GenEigsSolver.h>
#include <MatOp/SparseGenMatProd.h>
#include <iostream>

using namespace Spectra;

int main()
{
    // A band matrix with 1 on the main diagonal, 2 on the below-main subdiagonal,
    // and 3 on the above-main subdiagonal
    const int n = 10;
    Eigen::SparseMatrix<double> M(n, n);
    M.reserve(Eigen::VectorXi::Constant(n, 3));
    for(int i = 0; i < n; i++)
    {
        M.insert(i, i) = 1.0;
        if(i > 0)
            M.insert(i - 1, i) = 3.0;
        if(i < n - 1)
            M.insert(i + 1, i) = 2.0;
    }

    // Construct matrix operation object using the wrapper class SparseGenMatProd
    SparseGenMatProd<double> op(M);

    // Construct eigen solver object, requesting the largest three eigenvalues
    GenEigsSolver< double, LARGEST_MAGN, SparseGenMatProd<double> > eigs(&op, 3, 6);

    // Initialize and compute
    eigs.init();
    int nconv = eigs.compute();

    // Retrieve results
    Eigen::VectorXcd evalues;
    if(eigs.info() == SUCCESSFUL)
        evalues = eigs.eigenvalues();

    std::cout << "Eigenvalues found:\n" << evalues << std::endl;

    return 0;
}
```

And here is an example for user-supplied matrix operation class.

```
#include <Eigen/Core>
#include <SymEigsSolver.h>
#include <iostream>

using namespace Spectra;

// M = diag(1, 2, ..., 10)
class MyDiagonalTen
{
public:
    int rows() { return 10; }
    int cols() { return 10; }
    // y_out = M * x_in
    void perform_op(double *x_in, double *y_out)
    {
        for(int i = 0; i < rows(); i++)
        {
            y_out[i] = x_in[i] * (i + 1);
        }
    }
};

int main()
{
    MyDiagonalTen op;
    SymEigsSolver<double, LARGEST_ALGE, MyDiagonalTen> eigs(&op, 3, 6);
    eigs.init();
    eigs.compute();
    if(eigs.info() == SUCCESSFUL)
    {
        Eigen::VectorXd evalues = eigs.eigenvalues();
        std::cout << "Eigenvalues found:\n" << evalues << std::endl;
    }

    return 0;
}
```

14.4.4 Shift-and-invert Mode

When we want to find eigenvalues that are closest to a number σ, for example to find the smallest eigenvalues of a positive definite matrix (in which case $\sigma = 0$), it is advised to use the shift-and-invert mode of eigen solvers.

In the shift-and-invert mode, selection rules are applied to $1/(\lambda - \sigma)$ rather than λ, where λ are eigenvalues of `A`. To use this mode, users need to define the shift-solve matrix operation. See the documentation of SymEigsShiftSolver for details.

14.4.5 Documentation

The API reference page contains the documentation of **Spectra** generated by Doxygen, including all the background knowledge, example code and class APIs.

More information can be found in the project page http://yixuan.cos.name/spectra.

14.4.6 License

Spectra is an open source project licensed under MPL2, the same license used by **Eigen**.

14.5 {fmt}

build passing

BUILDING...

fmt is an open-source formatting library for C++. It can be used as a safe alternative to printf or as a fast alternative to IOStreams.

Documentation

14.5.1 Features

- Two APIs: faster concatenation-based write API and slower, but still very fast, replacement-based format API with positional arguments for localization.
- Write API similar to the one used by IOStreams but stateless allowing faster implementation.
- Format API with format string syntax similar to the one used by str.format in Python.
- Safe printf implementation including the POSIX extension for positional arguments.
- Support for user-defined types.
- High speed: performance of the format API is close to that of glibc's printf and better than the performance of IOStreams. See *Speed tests* and Fast integer to string conversion in C++.
- Small code size both in terms of source code (the core library consists of a single header file and a single source file) and compiled code. See *Compile time and code bloat*.
- Reliability: the library has an extensive set of unit tests.
- Safety: the library is fully type safe, errors in format strings are reported using exceptions, automatic memory management prevents buffer overflow errors.
- Ease of use: small self-contained code base, no external dependencies, permissive BSD license
- Portability with consistent output across platforms and support for older compilers.
- Clean warning-free codebase even on high warning levels (-Wall -Wextra -pedantic).
- Support for wide strings.
- Optional header-only configuration enabled with the `FMT_HEADER_ONLY` macro.

See the documentation for more details.

14.5.2 Examples

This prints `Hello, world!` to stdout:

```
fmt::print("Hello, {}!", "world");  // uses Python-like format string syntax
fmt::printf("Hello, %s!", "world"); // uses printf format string syntax
```

Arguments can be accessed by position and arguments' indices can be repeated:

```
std::string s = fmt::format("{0}{1}{0}", "abra", "cad");
// s == "abracadabra"
```

fmt can be used as a safe portable replacement for `itoa`:

```
fmt::MemoryWriter w;
w << 42;           // replaces itoa(42, buffer, 10)
w << fmt::hex(42); // replaces itoa(42, buffer, 16)
// access the string using w.str() or w.c_str()
```

An object of any user-defined type for which there is an overloaded `std::ostream` insertion operator (`operator<<`) can be formatted:

```
#include "fmt/ostream.h"

class Date {
  int year_, month_, day_;
 public:
  Date(int year, int month, int day) : year_(year), month_(month), day_(day) {}

  friend std::ostream &operator<<(std::ostream &os, const Date &d) {
    return os << d.year_ << '-' << d.month_ << '-' << d.day_;
  }
};

std::string s = fmt::format("The date is {}", Date(2012, 12, 9));
// s == "The date is 2012-12-9"
```

You can use the FMT_VARIADIC macro to create your own functions similar to format and print which take arbitrary arguments:

```
// Prints formatted error message.
void report_error(const char *format, fmt::ArgList args) {
  fmt::print("Error: ");
  fmt::print(format, args);
}
FMT_VARIADIC(void, report_error, const char *)

report_error("file not found: {}", path);
```

Note that you only need to define one function that takes `fmt::ArgList` argument. `FMT_VARIADIC` automatically defines necessary wrappers that accept variable number of arguments.

14.5.3 Projects using this library

- 0 A.D.: A free, open-source, cross-platform real-time strategy game
- AMPL/MP: An open-source library for mathematical programming
- CUAUV: Cornell University's autonomous underwater vehicle
- Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems (MIT)
- Envoy: C++ L7 proxy and communication bus (Lyft)
- FiveM: a modification framework for GTA V

- HarpyWar/pvpgn: Player vs Player Gaming Network with tweaks
- KBEngine: An open-source MMOG server engine
- Keypirinha: A semantic launcher for Windows
- Kodi (formerly xbmc): Home theater software
- Lifeline: A 2D game
- MongoDB Smasher: A small tool to generate randomized datasets
- OpenSpace: An open-source astrovisualization framework
- PenUltima Online (POL): An MMO server, compatible with most Ultima Online clients
- quasardb: A distributed, high-performance, associative database
- readpe: Read Portable Executable
- redis-cerberus: A Redis cluster proxy
- Saddy: Small crossplatform 2D graphic engine
- Salesforce Analytics Cloud: Business intelligence software
- Scylla: A Cassandra-compatible NoSQL data store that can handle 1 million transactions per second on a single server
- Seastar: An advanced, open-source C++ framework for high-performance server applications on modern hardware
- spdlog: Super fast C++ logging library
- Stellar: Financial platform
- Touch Surgery: Surgery simulator
- TrinityCore: Open-source MMORPG framework

More…

If you are aware of other projects using this library, please let me know by email or by submitting an issue.

14.5.4 Motivation

So why yet another formatting library?

There are plenty of methods for doing this task, from standard ones like the printf family of function and IOStreams to Boost Format library and FastFormat. The reason for creating a new library is that every existing solution that I found either had serious issues or didn't provide all the features I needed.

Printf

The good thing about printf is that it is pretty fast and readily available being a part of the C standard library. The main drawback is that it doesn't support user-defined types. Printf also has safety issues although they are mostly solved with __attribute__ ((format (printf, …)) in GCC. There is a POSIX extension that adds positional arguments required for i18n to printf but it is not a part of C99 and may not be available on some platforms.

IOStreams

The main issue with IOStreams is best illustrated with an example:

```
std::cout << std::setprecision(2) << std::fixed << 1.23456 << "\n";
```

which is a lot of typing compared to printf:

```
printf("%.2f\n", 1.23456);
```

Matthew Wilson, the author of FastFormat, referred to this situation with IOStreams as "chevron hell". IOStreams doesn't support positional arguments by design.

The good part is that IOStreams supports user-defined types and is safe although error reporting is awkward.

Boost Format library

This is a very powerful library which supports both printf-like format strings and positional arguments. The main its drawback is performance. According to various benchmarks it is much slower than other methods considered here. Boost Format also has excessive build times and severe code bloat issues (see *Benchmarks*).

FastFormat

This is an interesting library which is fast, safe and has positional arguments. However it has significant limitations, citing its author:

> Three features that have no hope of being accommodated within the current design are:
>
> - Leading zeros (or any other non-space padding)
> - Octal/hexadecimal encoding
> - Runtime width/alignment specification

It is also quite big and has a heavy dependency, STLSoft, which might be too restrictive for using it in some projects.

Loki SafeFormat

SafeFormat is a formatting library which uses printf-like format strings and is type safe. It doesn't support user-defined types or positional arguments. It makes unconventional use of `operator()` for passing format arguments.

Tinyformat

This library supports printf-like format strings and is very small and fast. Unfortunately it doesn't support positional arguments and wrapping it in C++98 is somewhat difficult. Also its performance and code compactness are limited by IOStreams.

Boost Spirit.Karma

This is not really a formatting library but I decided to include it here for completeness. As IOStreams it suffers from the problem of mixing verbatim text with arguments. The library is pretty fast, but slower on integer formatting than `fmt::Writer` on Karma's own benchmark, see Fast integer to string conversion in C++.

14.5.5 Benchmarks

Speed tests

The following speed tests results were generated by building `tinyformat_test.cpp` on Ubuntu GNU/Linux 14.04.1 with `g++-4.8.2 -O3 -DSPEED_TEST -DHAVE_FORMAT`, and taking the best of three runs. In the test, the format string `"%0.10f:%04d:%+g:%s:%p:%c:%%\n"` or equivalent is filled 2000000 times with output sent to `/dev/null`; for further details see the source.

Library	Method	Run Time, s
EGLIBC 2.19	printf	1.30
libstdc++ 4.8.2	std::ostream	1.85
fmt 1.0	fmt::print	1.42
tinyformat 2.0.1	tfm::printf	2.25
Boost Format 1.54	boost::format	9.94

As you can see `boost::format` is much slower than the alternative methods; this is confirmed by other tests. Tinyformat is quite good coming close to IOStreams. Unfortunately tinyformat cannot be faster than the IOStreams because it uses them internally. Performance of fmt is close to that of printf, being faster than printf on integer formatting, but slower on floating-point formatting which dominates this benchmark.

Compile time and code bloat

The script bloat-test.py from format-benchmark tests compile time and code bloat for nontrivial projects. It generates 100 translation units and uses `printf()` or its alternative five times in each to simulate a medium sized project. The resulting executable size and compile time (g++-4.8.1, Ubuntu GNU/Linux 13.10, best of three) is shown in the following tables.

Optimized build (-O3)

Method	Compile Time, s	Executable size, KiB	Stripped size, KiB
printf	2.6	41	30
IOStreams	19.4	92	70
fmt	46.8	46	34
tinyformat	64.6	418	386
Boost Format	222.8	990	923

As you can see, fmt has two times less overhead in terms of resulting code size compared to IOStreams and comes pretty close to `printf`. Boost Format has by far the largest overheads.

Non-optimized build

Method	Compile Time, s	Executable size, KiB	Stripped size, KiB
printf	2.1	41	30
IOStreams	19.7	86	62
fmt	47.9	108	86
tinyformat	27.7	234	190
Boost Format	122.6	884	763

`libc`, `libstdc++` and `libfmt` are all linked as shared libraries to compare formatting function overhead only. Boost Format and tinyformat are header-only libraries so they don't provide any linkage options.

Running the tests

Please refer to Building the library for the instructions on how to build the library and run the unit tests.

Benchmarks reside in a separate repository, format-benchmarks, so to run the benchmarks you first need to clone this repository and generate Makefiles with CMake:

```
$ git clone --recursive https://github.com/fmtlib/format-benchmark.git
$ cd format-benchmark
$ cmake .
```

Then you can run the speed test:

```
$ make speed-test
```

or the bloat test:

```
$ make bloat-test
```

14.5.6 License

fmt is distributed under the BSD license.

The Format String Syntax section in the documentation is based on the one from Python string module documentation adapted for the current library. For this reason the documentation is distributed under the Python Software Foundation license available in doc/python-license.txt. It only applies if you distribute the documentation of fmt.

14.5.7 Acknowledgments

The fmt library is maintained by Victor Zverovich (vitaut) and Jonathan Müller (foonathan) with contributions from many other people. See Contributors and Releases for some of the names. Let us know if your contribution is not listed or mentioned incorrectly and we'll make it right.

The benchmark section of this readme file and the performance tests are taken from the excellent tinyformat library written by Chris Foster. Boost Format library is acknowledged transitively since it had some influence on tinyformat. Some ideas used in the implementation are borrowed from Loki SafeFormat and Diagnostic API in Clang. Format string syntax and the documentation are based on Python's str.format. Thanks Doug Turnbull for his valuable comments and contribution to the design of the type-safe API and Gregory Czajkowski for implementing binary formatting. Thanks Ruslan Baratov for comprehensive comparison of integer formatting algorithms and useful comments regarding performance, Boris Kaul for C++ counting digits benchmark. Thanks to CarterLi for contributing various improvements to the code.

14.6 Termcolor

core/thirdparty/termcolor/docs/_static/example.png

Termcolor is a header-only C++ library for printing colored messages to the terminal. Written just for fun with a help of the Force. Termcolor uses ANSI color formatting, so you can use it on every system that is used such terminals

(most *nix systems, including Linux and Mac OS). On Windows, WinAPI is used instead but some limitations are applied.

It's licensed under the BSD (3-clause) License. That basically means: do whatever you want as long as copyright sticks around.

14.6.1 Installation

Add `termcolor.hpp` to the project and use provided stream manimulators from the `termcolor` namespace.

14.6.2 How to use?

It's very easy to use. The idea is based on the use of C++ stream manipulators. The typical «Hello World» application is below:

```
#include <iostream>
#include <termcolor/termcolor.hpp>

int main(int /*argc*/, char** /*argv*/)
{
    std::cout << termcolor::red << "Hello, Colorful World!" << std::endl;
    return 0;
}
```

The application above prints a string with red. It's obvious, isn't it? There is a one problem that is not obvious for the unexperienced users. If you write something this way:

```
std::cout << termcolor::red << "Hello, Colorful World!" << std::endl;
std::cout << "Here I'm!" << std::endl;
```

the phrase «Here I'm» will be printed with red too. Why? Because you don't reset termcolor's setting. So if you want to print text wit default terminal setting you have to reset termcolor's settings. It can be done by using `termcolor::reset` manipulator:

```
std::cout << termcolor::red << "Hello, Colorful World!" << std::endl;
std::cout << termcolor::reset << "Here I'm!" << std::endl;
```

By default, Termcolor ignores any colors for non-tty streams (e.g. `std::stringstream`), so:

```
std::stringstream ss;
ss << termcolor::red << "unicorn";
std::cout << ss.str();
```

would print «unicorn» using default color, not red. In order to change this behaviour one can use `termcolor::colorize` manipulator that enforce colors no matter what.

14.6.3 What manipulators are supported?

The manipulators are divided into four groups:

- *foreground*, which changes text color;
- *background*, which changes text background color;
- *attributes*, which changes some text style (bold, underline, etc);

- *control*, which changes termcolor's behaviour.

Foreground manipulators

1. `termcolor::grey`
2. `termcolor::red`
3. `termcolor::green`
4. `termcolor::yellow`
5. `termcolor::blue`
6. `termcolor::magenta`
7. `termcolor::cyan`
8. `termcolor::white`

Background manipulators

1. `termcolor::on_grey`
2. `termcolor::on_red`
3. `termcolor::on_green`
4. `termcolor::on_yellow`
5. `termcolor::on_blue`
6. `termcolor::on_magenta`
7. `termcolor::on_cyan`
8. `termcolor::on_white`

Attribute manipulators

(so far they aren't supported on Windows)

1. `termcolor::bold`
2. `termcolor::dark`
3. `termcolor::underline`
4. `termcolor::blink`
5. `termcolor::reverse`
6. `termcolor::concealed`

Control manipulators

(so far they aren't supported on Windows)

1. `termcolor::colorize`
2. `termcolor::nocolorize`

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

C

D

E

F

G

H

I

J

K

M

N

P

R

S

T

U