Spirit Documentation

Gideon Mueller and contributors

Jul 06, 2018

Introduction:

CHAPTER 1

SPIRIT

SPIN SIMULATION FRAMEWORK

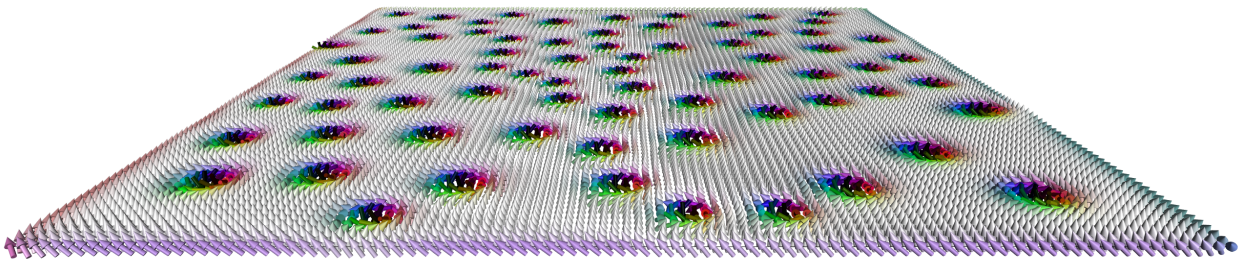
Core Library:

Python package:

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 read the REFERENCE.md

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

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



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*

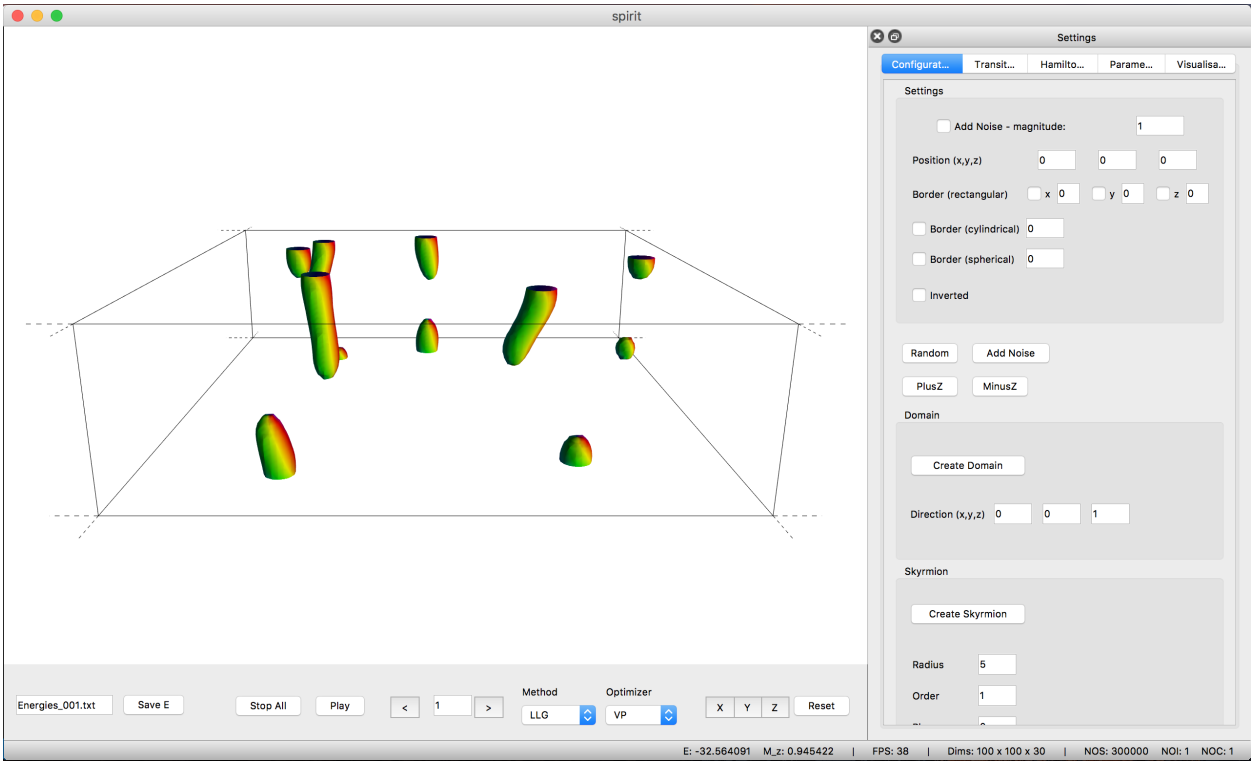
More details may be found at spirit-docs.readthedocs.io or in the Reference section including

- Framework build instructions
- Core build instructions
- Core API Reference
- Python API Reference
- Input File Reference

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

1.3 Getting started with the Desktop Interface

See BUILD.md on how to install the desktop user interface.



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 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.PlayPause(p_state, "LLG", "SIB")
```

Where `"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.PlusZ(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 a few times
chain.Image_to_Clipboard(p_state)
for number in range(1,7):
    chain.Insert_Image_After(p_state)
noi = chain.Get_NOI(p_state)

### First image is homogeneous with a Skyrmion in the center
configuration.PlusZ(p_state, idx_image=0)
configuration.Skyrmion(p_state, 5.0, phase=-90.0, idx_image=0)
simulation.PlayPause(p_state, "LLG", "VP", idx_image=0)
### Last image is homogeneous
configuration.PlusZ(p_state, idx_image=noi-1)
simulation.PlayPause(p_state, "LLG", "VP", idx_image=noi-1)

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

### GNEB calculation
simulation.PlayPause(p_state, "GNEB", "VP")
```

where `"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

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 framework build instructions for information on how to build the 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

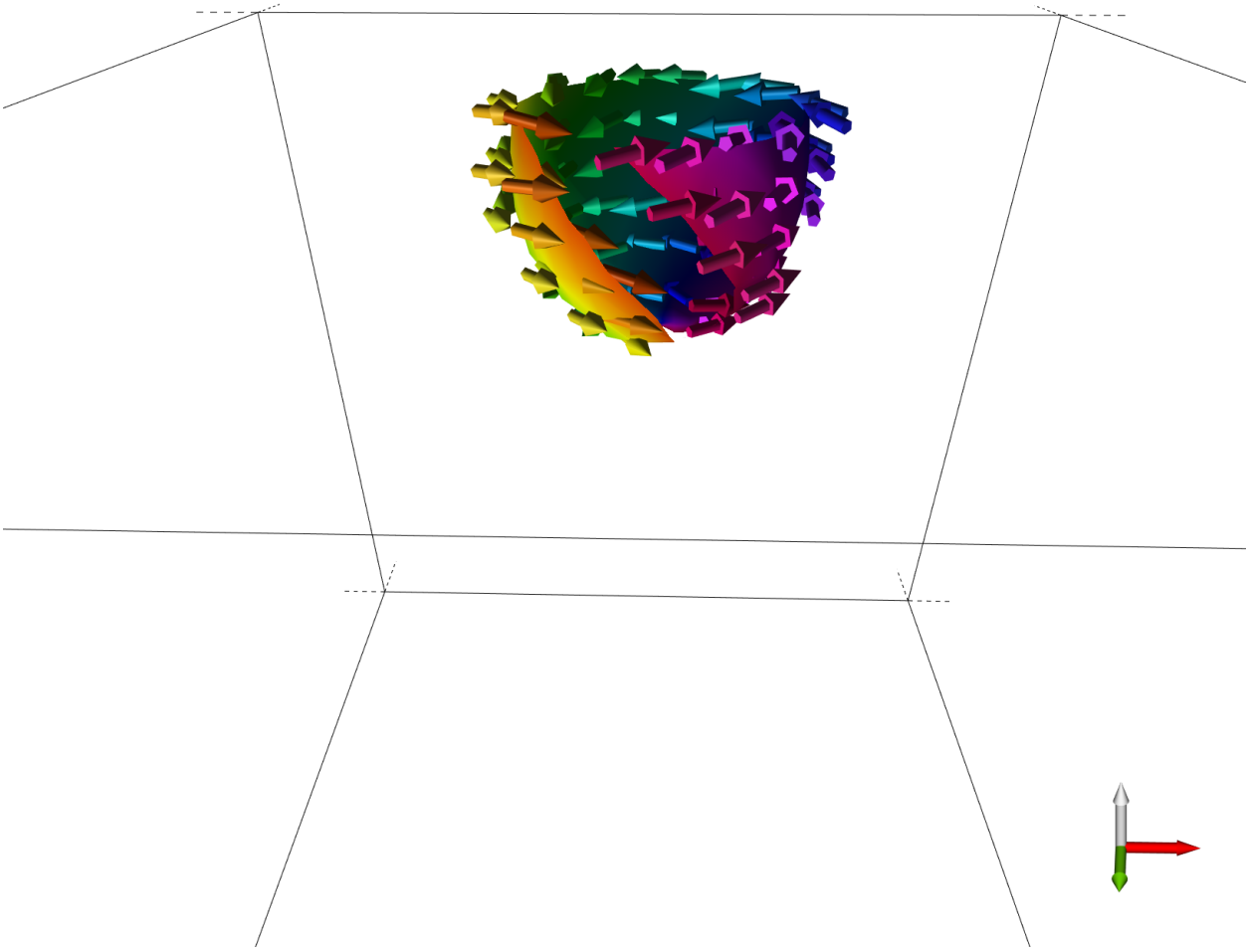
By copying and inserting spin systems and manipulating them you may create arbitrary transitions between different spin states to use them in GNEB calculations. Furthermore you can choose different images to be climbing or falling images during your calculation.

2.2 Real-time visualisation

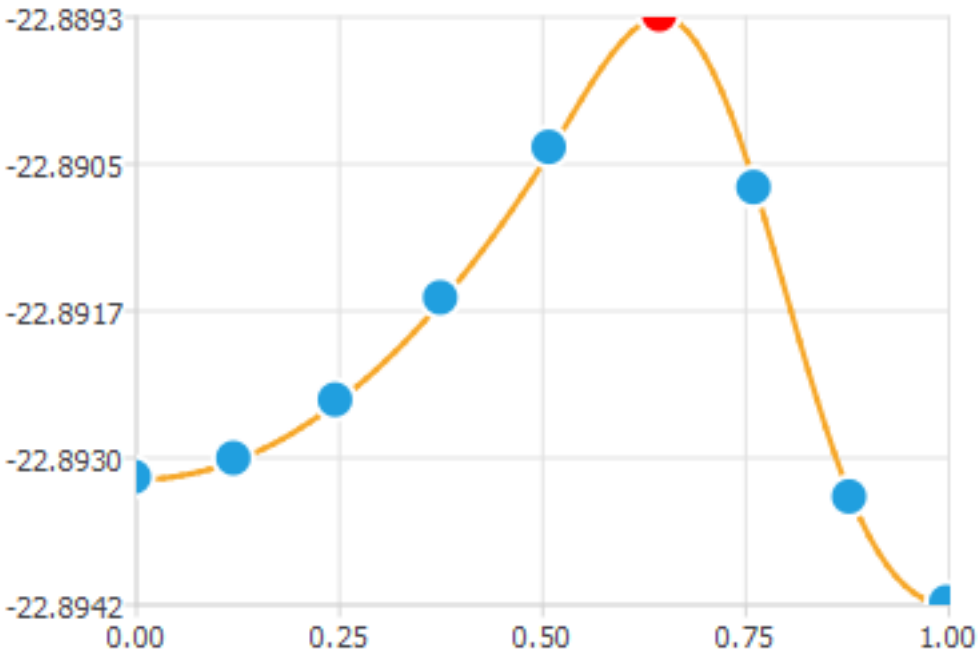
This feature is most powerful for 3D systems but shows great use for the analysis of dynamical processes and understanding what is happening in your system during a simulation instead of post-processing your data.

- Arrows, Surface (2D/3D), Isosurface
- Spins or Eff. Field
- Every n'th arrow
- Spin Sphere
- Directional & Position filters
- Colormaps

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



Note that a data plot is available to visualise your chain of spin systems. It can also show interpolated energies if you run a GNEB calculation.



2.3 Additional features

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

2.4 Key bindings

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

2.4.1 UI Controls

2.4.2 Camera Controls

2.4.3 Control Simulations

2.4.4 Manipulate the current images

2.4.5 Visualisation

2.4.6 Manipulate the chain of images

Home

CHAPTER 3

Building Spirit's Framework Components

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

Please be aware that our CMake scripts are written for our use cases and **you may need to adapt some paths and options in the Root CMakeLists.txt**.

3.1 Contents

3.2 General Build Process

The following assumes you are in the Spirit root directory.

3.2.1 Options

There are some important **Options** you may need to consider. You can find them under *### Build Flags ###* in the **Root CMakeLists.txt**. Otherwise, the developers' defaults will be used.

Some **Paths** you can set under *### User Paths ###* (just uncomment the corresponding line) are:

3.2.2 Clean

Clear the build directory using

```
./clean.sh
or
rm -rf build && mkdir build
```

Further helper scripts for clean-up are `clean_log.sh`, `clean_output.sh`,

3.2.3 Generate Build Files

`./cmake.sh` lets cmake generate makefiles for your system inside a 'build' folder. Simply call

```
./cmake.sh
or
cd build && cmake .. && cd ..
```

Passing `-debug` to the script will cause it to create a debug configuration, meaning that you will be able to properly debug the entire application.

On **Windows** (no MSys) you can simply use the git bash to do this or use the CMake GUI. When using MSys etc., CMake will create corresponding MSys makefiles.

3.2.4 Building the Projects

To execute the build and linking of the executable, simply call

```
./make.sh -jN
or
cd build && make -jN && cd ..
```

where `-jN` is optional, with `N` the number of parallel build processes you want to create.

On **Windows** (no MSys), CMake will by default have generated a Visual Studio Solution. Open the generated Solution in the Visual Studio IDE and build it there.

3.2.5 Running the Unit Tests

We use `CMakes CTest` for unit testing. You can run

```
ctest.sh
or
cd build && ctest --output-on-failure && cd ..
```

or execute any of the test executables manually. To execute the tests from the Visual Studio IDE, simply rebuild the `RUN_TESTS` project.

3.2.6 Installing Components

This is not yet supported! however, you can already run

```
./install.sh
or
cd build && make install && cd ..
```

Which on OSX should build a .app bundle.

3.3 Core Library

For detailed build instructions concerning the standalone core library or how to include it in your own project, see core/docs/BUILD.md.

- Shared and static library
- Python bindings
- Julia bindings
- Transpiling to JavaScript
- Unit Tests

The **Root CMakeLists.txt** has a few options you can set:

3.4 Desktop User Interface

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.

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

Home

CHAPTER 4

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*

4.1 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.

4.2 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

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

2D honeycomb example:

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

### 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

### 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
###   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
```

4.3 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 vector[T]
external_field_magnitude 25.0
external_field_normal    0.0 0.0 1.0
### µSpin
mu_s                     2.0

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

### Dipole-Dipole radius
dd_radius          0.0
```

If you have a nontrivial basis cell, note that you should specify `mu_s` for all atoms in your basis cell.

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`.

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
```

(continues on next page)

(continued from previous page)

```
### 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
```

4.4 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
```

4.5 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.

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
```

4.6 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
llg_dt               1.0E-3

### 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
```

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
```

4.7 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
```

(continues on next page)

(continued from previous page)

```
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 directions
### ispin S_x S_y S_z
n_pinned 3
0 1.0 0.0 0.0
1 0.0 1.0 0.0
2 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.

4.8 Disorder and Defects

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

Disorder is not yet implemented.

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
### ispin itype
n_defects 3
0 -1
1 -1
2 -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 5

BUILD THE SPIRIT LIBRARY

5.1 C/C++ Library

This is pretty much a standalone library and should be easy to implement into existing projects in which CMake is already used. It can be built as shared or static and API headers are located in `path/to/Spirit/core/include/Spirit`

Note that the CMake Options for the core library are also set in the root CMakeLists.txt.

5.1.1 Integrating into other CMake Projects

This is currently untested, but you should be able to use *Spirit* either via `ExternalProject` or `AddSubdirectory`. You should thus be able to simply copy the `core` directory as `Spirit` into the thirdparty folder of your Project and use it as is done commonly.

Note that setting `qhull_LIBS` and `qhull_INCLUDE_DIRS` if qhull is present on your disk removes the need for Spirit to clone and build it separately.

A set of CMake variables is pushed to the parent scope by the core CMakeLists. These include:

- SPIRIT_LIBRARIES
- SPIRIT_LIBRARIES_STATIC
- SPIRIT_INCLUDE_DIRS

5.2 Unit Tests

The core library includes a set of unit tests in order to make sure certain conditions are fulfilled by the librarys functions. We use `CMakes CTest` for unit testing. You can run

```
cd build && ctest --output-on-failure && cd ..
```

or execute any of the test executables manually. To execute the tests from the Visual Studio IDE, simply rebuild the `RUN_TESTS` project.

5.3 Python Package

You may use the python package simply by adding a `path/to/Spirit/core/python` to your PYTHONPATH.

To build a proper python package including binary wheel, you need to build the shared library for Python and then execute

```
cd path/to/Spirit/core/python
python setup.py sdist bdist_wheel
```

The resulting package can then be installed with e.g.

```
pip install -e /path/to/package/dist/spirit-<tags>.whl
```

5.4 Julia Package

These bindings are currently completely experimental. However, it seems it would be easy to create Julia bindings analogous to the *Python Bindings*.

Home

CHAPTER 6

SPIRIT API

This will list the available API functions of the Spirit library.

The API is exposed as a C interface and may thus also be used from other languages, such as Python, Julia or even JavaScript (see *ui-web*). The API revolves around a simulation `State` which contains all the necessary data and keeps track of running Solvers.

The API exposes functions for:

- Control of simulations
- Manipulation of parameters
- Extracting information
- Generating spin configurations and transitions
- Logging messages
- Reading spin configurations
- Saving datafiles

6.1 State Managment

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_PlayPause(p_state, "LLG", "SIB"); // Start a LLG simulation using the SIB␣
↪solver
State_Delete(p_state)                         // State cleanup
```

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.**

The State struct is passed around in an application to make the simulation's state available.

6.2 System

6.3 Simulation

With `Simulation_*` functions one can control and get information from the State's Solvers and their Methods.

6.4 Geometry

6.5 Transitions

6.6 Quantities

6.7 Parameters

6.8 Chain

Get Chain's information

Move between images (change `active_image`)

Insertion/deletion and replacement of images are done by

6.9 Hamiltonian

Set Hamiltonian's parameters

Get Hamiltonian's parameters

6.10 Constants

6.11 Log

Log macro variables for Levels

Log macro variables for Senders

6.12 IO

Read and Write functions

Energies from `System` and `Chain`

| Chain Energies | Return | | ---------- | | `IO_Chain_Write_Energies( State *, const char* file, int idx_chain )` | `void` | | `IO_Chain_Write_Energies_Interpolated( State *, const char* file, int idx_chain )` | `void` |

Home

CHAPTER 7

SPIRIT Python API

7.1 State

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**:

```
from spirit import state
from spirit import simulation

cfgfile = "input/input.cfg"                     # Input File
with state.State(cfgfile) as p_state:           # State setup
    simulation.PlayPause(p_state, "LLG", "SIB") # Start a LLG simulation using the␣
↪SIB solver
```

or call setup and delete manually:

```
from spirit import state
from spirit import simulation

cfgfile = "input/input.cfg"                 # Input File
p_state = state.setup(cfgfile)              # State setup
simulation.PlayPause(p_state, "LLG", "SIB") # Start a LLG simulation using the SIB␣
↪solver
state.delete(p_state)                       # State cleanup
```

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.**

7.2 System

7.3 Chain

For having more images one can copy the active image in the Clipboard and then insert in a specified position of the chain.

```
chain.Image_to_Clipboard(p_state )    # Copy p_state to Clipboard
chain.Insert_Image_After(p_state )    # Insert the image from Clipboard right after␣
↪the currently active image
```

For getting the total number of images in the chain

```
number_of_images = chain.Get_NOI(p_state )
```

7.4 Constants

7.5 Geometry

7.6 Hamiltonian

7.7 Log

7.8 Parameters

7.8.1 LLG

7.8.2 GNEB

7.9 Quantities

7.10 Simulation

The available `method_types` are:

The available `solver_types` are:

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

7.11 Transition

7.12 Input/Output

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.

Home

CHAPTER 8

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.

8.1 *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 9

Contributors

9.1 Gideon 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

(Oct. 2014 - ongoing)

9.2 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)

9.3 Nikolai 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

(2007 - ongoing)

9.4 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)

9.5 Stefanos Mavros

- RWTH Aachen

Work on unit testing and documentation, implementation of the Depondt solver. Also some general code design and IO improvements.

(April 2017 - ongoing)

9.6 Constantin Disselkamp

- RWTH Aachen

Implementation and testing of gradient approximation of spin transfer torque.

(April 2017 - Juli 2017)

9.7 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.

(June 2016 - ongoing)

9.8 Pavel Bessarab

- Various Universities

Help with the initial GNEB implementation. Initial implementation of the HTST method.

(April 2015 - ongoing)

9.9 Filipp 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)

9.10 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)

9.11 Mathias Redies, Maximilian Merte, Rene Suckert

- RWTH Aachen

Initial CUDA implementation and tests. Code optimizations, suggestions and feedback.

(Sept. 2016 - Dec. 2016)

9.12 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)

9.13 Graph

You may also take a look at the contributors graph.

Home

CHAPTER 10

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.

10.1 The Framework

If you used any components 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)
```

10.2 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).
```

10.3 Papers

Here we will list select papers for which the Spirit framework was used, which may be of interest to you.

Home

CHAPTER 11

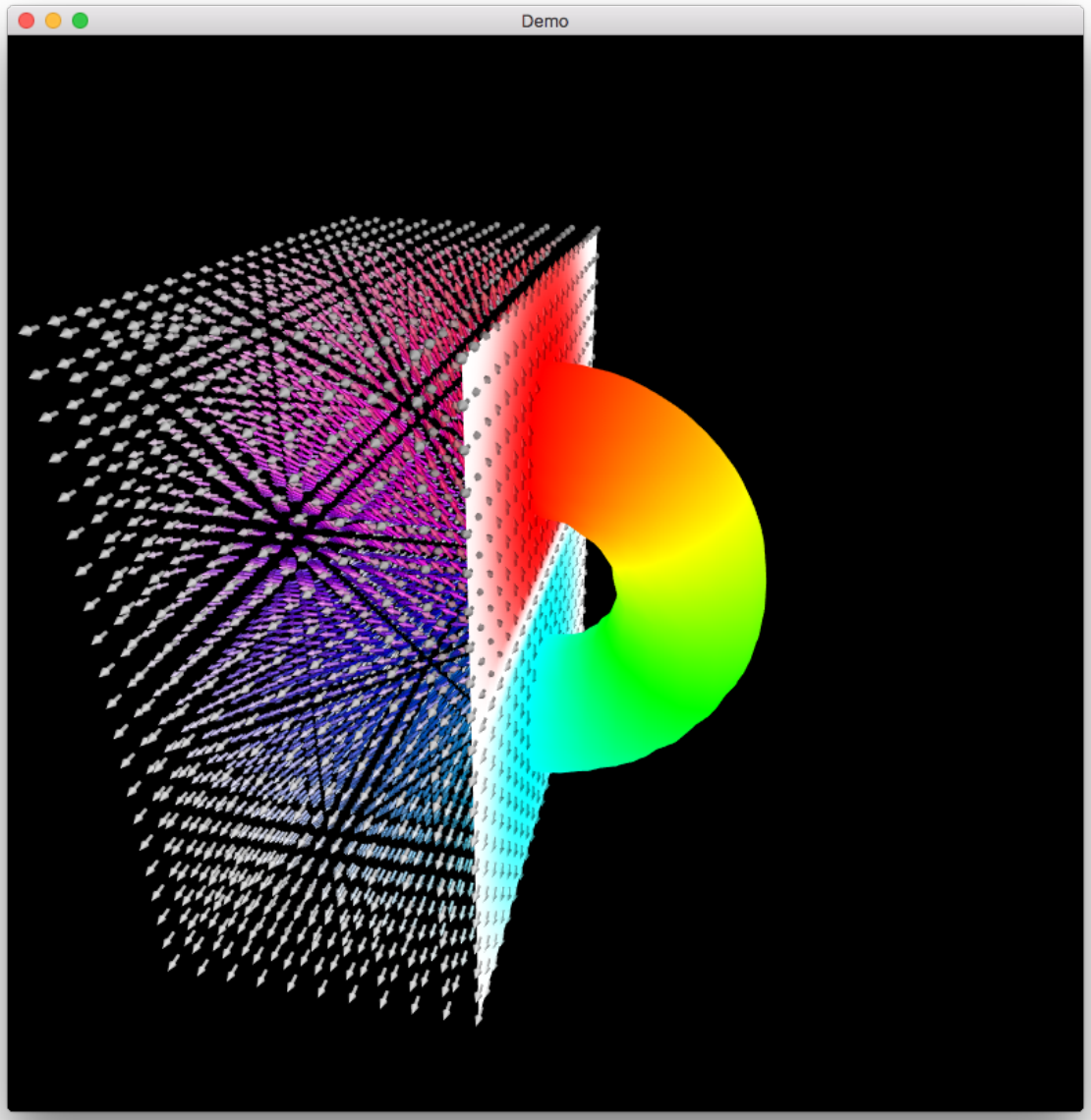
Included Dependencies

11.1 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!



11.1.1 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.

11.1.2 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.

11.1.3 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`.

11.1.4 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:

11.1.5 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.

11.2 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/.

11.3 Spectra

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.

11.3.1 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.

11.3.2 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

11.3.3 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;

```

(continues on next page)

(continued from previous page)

```
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();

    // 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);
```

(continues on next page)

(continued from previous page)

```
    // 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;
}
```

11.3.4 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.

11.3.5 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.

11.3.6 License

Spectra is an open source project licensed under MPL2, the same license used by **Eigen**.

11.4 {fmt}

Documentation

11.4.1 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.

11.4.2 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.

11.4.3 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.

11.4.4 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++.

11.4.5 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
```

11.4.6 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.

11.4.7 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.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search