pyALF Documentation

Jonas Schwab

Institut für Theoretische Physik und Astrophysik, Universität Würzburg, 97074 Würzburg, Germany
Würzburg-Dresden Cluster of Excellence ct.qmat, Universität Würzburg, 97074 Würzburg, Germany
jonas.schwab@uni-wuerzburg.de

February 14, 2026

Abstract

The auxiliary-field quantum Monte Carlo package ALF [1, 2] is a powerful tool for simulating a broad set of fermionic systems, but since it is written in Fortran, it is not very dynamic and can be a bit daunting for new users.

Aiming to address this challenge, pyALF is a set of Python scripts built on top of ALF. It is meant to simplify the different steps of working with ALF, including:

- Obtaining and compiling the ALF source code
- Preparing and running simulations
- Postprocessing and displaying the data obtained during the simulation

The source codes for both ALF and pyALF are publicly available at https://github.com/ALF-QMC.

This documentation is structured in the following way:

1. Section 1 describes the prerequisites of pyALF and how to set things up to be able to use it in a productive manner.
2. Section 2 displays the features of pyALF and how to use them on small examples.
3. For a reference on pyALF's features, see Section 3.

CONTENTS

CHAPTER

ONE

PREREQUISITES AND INSTALLATION

This section lists the prerequisites of pyALF and how to set things up to be able to use it in a productive manner.

1.1 ALF prerequisites

Since pyALF builds on ALF, we also want to satisfy its requirements. Note, however, that pyALF's postprocessing features are independent from ALF. This might be relevant, for example, when performing QMC runs and analysis on different machines.

The minimal ALF prerequisites are:

- The Unix shell Bash
- Make
- A recent Fortran Compiler (e. g. Submodules must be supported)
- BLAS+LAPACK
- Python 3

For parallelization, an MPI development library, e. g. Open MPI, is necessary.

Results from ALF can either be saved in a plain text format or HDF5, but full pyALF support is only provided for the latter, which is why in pyALF, HDF5 is enabled by default. ALF automatically downloads and compiles HDF5. For this to succeed, the following is needed:

- A C compiler (which is most often automatically included when installing a Fortran Compiler)
- A C++ preprocessor
- Curl or Wget
- gzip development libraries

The recommended way for obtaining the source code is through git.

Finally, the ALF testsuite needs:

- CMake

As an example, the requirements mentioned above can be satisfied on a Debian, Ubuntu, or similar operating system using the APT package manager, by executing the command:

```
sudo apt install make gfortran libblas-dev liblapack-dev \
          python3 libopenmpi-dev g++ curl libghc-zlib-dev \
          git ca-certificates cmake bash
```

The above installs compilers from the GNU compiler collection. Other supported and tested compiler frameworks are from the Intel® oneAPI Toolkits and the NVIDIA HPC SDK. The latter is denoted as `PGI` in ALF.

1.2 pyALF installation

> **Warning**
>
> In previous versions of pyALF, the installation instructions asked the users to set the environment variable `PYTHONPATH`. This conflicts with the newer pip package, therefore you should remove definitions of the `PYTHONPATH` environment variable related to pyALF.

pyALF can be installed via the Python package installer pip.

```
pip install pyALF
```

It automatically installs all requirements, but in case you want to install them in a different way, e.g. through apt or conda, these are the Python packages pyALF depends on:

- f90nml
- h5py
- ipympl
- ipywidgets
- matplotlib
- numba
- numpy
- pandas
- scipy
- tkinter

1.2.1 Development installation

If you want to develop pyALF, you can clone the repository and install it in development mode, which allows you to edit the files while using them like an installed package. For this, it is highly recommended to use a dedicated Python environment using e.g. Python venv or a conda environment. The following example shows how to install pyALF in development mode using venv.

```
git clone https://https://github.com/ALF-QMC/pyALF.git
cd pyALF
python -m venv .venv
source .venv/bin/activate

pip install --editable .
```

1.3 Setting ALF directory through environment variable

Since pyALF is set up to automatically clone ALF with git, it is not strictly necessary to download ALF manually, but pyALF will download ALF every time it does not find it. Therefore it is recommended to clone ALF once manually from here and setting its location in the environment variable `ALF_DIR`. This way, pyALF will use the same ALF source code directory every time.

ALF can be cloned with the Unix shell command

```
git clone https://github.com/ALF-QMC/ALF.git
```

This will create a folder called `ALF` in the current working directory of the terminal and download the repository there[1].

The environment variable can then be set with the command

```
export ALF_DIR="/path/to/ALF"
```

where `/path/to/ALF` is the location of the ALF code, for example `/home/jonas/Programs/ALF`. To not have to repeat this command in every terminal session, it is advisable to add it to a file sourced when starting the shell, e.g. `~/.bashrc` or `~/.zshrc`.

1.4 Check setup

To check if most things have been set up correctly, the script `minimal_ALF_run` can be used. It executes the same commands as the *Minimal example*. One should therefore be able to run it by executing

```
minimal_ALF_run
```

in the Unix shell. If it does clone the ALF repository, `ALF_DIR` has not been set up correctly. Note that on the first compilation, ALF downloads and compiles HDF5, which can take up to ~15 minutes.

1.5 Using Jupyter Notebooks

A convenient way to work with pyALF (and Python in general) is through Jupyter Notebooks. These are interactively usable documents that combine source code, results and narration (through Markdown) in one file. pyALF includes example notebooks, online available from here, or by cloning the pyALF repository.

The canonical way to use the Jupyter Notebooks, is through a JupyterLab, which can for example be installed via pip (for more details see here):

```
pip install jupyterlab
```

A JupyterLab can then be started with the shell command `jupyter-lab`, which launches a web server that should be automatically opened in your default browser.

Another convenient way to work with the notebooks is with Visual Studio Code, a versatile and extendable source-code editor.

1.6 Ready-to-use container image

For a ready-to-use environment, one can use the Docker image alfcollaboration/jupyter-pyalf-full, which has the above mentioned dependencies, ALF and pyALF installed. With a suitable container runtime e.g. Docker or Podman, it can be used to run ALF and pyALF without any further setup. It is derived from the Jupyter Docker Stacks, therefore this documentation applies. For example, one could run a container like this:

```
docker run -it --rm -p 127.0.0.1:8888:8888 -v "$PWD":/home/jovyan/work \
    docker.io/alfcollaboration/jupyter-pyalf-full
```

[1] It is a lesser known fact that git is completely decentralized and the concept of a central repository is rather only a convention. Every git repository is an autonomous repository of itself. If, for example, pyALF has been cloned to `/path/to/ALF`, one could clone this repository with `git clone /path/to/ALF`.

- The -p flag is used to expose port 8888 and you can access a JupyterLab running within the container by navigating to `http://localhost:8888/lab?token=<token>` with you browser, where `<token>` has to be replaced by the token echoed to the terminal on startup.
- The -v flag mounts the current working directory to `/home/jovyan/work` within the container, allowing to work on the same data in- and outside of the container.
- The –rm flag instructs Docker to automatically remove the container after it exits, avoiding cluttering up the system with unused containers.
- The -i and -t flags keep the container's `STDIN` open and attach a pseudo-terminal, allowing interactive input on the terminal.

It is also possible to use the container without launching the included JupyterLab. The following command launches a container, which executes `minimal_ALF_run`, saving the results in the current working directory and removing the container right after that.

```
docker run -it --rm -v "$PWD":/home/jovyan/work \
    docker.io/alfcollaboration/jupyter-pyalf-full \
    bash -c 'cd /home/jovyan/work && minimal_ALF_run'
```

1.7 Some SSH port forwarding applications

ALF simulations are often performed on remote clusters that are accessed via SSH. Notably, SSH can be used for much more than running a remote shell. In this section, I will show how one can use SSH port forwarding to download data to HPC clusters with restrictive firewalls and how to access a JupyterLab launched on an HPC cluster.

1.7.1 Use remote forwarding to circumvent restrictive firewalls

If one wanted to git clone the ALF source code, this could usually be done with one of the following commands, using HTTPS or SSH, respectively.

```
git clone https://github.com/ALF-QMC/ALF.git
git clone git@github.com:ALF-QMC/ALF.git
```

But on some systems with very restrictive firewalls, this approach might not work. This is where the ssh option -R might come in handy. It maps a port on the remote machine to a an address connected to from the local machine on which the SSH command was executed. To facilitate a connection to `github.com`, the following commands can be used, connecting to port 443 or 22, for the HTTPS or SSH protocol, respectively.

```
ssh -R <PortNum>:github.com:443 <username>@<servername>
ssh -R <PortNum>:github.com:22 <username>@<servername>
```

Here `<PortNum>` refers to a port on the remote machine, a value in the range from 49152 to 65535 would be best here [3]. And `<username>@<servername>` is the usual SSH address. Alternatively to the command line option `-R`, the SSH config file option RemoteForward can be used.

With these port forwarding options, the ALF source code can then be cloned on the remote machine with:

```
git clone -c http.sslVerify=false https://localhost:<PortNum>/ALF-QMC/ALF.git
git clone ssh://git@localhost:<PortNum>/ALF-QMC/ALF.git
```

The HTTPS version needs the option `-c http.sslVerify=false` because the SSL certificate for `github.com` does not apply to `localhost`.

One can omit the host value in the `-R` option (in the example above `github.com:443`) which will set up a dynamic SOCKS proxy, able to connect to arbitrary addresses. This can be used, for example, to download and install packages with `pip`.

Warning

Ports on the remote machine opened with `-R` / `RemoteForward` can not only be used by you, but possibly also by other users of the machine. Therefore one should be careful when using the options, in particular without specifying a host.

Using `-R` without a host to install pyALF with pip:

```
ssh -R <PortNum> <username>@<servername>
```

That pip can use the SOCKS proxy, the python package `pysocks` is necessary. If the package is not yet available, it is enough to get the file `socks.py` from here and have Python find it, e.g. with the environment variable `PYTHONPATH`.

Then pyALF can be installed with:

```
pip install --proxy socks4://localhost:<PortNum> pyALF
```

1.7.2 Using Jupyter via SSH tunnel

When launching JupyterLab, it sets up a webserver and prints out how to access it locally, like:

```
http://localhost:<remote_port_number>/lab?token=<token>
```

Where `<remote_port_number>` is some port number (default 8888) and `<token>` is the password to access the server.

Now, to access this web server on the remote machine, one can forward this port to the local machine using the SSH option -L and open it with the browser.

```
ssh -L <local_port_number>:localhost:<remote_port_number> <username>@<servername>
```

With the command from above, a remote JupyterLab will be accessible through the address `http://localhost:<local_port_number>:/lab?token=<token>`.

1.7.3 Using SSH in Visual Studio Code

Here, a reference to use ssh in Visual Studio Code is provided: https://code.visualstudio.com/docs/remote/ssh

CHAPTER

TWO

USAGE

This section demonstrates how to use pyALF through small examples that can be directly executed, if everything has been set up as described in Section 1. It first shows on a minimal example how to run an ALF simulation and get some results. Then the different features of pyALF are expanded in more detail.

- *Minimal example*
- *Compiling and running ALF*
- *Postprocessing*
  - *Basic analysis*
  - *Custom/Derived Observables*
  - *Checking warmup and autocorrelation times*
  - *Symmetrization of correlations on the lattice*
- *Command line tools*

For a reference on all features, see Section 3.

Tip

The Python builtin `help()` is very useful for getting information on an object. Try e.g. `help(Simulation)` after importing `Simulation` from `py_alf`.

2.1 Minimal example

In this bare-bones example we simulate the Hubbard model with default the default presets: a 6×6 square grid, with interaction strength $U = 4$ and inverse temperature $\beta = 5$.

Bellow we go through the steps for performing the simulation and outputting observables.

---

1. Import `ALF_source` and `Simulation` classes from the `py_alf` python module, which provide the interface with ALF:

```
from py_alf import ALF_source, Simulation  # Interface with ALF
```

2. Create an instance of `ALF_source`, downloading the ALF source code from the ALF repository, if `alf_dir` does not exist. Gets `alf_dir` from environment variable `$ALF_DIR`, or defaults to `"./ALF"`, if not present:

```
alf_src = ALF_source()
```

3. Create an instance of `Simulation`, overwriting default parameters as desired:

```
sim = Simulation(
    alf_src,
    "Hubbard",                      # Name of Hamiltonian
    {                               # Dictionary overwriting default parameters
        "Lattice_type": "Square"
    },
    machine='GNU'  # Change to "intel", or "PGI" if gfortran is not installed
)
```

4. Compile ALF. The first time it will also download and compile HDF5, which could take ∼15 minutes.

```
sim.compile()
```

```
Compiling ALF...
Cleaning up Prog/
Cleaning up Libraries/
Cleaning up Analysis/
Compiling Libraries
```

```
Compiling Analysis
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
Link program
Done.
```

```
Compiling Analysis
```

```
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
```

```
Link program
```

```
Done.
```

5. Perform the simulation as specified in `sim`:

```
sim.run()
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪Square" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

6. Perform some simple analysis:

```
sim.analysis()
```

```
### Analyzing /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪Square ###
/home/jonas/Programs/pyALF/doc/source/usage
Scalar observables:
Ener_scal
Kin_scal
Part_scal
Pot_scal
Histogram observables:
Equal time observables:
Den_eq
Green_eq
SpinT_eq
SpinXY_eq
SpinZ_eq
Time displaced observables:
Den_tau
Green_tau
SpinT_tau
SpinXY_tau
SpinZ_tau
```

7. Read analysis results into a Pandas Dataframe with one row per simulation, containing parameters and observables:

```
obs = sim.get_obs()
```

```
/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_Square
No orbital locations saved.
```

```
obs
```

```
                                                  continuous  ham_chem  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...           0       0.0

                                                  ham_t  ham_t2  ham_tperp  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...   1.0     1.0        1.0
```

(continues on next page)

(continued from previous page)

```
                                                    ham_u  ham_u2  mz  l1  l2  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...    4.0     4.0   1   6   6

                                                    ...  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  ...

                                                                                 ␣
 ↪      SpinXY_tauK_err  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  [[0.2262872250454745, 0.
 ↪3768545985400847, 0.01...

                                                                                 ␣
 ↪          SpinXY_tauR  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  [[0.05759880037574172, -0.
 ↪10378742200159607, 0...

                                                                                 ␣
 ↪      SpinXY_tauR_err  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  [[0.012488346667982868, 0.
 ↪03687128937818297, 0...

                                                                                 ␣
 ↪   SpinXY_tau_lattice  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  {'L1': [6.0, 0.0], 'L2': [0.
 ↪0, 6.0], 'a1': [1....

                                                                                 ␣
 ↪           SpinZ_tauK  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  [[0.904003636203397, 0.
 ↪562819189602052, 0.6101...

                                                                                 ␣
 ↪       SpinZ_tauK_err  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  [[0.14947867503119427, 0.
 ↪06006336638594533, 0....

                                                                                 ␣
 ↪           SpinZ_tauR  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  [[0.10134420531394392, -0.
 ↪1144555239159215, 0....

                                                                                 ␣
 ↪       SpinZ_tauR_err  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  [[0.06273438448506144, 0.
 ↪05639078043063978, 0....

                                                                                 ␣
 ↪    SpinZ_tau_lattice  \
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  {'L1': [6.0, 0.0], 'L2': [0.
 ↪0, 6.0], 'a1': [1....

                                                                                 ␣
 ↪              lattice
/home/jonas/Programs/pyALF/doc/source/usage/ALF...  {'L1': [6.0, 0.0], 'L2': [0.
 ↪0, 6.0], 'N_coord'...

[1 rows x 111 columns]
```

• The internal energy of the system (and its error) are accessed by:

```
obs.iloc[0][['Ener_scal0', 'Ener_scal0_err', 'Ener_scal_sign', 'Ener_scal_sign_err
```

(continues on next page)

(continued from previous page)

```
↪']]
```

```
Ener_scal0           -29.821914
Ener_scal0_err          0.13032
Ener_scal_sign              1.0
Ener_scal_sign_err          0.0
Name: /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_Square,␣
 ↪dtype: object
```

Warning

While it is very easy to get some results, as demonstrated right now, there are many caveats with using QMC, and a naive approach will quickly lead to wrong results.

Three of those caveats, namely numerical stability, warmup and autocorrelation will later be briefly addressed. For more details, please refer to the ALF documentation.

- The simulation can be resumed by calling `sim.run()` again, increasing the precision of results:

```
sim.run()
sim.analysis()
obs2 = sim.get_obs()
obs2.iloc[0][['Ener_scal0', 'Ener_scal0_err', 'Ener_scal_sign', 'Ener_scal_sign_
 ↪err']]
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪Square" for Monte Carlo run.
Resuming previous run.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
### Analyzing /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪Square ###
/home/jonas/Programs/pyALF/doc/source/usage
Scalar observables:
Ener_scal
Kin_scal
Part_scal
Pot_scal
Histogram observables:
Equal time observables:
Den_eq
Green_eq
SpinT_eq
SpinXY_eq
SpinZ_eq
Time displaced observables:
Den_tau
Green_tau
SpinT_tau
SpinXY_tau
SpinZ_tau
/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_Square
No orbital locations saved.
```

```
Ener_scal0           -29.609245
```

(continues on next page)

(continued from previous page)

```
Ener_scal0_err         0.136803
Ener_scal_sign              1.0
Ener_scal_sign_err          0.0
Name: /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_Square,␣
↪dtype: object
```

```
### Analyzing /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
↪Square ###
/home/jonas/Programs/pyALF/doc/source/usage
Scalar observables:
Ener_scal
Kin_scal
Part_scal
Pot_scal
Histogram observables:
Equal time observables:
Den_eq
Green_eq
SpinT_eq
SpinXY_eq
SpinZ_eq
Time displaced observables:
Den_tau
Green_tau
SpinT_tau
SpinXY_tau
SpinZ_tau
/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_Square
No orbital locations saved.
```

```
Ener_scal0           -29.609245
Ener_scal0_err         0.136803
Ener_scal_sign              1.0
Ener_scal_sign_err          0.0
Name: /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_Square,␣
↪dtype: object
```

```
print(f"""Running again changed the error
from {obs.iloc[0]['Ener_scal0_err']}
to {obs2.iloc[0]['Ener_scal0_err']}""")
```

```
Running again changed the error
from 0.13032001866398857
to 0.13680286325153657
```

The error was not actually reduced as expected, hinting at problems with e.g. warmup, autocorrelation, or fat tails.

2.2 Compiling and running ALF

This section focuses on the "ALF interface" part of pyALF, i.e. how to compile ALF and run ALF simulations. This revolves around the classes `ALF_source` and `Simulation` defined in the module `py_alf` that have already been briefly introduced in Section 2.1.

We start with some imports:

```
from pprint import pprint  # Pretty print

from py_alf import ALF_source, Simulation  # Interface with ALF
```

2.2.1 Class `ALF_source`

The Class *py_alf.ALF_source* points to a folder containing the ALF source code. It has the following signature:

```
class ALF_source(
    alf_dir=os.getenv('ALF_DIR', './ALF'),
    branch=None,
    url='https://github.com/ALF-QMC/ALF.git'
)
```

Where `os.getenv('ALF_DIR', './ALF')` gets the environment variable `$ALF_DIR` if present and otherwise returns `'./ALF'`. If the directory `alf_dir` does exist, the program assumes it contains the ALF source code and will raise an Exception if that is not the case. If `alf_dir` does not exist, the source code will be cloned form `url`. If `branch` is set, git checks it out.

We will just use the default:

```
alf_src = ALF_source()
```

And see if it successfully found ALF:

```
alf_src.alf_dir
```

```
'/home/jonas/Programs/ALF'
```

We can use the function *py_alf.ALF_source.get_ham_names()* to see which Hamiltonians are implemented:

```
alf_src.get_ham_names()
```

```
['Kondo',
 'Hubbard',
 'Hubbard_Plain_Vanilla',
 'tV',
 'LRC',
 'Z2_Matter',
 'Spin_Peierls']
```

And then view the list of parameters and their default values for a particular Hamiltonian. The Hamiltonian-specific parameters are listed first, followed by the Hamiltonian-independent parameters.

```
pprint(alf_src.get_default_params('Hubbard'))
```

2.2.2 Class `Simulation`

To set up a simulation, we create an instance of *py_alf.Simulation*, which has the signature

```
class Simulation(alf_src, ham_name, sim_dict, **kwargs)
```

where `alf_src` is an instance of *py_alf.ALF_source*, `ham_name` is the name of the Hamiltonian to simulate, `sim_dict` is a dictionary of `parameter: value` pairs overwriting the default parameters and `**kwargs` represents optional keyword arguments.

The minimal set of required arguments does not overwrite any default parameters:

```
sim = Simulation(alf_src, 'Hubbard', {})
```

Before running the simulation, ALF needs to be compiled.

```
sim.compile()
```

```
Compiling ALF...
Cleaning up Prog/
Cleaning up Libraries/
Cleaning up Analysis/
Compiling Libraries
```

```
Compiling Analysis
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
Link program
Done.
```

```
Compiling Analysis
```

```
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
```

```
Link program
Done.
```

Preparation of the simulation is done by executing the following command:

```
sim.run()
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard
 ↪" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
```

(continues on next page)

(continued from previous page)

```
ALF Copyright (C) 2016 - 2022 The ALF project contributors
This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
This is free software, and you are welcome to redistribute it under certain␣
↪conditions.
No initial configuration
```

It is strongly advised to take a look at the info file `info` produced by ALF after a finished run, in particular the value of "Precision Green" and "Precision Phase". As a rule of thumb, the means should be of order 10^{-8} or smaller and the max should not be bigger than 10^{-3}. If they're bigger, one should decrease the stabilization interval `Nwrap` (see parameter list `'VAR_QMC'` above). In our case, they're about right.

```
sim.print_info_file()
```

```
===== /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard/info =====
 ====================================
 Model is      : Hubbard
 Lattice is    : Square
 # unit cells  :           36
 # of orbitals :            1
 Flux_1        :    0.0000000000000000
 Flux_2        :    0.0000000000000000
 Twist as phase factor in bulk
 HS  couples to z-component of spin
 Checkerboard  :  T
 Symm. decomp  :  T
 Finite temperture version
 Beta          :    5.0000000000000000
 dtau,Ltrot_eff:   0.10000000000000001               50
 N_SUN         :            2
 N_FL          :            2
 t             :    1.0000000000000000
 Ham_U         :    4.0000000000000000
 Ham_chem      :    0.0000000000000000
 No initial configuration, Seed_in      790789
 Sweeps                              :           20
 Bins                                :            5
 No CPU-time limitation
 Measure Int.                        :            1          50
 Stabilization,Wrap                  :           10
 Nstm                                :            5
 Ltau                                :            1
 # of interacting Ops per time slice :           36
 Default sequential updating
 This executable represents commit 76e12b0e of branch master.
  Precision Green  Mean, Max :    2.5417763978441579E-011   2.2375287356268814E-
↪007
  Precision Phase, Max       :    0.0000000000000000
  Precision tau    Mean, Max :    5.5915761887550577E-012   5.9682744746325511E-
↪008
  Acceptance                 :   0.42958333333333332
  Effective Acceptance       :   0.42958333333333332
  CPU Time                   :    2.7690582980000000
```

2.2.3 Specifying parameters

Here is an example of a simulation with non-default parameters. We have changed the dimensions to 4 by 4 sites and increased the interaction U to 4.0 and the number of bins calculated to 20. Since we did not change the compile-time configuration (some of the `**kwargs` do), a recompilation is not required.

```
sim = Simulation(
    alf_src,
    'Hubbard',
    {
        # Model specific parameters
        'L1': 4,
        'L2': 4,
        'Ham_U': 4.0,
        # QMC parameters
        'Nbin': 20,
    },
)
sim.run()
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
↪L1=4_L2=4_U=4.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

Note that the new simulation has been placed in `ALF_data/Hubbard_L1=4_L2=4_U=4.0` relative to the current working directory. That is, simulations are placed in the folder `{sim_root}/{sim_dir}`, where `sim_root` defaults to `'ALF_data'` and `sim_dir` is generated out of the Hamiltonian name and the non-default model specific parameters. A behavior that can be overwritten through the `**kwargs`. Note that `Nbin` does not enter `sim_dir`, since it is a QMC parameter and not a Hamiltonian parameter.

The monitoring in the info file does not show any stabilization issues:

```
sim.print_info_file()
```

```
===== /home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_L1=4_L2=4_
 ↪U=4.0/info =====
 ====================================
 Model is      : Hubbard
 Lattice is    : Square
 # unit cells  :           16
 # of orbitals :            1
 Flux_1        :    0.0000000000000000
 Flux_2        :    0.0000000000000000
 Twist as phase factor in bulk
 HS  couples to z-component of spin
 Checkerboard  :  T
 Symm. decomp  :  T
 Finite temperture version
 Beta          :    5.0000000000000000
 dtau,Ltrot_eff:   0.10000000000000001               50
 N_SUN         :            2
 N_FL          :            2
 t             :    1.0000000000000000
 Ham_U         :    4.0000000000000000
 Ham_chem      :    0.0000000000000000
```

(continues on next page)

(continued from previous page)

```
No initial configuration, Seed_in      790789
Sweeps                              :           20
Bins                                :           20
No CPU-time limitation
Measure Int.                        :            1          50
Stabilization,Wrap                  :           10
Nstm                                :            5
Ltau                                :            1
# of interacting Ops per time slice :           16
Default sequential updating
This executable represents commit 76e12b0e of branch master.
 Precision Green  Mean, Max :    3.0903935060176945E-011   2.9012717530502163E-
↪007
 Precision Phase, Max       :    0.0000000000000000
 Precision tau    Mean, Max :    7.6475551634990995E-012   1.3561035078213379E-
↪007
 Acceptance                 :   0.42600781250000003
 Effective Acceptance       :   0.42600781250000003
 CPU Time                   :    3.1458244400000002
```

2.2.4 Series of MPI runs

Starting each run separately can be cumbersome, therefore we provide the following example, which creates a list of `Simulation` instances that can be run in a loop, performing a sweep in U. To increase the statistics of the results, MPI parallelization is employed. Since the default MPI executable `mpiexec` does not fit with the MPI libraries used during compilation on the test machine, we have changed it to `orterun`. The option `mpiexec_args=['--oversubscribe']` hands over the flag `--oversubscribe` to `orterun`, which allows it to run more MPI tasks than there are slots available, see the Open MPI documentation for details.

```
sims = [
    Simulation(
        alf_src,
        'Hubbard',
        {
            # Model specific parameters
            'L1': 4,
            'L2': 4,
            'Ham_U': U,
            # QMC parameters
            'Nbin': 20,
        },
        mpi=True,
        n_mpi=4,
        # mpiexec='orterun',
        # mpiexec_args=['--oversubscribe'],
    )
    for U in [1.0, 2.0, 3.0]]
```

```
sims
```

```
[<py_alf.simulation.Simulation at 0x72695d0ee210>,
 <py_alf.simulation.Simulation at 0x72695f354b00>,
 <py_alf.simulation.Simulation at 0x7269a954da70>]
```

Note

The above employs Python's list comprehensions, a convenient and readable way to create Python lists. Here is a simple example, employing list comprehension (and f-strings):

```
>>> [f'x={x}' for x in [1, 2, 3]]
['x=1', 'x=2', 'x=3']
```

Since we are changing from non-MPI to MPI, ALF has to be recompiled:

Warning

pyALF does not check how ALF has been compiled previously, so the user has to take care of issuing recompilation if necessary.

```
sims[0].compile()
```

```
Compiling ALF...
Cleaning up Prog/
Cleaning up Libraries/
Cleaning up Analysis/
Compiling Libraries
```

```
Compiling Analysis
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
Link program
Done.
```

```
Compiling Analysis
```

```
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
```

(continues on next page)

(continued from previous page)

```
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
```

```
Link program
```

```
Done.
```

Loop over list of jobs:

```
for sim in sims:
    sim.run()
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪L1=4_L2=4_U=1.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪L1=4_L2=4_U=2.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪L1=4_L2=4_U=3.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

```
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪L1=4_L2=4_U=2.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
```

```
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
↪L1=4_L2=4_U=3.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
```

```
ALF Copyright (C) 2016 - 2022 The ALF project contributors
This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
This is free software, and you are welcome to redistribute it under certain␣
↪conditions.
No initial configuration
```

```
for sim in sims:
    sim.print_info_file()
```

2.2.5 Parallel Tempering

ALF offers the possibility to employ Parallel Tempering [4], also known as Exchange Monte Carlo [5], where simulations with different parameters but the same configuration space are run in parallel and can exchange configurations. A method developed to overcome ergodicity issues.

To use Parallel Tempering in pyALF, `sim_dict` has to be replaced by a list of dictionaries, for this we use again Python's list comprehension syntax. This does also imply `mpi=True`, since Parallel Tempering needs MPI.

```
sim = Simulation(
    alf_src,
    'Hubbard',
    [
        {
            # Model specific parameters
            'L1': 4,
            'L2': 4,
            'Ham_U': U,
            # QMC parameters
            'Nbin': 20,
            'mpi_per_parameter_set': 2
        } for U in [2.5, 3.5]
    ],
    mpi=True,
    n_mpi=4,
    # mpiexec='orterun',
    # mpiexec_args=['--oversubscribe'],
)
```

Recompile for Parallel Tempering:

```
sim.compile()
```

```
Compiling ALF...
Cleaning up Prog/
Cleaning up Libraries/
Cleaning up Analysis/
Compiling Libraries
```

```
Compiling Analysis
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
```

(continues on next page)

(continued from previous page)

```
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
Link program
Done.
```

```
Compiling Analysis
```

```
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
```

```
Link program
```

```
Done.
```

```
sim.run()
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/temper_
 ↪Hubbard_L1=4_L2=4_U=2.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/temper_
 ↪Hubbard_L1=4_L2=4_U=2.5/Temp_0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/temper_
 ↪Hubbard_L1=4_L2=4_U=2.5/Temp_1" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

```
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
```

(continues on next page)

(continued from previous page)

```
This is free software, and you are welcome to redistribute it under certain␣
↪conditions.
No initial configuration
```

```
sim.print_info_file()
```

The output from this command has been omitted for brevity.

2.2.6 Only preparing runs

In many cases, it might not be feasible to execute ALF directly through pyALF, for example when using an HPC scheduler, but one might still like to use pyALF for preparing the simulation directories. In this case the two options `copy_bin` and `only_prep` of *`py_alf.Simulation.run()`* come in handy. Here we also demonstrate the keyword arguments `sim_root` and `sim_dir`.

```
import numpy as np

JK_list = np.linspace(0.0, 3.0, num=11)
print(JK_list)

sims = [
    Simulation(
        alf_src,
        'Kondo',
        {
            "Model": "Kondo",
            "Lattice_type": "Bilayer_square",
            "L1": 16,
            "L2": 16,
            "Ham_JK": JK,
            "Ham_Uf": 1.,
            "Beta": 20.0,
            "Nsweep": 500,
            "NBin": 400,
            "Ltau": 0,
            "CPU_MAX": 48
        },
        mpi=True,
        sim_root="KondoBilayerSquareL16",
        sim_dir=f"JK{JK:2.1f}",
    ) for JK in JK_list
]
```

```
[0.  0.3 0.6 0.9 1.2 1.5 1.8 2.1 2.4 2.7 3. ]
```

Do not forget to recompile when switching from Parallel Tempering back to normal MPI runs.

```
sims[0].compile()
```

```
Compiling ALF...
Cleaning up Prog/
Cleaning up Libraries/
Cleaning up Analysis/
Compiling Libraries
```

```
Compiling Analysis
Compiling Program
Parsing Hamiltonian parameters
```

(continues on next page)

(continued from previous page)

```
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
Link program
Done.
```

```
Compiling Analysis
```

```
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
```

```
Link program
```

```
Done.
```

```
for sim in sims:
    sim.run(copy_bin=True, only_prep=True)
```

```
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK0.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK0.3" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK0.6" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK0.9" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK1.2" for Monte Carlo run.
```

(continues on next page)

(continued from previous page)

```
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK1.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK1.8" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK2.1" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK2.4" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK2.7" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/
 ↪KondoBilayerSquareL16/JK3.0" for Monte Carlo run.
Create new directory.
```

Now there are 11 directories, ready for the job scheduler.

```
!tree KondoBilayerSquareL16
```

```
KondoBilayerSquareL16
├── JK0.0
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK0.3
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK0.6
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK0.9
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK1.2
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK1.5
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK1.8
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK2.1
│   ├── ALF.out
│   ├── parameters
│   └── seeds
├── JK2.4
│   ├── ALF.out
│   ├── parameters
│   └── seeds
```

(continues on next page)

(continued from previous page)

```
├── JK2.7
│   ├── ALF.out
│   ├── parameters
│   └── seeds
└── JK3.0
    ├── ALF.out
    ├── parameters
    └── seeds

12 directories, 33 files
```

2.3 Postprocessing

The following sections demonstrate the postprocessing features in pyALF, each section can be executed individually, if QMC raw data from Section 2.2 is present.

- *Basic analysis*
- *Custom/Derived Observables*
- *Checking warmup and autocorrelation times*
- *Symmetrization of correlations on the lattice*

2.3.1 Basic analysis

As already shown in Section 2.1, the basic analysis can be executed through *py_alf.Simulation.analysis()*, which in turn calls `py_alf.analysis()`. This section demonstrates how to directly use the latter function and how to access and work with analysis results.

As a first step, some libraries and functions are imported. The Jupyter magic command `%matplotlib widget` enables the Matplotlib Jupyter Widget Backend, which is not necessary in this part, but for the functions used in Section 2.3.3, therefore it makes sense to establish it as a default.

```
# Enable Matplotlib Jupyter Widget Backend
%matplotlib widget

# Imports
import matplotlib.pyplot as plt  # Plotting library
import numpy as np  # Numerical libary

from py_alf.ana import load_res  # Function for loading analysis results
from py_alf.analysis import analysis  # Analysis function
from py_alf.utils import find_sim_dirs  # Function for finding QMC bins
```

The function *find_sim_dirs()* returns a list of all directories containing a file named `data.h5`, the file containing all QMC measurements if ALF has been compiled with HDF5. We use it to conveniently list all simulations run in the previous sections.

```
dirs = find_sim_dirs()
dirs
```

```
['./ALF_data/Hubbard',
 './ALF_data/Hubbard_L1=4_L2=4_U=1.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=2.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=3.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=4.0',
 './ALF_data/Hubbard_Square',
```

(continues on next page)

(continued from previous page)

```
 './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_0',
 './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_1']
```

Looping over this list, we call `analysis()` for each directory. The function reads QMC bins from `data.h5`, or if this file does not exist alternatively from all files ending in `_scal`, `_eq` and `_tau`. Furthermore, `n_skip` and `n_rebin` are read from the file `parameters`. The analysis omits the first `n_skip` bins and combines `n_rebin` original bins into a new one[1]. On the resulting bins, Jackknife resampling [6] is applied to estimate expectation values and their standard error.

```
for directory in dirs:
    analysis(directory)
```

2.3.1.1 Get analysis results

The analysis results are saved in each simulation directory, both in plain text in the folder `res` and as a `pickled` Python dictionary in the file `res.pkl`.

The binary data from multiple `res.pkl` files can be conveniently read with *`load_res()`*, which returns a single `pandas DataFrame`, a tabular data structure. It not only contains analysis results, but also the Hamiltonian-specific parameters. The parameter names are in all lower case.

```
res = load_res(dirs)
```

```
./ALF_data/Hubbard
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=1.0
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=2.0
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=3.0
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=4.0
No orbital locations saved.
./ALF_data/Hubbard_Square
No orbital locations saved.
./ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_0
No orbital locations saved.
./ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_1
No orbital locations saved.
```

The DataFrame has one row per simulation directory, which is also used as the index:

```
res.index
```

```
Index(['./ALF_data/Hubbard', './ALF_data/Hubbard_L1=4_L2=4_U=1.0',
       './ALF_data/Hubbard_L1=4_L2=4_U=2.0',
       './ALF_data/Hubbard_L1=4_L2=4_U=3.0',
       './ALF_data/Hubbard_L1=4_L2=4_U=4.0', './ALF_data/Hubbard_Square',
       './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_0',
       './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_1'],
      dtype='object')
```

Column indices can be accessed through:

```
res.columns
```

[1] We will elaborate further on rebinning in Section 2.3.3.

```
Index(['continuous', 'ham_chem', 'ham_t', 'ham_t2', 'ham_tperp', 'ham_u',
       'ham_u2', 'mz', 'l1', 'l2',
       ...
       'SpinZ_tauK', 'SpinZ_tauK_err', 'SpinZ_tauR', 'SpinZ_tauR_err',
       'SpinZ_tau_lattice', 'lattice', 'Acc_Temp_scal_sign',
       'Acc_Temp_scal_sign_err', 'Acc_Temp_scal0', 'Acc_Temp_scal0_err'],
      dtype='object', length=115)
```

In the following, we will only use results from one simulation, corresponding to one row in the DataFrame. It is selected with:

```
item = res.loc['./ALF_data/Hubbard']
```

Which is equivalent to

```
item = res.iloc[0]
```

Most, but not all of the same data is also stored in plain text form in the folder `ALF_data/Hubbard/res`:

```
!ls --color -p ALF_data/Hubbard/res
```

```
Den_eq_K      Green_eq_K      Part_scal       SpinT_tau/       SpinZ_eq_K
Den_eq_K_sum  Green_eq_K_sum  Pot_scal        SpinXY_eq_K      SpinZ_eq_K_sum
Den_eq_R      Green_eq_R      SpinT_eq_K      SpinXY_eq_K_sum  SpinZ_eq_R
Den_eq_R_sum  Green_eq_R_sum  SpinT_eq_K_sum  SpinXY_eq_R      SpinZ_eq_R_sum
Den_tau/      Green_tau/      SpinT_eq_R      SpinXY_eq_R_sum  SpinZ_tau/
Ener_scal     Kin_scal        SpinT_eq_R_sum  SpinXY_tau/
```

2.3.1.1.1 Scalar observables

Scalar observable results are stored as multiple scalar values, storing the sign, observable expectation value and their statistical errors. Here are, for example, the results for the internal energy `Ener_scal`, consisting of four scalar values:

```
for i in item.index:
    if i.startswith('Ener_scal'):
        print(i, item[i])
```

```
Ener_scal_sign 1.0
Ener_scal_sign_err 0.0
Ener_scal0 -29.821914139694446
Ener_scal0_err 0.13032001866398857
```

Note the `0` in `Ener_scal0` and `Ener_scal0_err`. This is the index in the vector of observables `Ener_scal`, since a scalar observable can hold a vector of scalars.

The same data is present in this plain text file:

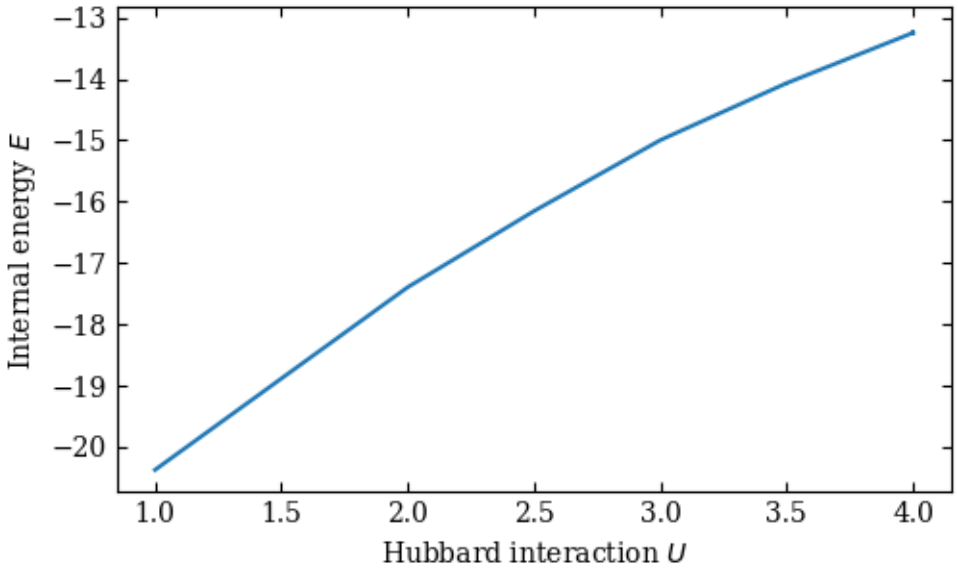
```
!cat ALF_data/Hubbard/res/Ener_scal
```

```
# Sign: 1.0 0.0
-2.98219141396944582e+01 1.30320018663988568 3e-01
```

2.3.1.1.1.1 Example

Here is a simple example that demonstrates the convenience of working with pandas DataFrames. We select out of all simulations the one with $L_1 = 4$ and plot their internal energy against The value of the Hubbard U.

```
# Create figure with axis labels
fig, ax = plt.subplots()
ax.set_xlabel(r'Hubbard interaction $U$')
ax.set_ylabel(r'Internal energy $E$')

# Select only rows with l1==4 and sort by ham_u
df = res[res.l1 == 4].sort_values(by='ham_u')

# Plot data
ax.errorbar(df.ham_u, df.Ener_scal0, df.Ener_scal0_err);
```



2.3.1.1.2 Equal-time correlation functions

ALF and pyALF offer support for correlation functions of the form

$$\begin{aligned} C(r, n_1, n_2) &= \frac{1}{N_r} \sum_{r_0} \langle O(r_0, n_1) O(r_0 + r, n_2) \rangle - \langle O(n_1) \rangle \langle O(n_2) \rangle \\ C(k, n_1, n_2) &= \frac{1}{N_r} \sum_{r} e^{ikr} C(r, n_1, n_2) \end{aligned} \tag{2.1}$$

Where the sums go over the unit cells of the finite size Bravais lattice, N_r is the number of unit cells and n_1, n_2 denominate the orbitals within a unit cell.

Each observable produces a set of members in the results, these are for example the ones for the equal-time Green's function:

```
for i in item.index:
    if i.startswith('Green_eq'):
        print(i, np.shape(item[i]))
```

```
Green_eqK (1, 1, 36)
Green_eqK_err (1, 1, 36)
Green_eqK_sum (36,)
Green_eqK_sum_err (36,)
```

(continues on next page)

(continued from previous page)

```
Green_eqR (1, 1, 36)
Green_eqR_err (1, 1, 36)
Green_eqR_sum (36,)
Green_eqR_sum_err (36,)
Green_eq_lattice ()
```

Members ending in `K`, `K_err`, `R` and `R_err` correspond to Eq. (2.1) and their errors. They have the shape $(N_{\mathrm{orb}}, N_{\mathrm{orb}}, N_r)$, where N_{orb} is the number of orbitals per unit cell. The objects ending in `_sum` have been traced over the orbital degrees of freedom. To correctly interpret the index over the unit cells, the member ending in `_lattice` is a dictionary containing the parameters for creating a Bravais lattice object *py_alf.Lattice*:

```
item['Green_eq_lattice']
```

```
{'L1': array([6., 0.]),
 'L2': array([0., 6.]),
 'a1': array([1., 0.]),
 'a2': array([0., 1.])}
```

```
from py_alf import Lattice

latt = Lattice(item['Green_eq_lattice'])
```

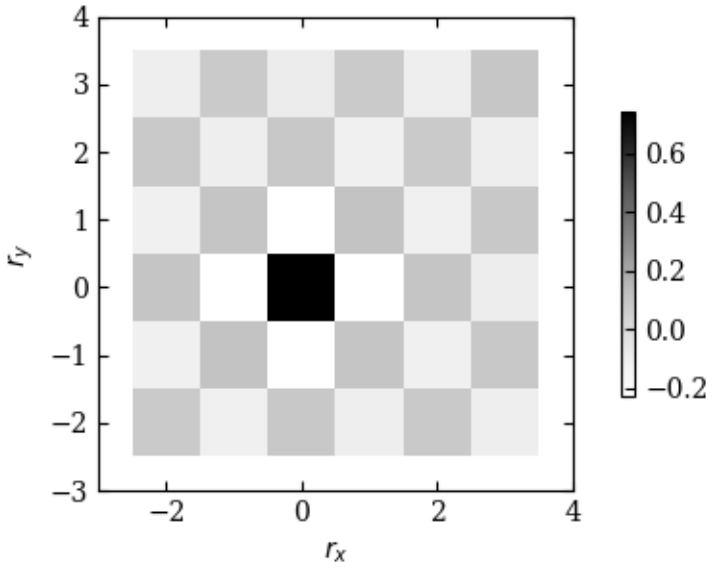
Here is, for example, the equal-time Greens function at $k = (\pi, \pi)$ with its error:

```
n = latt.k_to_n([np.pi, np.pi])
print(item.Green_eqK_sum[n], item.Green_eqK_sum_err[n])
```

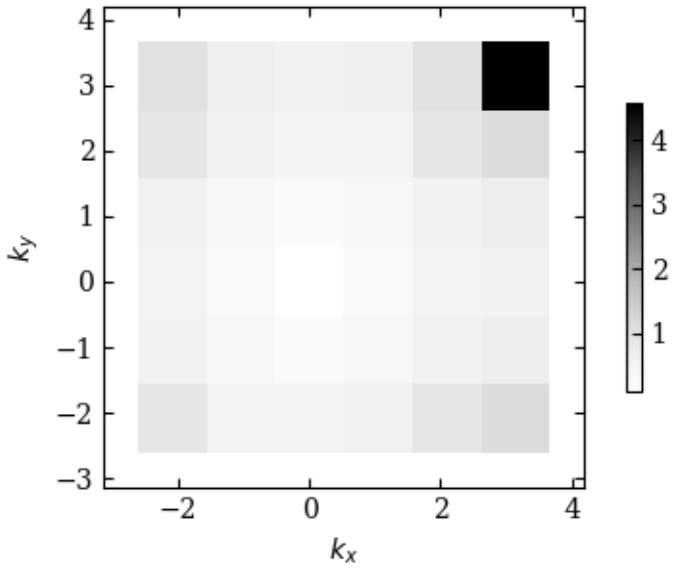
```
0.06526692607778947 0.001304607020942857
```

The lattice object offers functions for conveniently plotting correlation functions in real and momentum space. Below, we plot the Spin-Spin correlations in real and momentum space, showing signs of antiferromagnetic order.

```
latt.plot_r(item.SpinZ_eqR_sum)
```




```
latt.plot_k(item.SpinZ_eqK_sum)
```



The plain text result files ending in `_K` and `_R` contain momentum and real-space resolved correlations, respectively. Here is an excerpt from the Greens function in momentum space:

```
!head -n 3 ALF_data/Hubbard/res/Green_eq_K
```

```
#    kx       ky                          (0, 0)                                ␣
 ↪trace over n_orb
-2.09440 -2.09440         1.4417820888e-01  6.0737564928e-03         1.
 ↪4417820888e-01  6.0737564928e-03
-2.09440 -1.04720         1.0106239637e+00  1.1329334950e-02         1.
 ↪0106239637e+00  1.1329334950e-02
```

Where `(0, 0)` refers to the orbital indices. Since there is only one orbital per unit cell, this is the only combination and identical to the trace over all orbitals. The first two columns represent the coordinates, followed by alternating expectation values and standard errors.

2.3.1.1.3 Time-displaced correlation functions

The structure for time-displaced correlation functions is very similar to equal-time correlations, but by default only the trace over the orbital degrees of freedom is stored. These are the results for the time-displaced Green function:

```
for i in item.index:
    if i.startswith('Green_tau'):
        print(i, np.shape(item[i]))
```

```
Green_tauK (51, 36)
Green_tauK_err (51, 36)
Green_tauR (51, 36)
Green_tauR_err (51, 36)
Green_tau_lattice ()
```

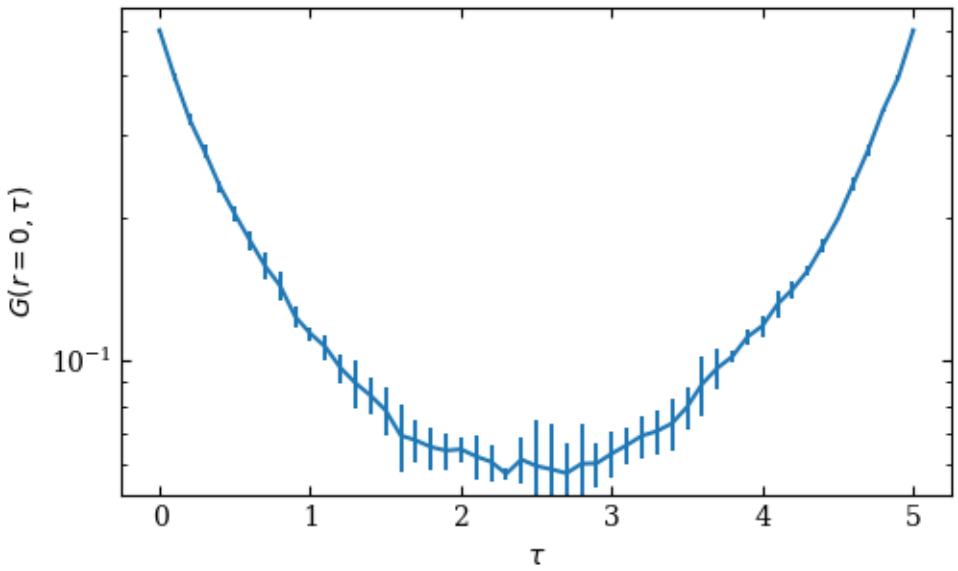
Here we plot the time-displaced Greens function at $r = 0$:

```
# Create figure with axis labels and logscale on y-axis
fig, ax = plt.subplots()
ax.set_xlabel(r'$\tau$')
ax.set_ylabel(r'$G(r=0, \tau)$')
ax.set_yscale('log')
```

(continues on next page)

(continued from previous page)

```
# Create lattice object
latt = Lattice(item['Green_tau_lattice'])

# Get index corresponding to r=0
n = latt.r_to_n([0, 0])

# Plot data
ax.errorbar(
    item.dtau*range(len(item.Green_tauR[:, n])),
    item.Green_tauR[:, n],
    item.Green_tauK_err[:, n]
);
```



Again, plain text results data are available in the folder `res`. There is a separate folder for each k-point and the data for $r = 0$:

```
!ls --color -p ALF_data/Hubbard/res/Green_tau
```

```
0.00_0.00/   1.05_0.00/    1.05_-2.09/   -2.09_1.05/   -2.09_3.14/  3.14_3.14/
0.00_-1.05/  -1.05_-1.05/  1.05_2.09/    2.09_-1.05/   2.09_3.14/   R0
0.00_1.05/   -1.05_1.05/   -1.05_3.14/   2.09_1.05/    3.14_0.00/
0.00_-2.09/  1.05_-1.05/   1.05_3.14/    -2.09_-2.09/  3.14_-1.05/
0.00_2.09/   1.05_1.05/    -2.09_0.00/   -2.09_2.09/   3.14_1.05/
0.00_3.14/   -1.05_-2.09/  2.09_0.00/    2.09_-2.09/   3.14_-2.09/
-1.05_0.00/  -1.05_2.09/   -2.09_-1.05/  2.09_2.09/    3.14_2.09/
```

The data is in the following format with tree columns: τ, expectation value and error:

```
!head ALF_data/Hubbard/res/Green_tau/0.00_0.00/dat
```

```
     0.0000000       0.03348685       0.00547912
     0.1000000       0.01686925       0.00585814
     0.2000000       0.01793592       0.00935758
     0.3000000       0.01575110       0.00906065
     0.4000000       0.01096207       0.00657646
     0.5000000       0.00372588       0.00710389
     0.6000000       0.01155079       0.00842672
     0.7000000       0.00473281       0.01066049
     0.8000000       0.00177665       0.00989675
     0.9000000       0.00691428       0.00589819
```

2.3.2 Custom/Derived Observables

The previous section showed how to use the observables defined directly in the ALF simulation, but one often needs quantities derived from these. pyALF offers a convenient way for getting results for such derived observables, including a way to check for warmup and autocorrelation issues (more on the latter in the next section).

As usual, we start with some imports:

```
# Enable Matplotlib Jupyter Widget Backend
%matplotlib widget

import matplotlib.pyplot as plt  # Plotting library
import numpy as np  # Numerical libary

from py_alf.ana import load_res  # Function for loading analysis results
from py_alf.analysis import analysis  # Analysis function
from py_alf.utils import find_sim_dirs  # Function for finding QMC bins
```

Create list with directories to analyze:

```
dirs = find_sim_dirs()
dirs
```

```
['./ALF_data/Hubbard',
 './ALF_data/Hubbard_L1=4_L2=4_U=1.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=2.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=3.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=4.0',
 './ALF_data/Hubbard_Square',
 './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_0',
 './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_1']
```

The custom observables are defined in a Python dictionary, where the keys are the names of the new observables. The value is another dictionary in the format:

```
{'needs': some_list,
 'function': some_function,
 'kwargs': some_dict,}
```

Where `some_list` is a list of observable names, this can be any combination of scalar, equal-time, or time-displaced observables. They are being read by *`py_alf.ana.ReadObs`*. These Jackknife bins as well as *kwargs* from `some_dict` are handed to `some_function` with a separate call for each bin. Currently, only scalars and 1d arrays are supported as return value of `some_function`. We go through some examples to make this procedure clearer.

We start with an empty dictionary, which will hold all the custom observable definitions:

```
custom_obs = {}
```

The first custom observable will just be the square of the energy. For this, we define a function taking three arguments, which correspond to one jackknifed bin from *`py_alf.ana.read_scal()`*:

- `obs`: Array of observable values
- `sign`: Float
- `N_obs`: Length of `obs`, in this case 1.

The next step is to add an entry to `custom_obs`. The name of the new observable shall be *E_squared*. It needs the observable *Ener_scal*, the function defined previously, and we don't hand over any keyword arguments.

```
def obs_squared(obs, sign, N_obs):
    """Square of a scalar observable.

    obs.shape = (N_obs,)
    """
    return obs[0]**2 / sign

# Energy squared
custom_obs['E_squared']= {
    'needs': ['Ener_scal'],
    'function': obs_squared,
    'kwargs': {}
}
```

Another custom observable shall be the potential energy divided by kinetic energy. The approach is similar to before, except that this now uses two observables *Pot_scal* and *Kin_scal*:

```
def E_pot_kin(E_pot_obs, E_pot_sign, E_pot_N_obs,
              E_kin_obs, E_kin_sign, E_kin_N_obs):
    """Ratio of two scalar observables, first observable divided by second."""
    return E_pot_obs/E_kin_obs / (E_pot_sign/E_kin_sign)

# Potential Energy / Kinetic Energy
custom_obs['E_pot_kin']= {
    'needs': ['Pot_scal', 'Kin_scal'],
    'function': E_pot_kin,
    'kwargs': {}
}
```

Finally, we want to calculate some correlation ratios. A correlation ratio is a renormalisation group invariant quantity, that can be a powerful tool for identifying ordered phases and phase transitions. It is defined as:

$$R(O, k_*) = 1 - \frac{O(k_* + \delta)}{O(k_*)} \tag{2.2}$$

Where $O(k)$ is a correlation function that has a divergence at $k = k_*$ in the ordered phase and δ scales with $1/L$, where L is the linear system size. A usual choice for δ is the smallest k on the finite-sized Bravais lattice. With these properties, $R(O, k_*)$ will take only one of two values in the thermodynamic limit: 0 in the unordered phase and 1 in the ordered phase.

The above can be generalized, to an average over multiple singular points k_i and distances from those points δ_j, which results in:

$$R = \frac{1}{N_k} \sum_{i=1}^{N_k} \left(1 - \frac{\frac{1}{N_\delta} \sum_{j=1}^{N_\delta} O(k_i + \delta_j)}{O(k_i)} \right) \tag{2.3}$$

Furthermore, the correlation function might have an orbital structure to be considered:

$$O(k) = \sum_{n,m} \tilde{O}(k)_{n,m} M_{n,m} \tag{2.4}$$

All in all, this can be expressed in a function like this:

```
def R_k(obs, back, sign, N_orb, N_tau, dtau, latt,
        ks=((0., 0.),), mat=None, NNs=((1, 0), (0, 1), (-1, 0), (0, -1))):
    """Calculate correlation ratio, an RG-invariant quantity derived from
    a correlation function.

    Parameters
    ----------
    obs : array of shape (N_orb, N_orb, N_tau, latt.N)
        Correlation function, the background is already subtracted.
```

(continues on next page)

(continued from previous page)

```
    back : array of shape (N_orb,)
        Background of Correlation function.
    sign : float
        Monte Carlo sign.
    N_orb : int
        Number of orbitals per unit cell.
    N_tau : int
        Number of imaginary time slices. 1 for equal-time correlations.
    dtau : float
        Imaginary time step.
    latt : py_alf.Lattice
        Bravais lattice object.
    ks : list of k-points, default=((0., 0.),)
        Singular points of the correlation function in the intended order.
    mat : array of shape (N_orb, N_orb), default=None
        Orbital structure of the order parameter. Default: Trace over orbitals.
    NNs : list of tuples, default=((1, 0), (0, 1), (-1, 0), (0, -1))
        Deltas in terms of primitive k-vectors of the Bravais lattice.

    """
    if mat is None:
        mat = np.identity(N_orb)
    out = 0
    for k in ks:
        n = latt.k_to_n(k)

        J1 = (obs[..., n].sum(axis=-1) * mat).sum()
        J2 = 0
        for NN in NNs:
            i = latt.nnlistk[n, NN[0], NN[1]]
            J2 += (obs[..., i].sum(axis=-1) * mat).sum() / len(NNs)
        out += (1 - J2/J1)

    return out / len(ks)
```

This function works for both equal-time and time-displaced correlations. The first 7 arguments (`obs, back, sign, N_orb, N_tau, dtau, latt`) are supplied by `analysis()` if a correlation function is requested in *needs*. The optional keyword arguments specify the singular k points, the orbital structure and δ_j to be considered.

Correlation ratios for ferromagnetic and antiferromagnetic order can now be defined with:

```
# RG-invariant quantity for ferromagnetic order
custom_obs['R_Ferro']= {
    'needs': ['SpinT_eq'],
    'function': R_k,
    'kwargs': {'ks': [(0., 0.)]}
}

# RG-invariant quantity for antiferromagnetic order
custom_obs['R_AFM']= {
    'needs': ['SpinT_eq'],
    'function': R_k,
    'kwargs': {'ks': [(np.pi, np.pi)]}
}
```

```
def obs_k(obs, back, sign, N_orb, N_tau, dtau, latt,
          ks=((0., 0.),), mat=None):
    """Mean of correlation function at one, or multiple k-points.

    Calculates integral over tau (=susceptibility) if time-displaced
    correlation is supplied.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------
    obs : array of shape (N_orb, N_orb, N_tau, latt.N)
        Correlation function, the background is already subtracted.
    back : array of shape (N_orb,)
        Background of Correlation function.
    sign : float
        Monte Carlo sign.
    N_orb : int
        Number of orbitals per unit cell.
    N_tau : int
        Number of imaginary time slices. 1 for equal-time correlations.
    dtau : float
        Imaginary time step.
    latt : py_alf.Lattice
        Bravais lattice object.
    ks : list of k-points, default=[(0., 0.)]
    mat : array of shape (N_orb, N_orb), default=None
        Orbital structure. Default: Trace over orbitals.

    """
    if mat is None:
        mat = np.identity(N_orb)
    out = 0
    for k in ks:
        n = latt.k_to_n(k)

        if N_tau == 1:
            out += (obs[:, :, 0, n] * mat).sum()
        else:
            out += (obs[..., n].sum(axis=-1) * mat).sum()*dtau

    return out / len(ks)

# Correlation of Spin z-component at k=(pi, pi)
custom_obs['SpinZ_pipi']= {
    'needs': ['SpinZ_eq'],
    'function': obs_k,
    'kwargs': {'ks': [(np.pi, np.pi)]}
}

# Correlation of Spin x+y-component at k=(pi, pi)
custom_obs['SpinXY_pipi']= {
    'needs': ['SpinXY_eq'],
    'function': obs_k,
    'kwargs': {'ks': [(np.pi, np.pi)]}
}

# Correlation of total Spin at k=(pi, pi)
custom_obs['SpinXYZ_pipi']= {
    'needs': ['SpinT_eq'],
    'function': obs_k,
    'kwargs': {'ks': [(np.pi, np.pi)]}
}
```

The same definitions for `custom_obs` are also written in the local file `custom_obs.py` to be used in further sections.

To now analyze with these custom observables, the dictionary has to be handed over as a keyword argument to `analysis()`. The analysis skips a directory by default if the QMC bins file `data.h5` and the parameter file `parameters` are both older than `res.pkl`, which is the case since `res.pkl` has been freshly created in the

previous section. Therefore, we use the option `always=True` to overwrite this behavior.

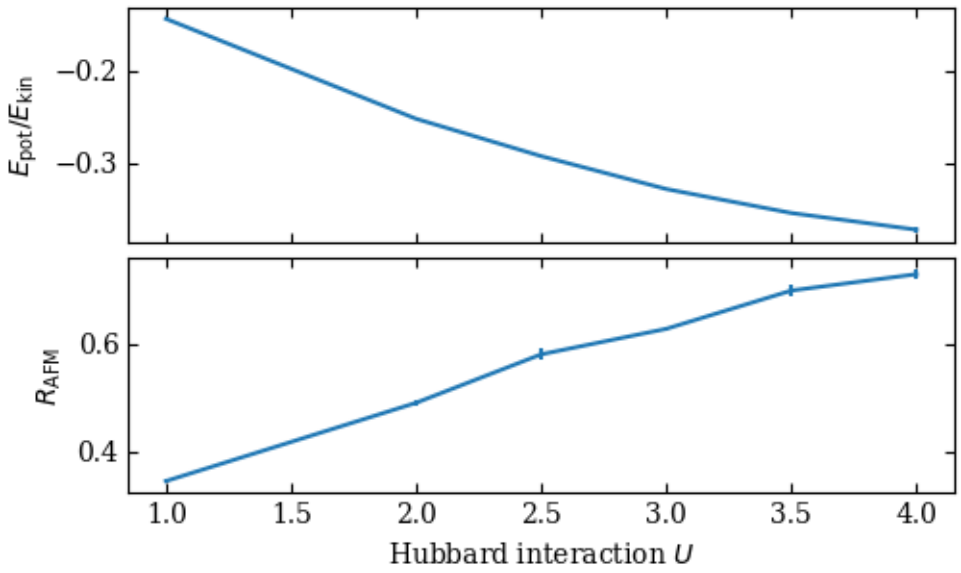
The results are loaded the same way as in the previous section:

```
res = load_res(dirs)
```

```
./ALF_data/Hubbard
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=1.0
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=2.0
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=3.0
No orbital locations saved.
./ALF_data/Hubbard_L1=4_L2=4_U=4.0
No orbital locations saved.
./ALF_data/Hubbard_Square
No orbital locations saved.
./ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_0
No orbital locations saved.
./ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_1
No orbital locations saved.
```

Access to the values is analogues to scalar observables:

```
# Create figure with two axes and axis labels
fig, (ax1, ax2) = plt.subplots(2, 1,
                               sharex=True,
                               constrained_layout=True)
ax1.set_ylabel(r'$E_{\rm pot} / E_{\rm kin}$')
ax2.set_ylabel(r'$R_{\rm AFM}$')
ax2.set_xlabel(r'Hubbard interaction $U$')

# Select only rows with l1==4 and sort by ham_u
df = res[res.l1 == 4].sort_values(by='ham_u')

# Plot data
ax1.errorbar(df.ham_u, df.E_pot_kin, df.E_pot_kin_err)
ax2.errorbar(df.ham_u, df.R_AFM, df.R_AFM_err);
```



2.3.3 Checking warmup and autocorrelation times

Two common challenges in Monte Carlo studies are ensuring that the measured bins represent **equilibrated** configurations and that different bins are **statistically independent**. In this section, we will briefly explain these issues and present the tools pyALF offers for dealing with them.

2.3.3.1 Preparations

As a first step, we use the same import as in previous sections.

```
# Enable Matplotlib Jupyter Widget Backend
%matplotlib widget

from py_alf.utils import find_sim_dirs  # Function for finding QMC bins
```

We also import the functions *py_alf.check_warmup()* and *py_alf.check_rebin()*, which play the main role in this section.

```
from py_alf import check_rebin, check_warmup
```

Finally, from the local file `custom_obs.py`, we import the same `custom_obs` defined in the previous section.

```
from custom_obs import custom_obs
```

For demonstration purposes, we run a simulation with very small bins.

```
from py_alf import ALF_source, Simulation

sim = Simulation(
    ALF_source(),
    'Hubbard',
    {
        # Model specific parameters
        'L1': 4,
        'L2': 4,
        'Ham_U': 5.0,
        # QMC parameters
        'Nbin': 5000,
        'Nsweep': 5,
        'Ltau': 0,
    },
)
sim.compile()
sim.run()
```

```
Compiling ALF...
Cleaning up Prog/
Cleaning up Libraries/
Cleaning up Analysis/
Compiling Libraries
```

```
Compiling Analysis
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
```

(continues on next page)

(continued from previous page)

```
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
Link program
Done.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪L1=4_L2=4_U=5.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

```
Compiling Analysis
```

```
Compiling Program
Parsing Hamiltonian parameters
filenames: Hamiltonians/Hamiltonian_Kondo_smod.F90 Hamiltonians/Hamiltonian_
 ↪Kondo_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_smod.F90 Hamiltonians/Hamiltonian_
 ↪Hubbard_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90 Hamiltonians/
 ↪Hamiltonian_Hubbard_Plain_Vanilla_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_tV_smod.F90 Hamiltonians/Hamiltonian_tV_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_LRC_smod.F90 Hamiltonians/Hamiltonian_LRC_
 ↪read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90 Hamiltonians/Hamiltonian_
 ↪Z2_Matter_read_write_parameters.F90
filenames: Hamiltonians/Hamiltonian_Spin_Peierls_smod.F90 Hamiltonians/
 ↪Hamiltonian_Spin_Peierls_read_write_parameters.F90
Compiling program modules
```

```
Link program
```

```
Done.
Prepare directory "/home/jonas/Programs/pyALF/doc/source/usage/ALF_data/Hubbard_
 ↪L1=4_L2=4_U=5.0" for Monte Carlo run.
Create new directory.
Run /home/jonas/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2022 The ALF project contributors
 This Program comes with ABSOLUTELY NO WARRANTY; for details see license.GPL
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
```

We set the directories to be considered.

```
dirs = find_sim_dirs()
dirs
```

```
['./ALF_data/Hubbard',
 './ALF_data/Hubbard_L1=4_L2=4_U=1.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=2.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=3.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=4.0',
 './ALF_data/Hubbard_L1=4_L2=4_U=5.0',
 './ALF_data/Hubbard_Square',
 './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_0',
 './ALF_data/temper_Hubbard_L1=4_L2=4_U=2.5/Temp_1']
```

2.3.3.2 Check warmup

A Monte Carlo simulation creates a time series of configurations through stochastic updates. Usually, measurements from a number of updates get combined in one so-called bin. In the case of ALF, `Nsweep` sweeps create one bin of measurements (for more details on updating procedures we refer to the ALF documentation). Usually, the simulation starts in a non-optimal state and it takes some time to reach equilibrium. Bins from this "warming up" period should be dismissed before calculating results. This is achieved by setting the variable `N_skip` in the file `parameters`, which will make the analysis omit the first `N_skip` bins.

> **Warning**
>
> Different observables can have different warmup and autocorrelation times. For example, charge degrees of freedom may equilibrate much faster than spin degrees of freedom. Or a sum of observables might have much shorter autocorrelation times than an individual observable, e.g. the total spin versus one spin component.

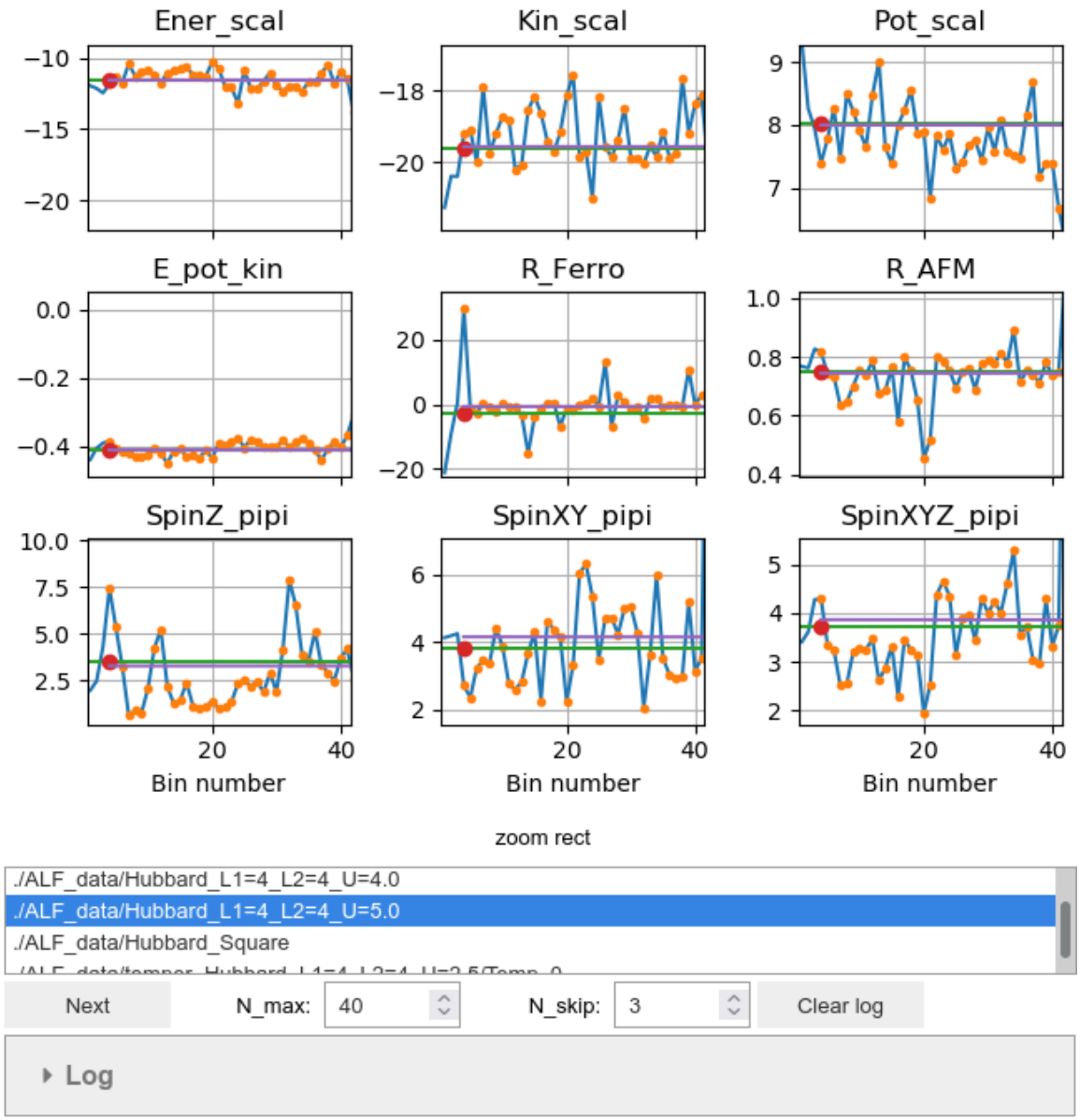
To judge the correct value for `N_skip`, pyALF offers the function *`check_warmup()`*, which plots the time series of bins for a given list of scalar and custom observables. It can be used with the previous simulations as:

```
warmup_widget = check_warmup(
    dirs,
    ['Ener_scal', 'Kin_scal', 'Pot_scal',
     'E_pot_kin', 'R_Ferro', 'R_AFM',
     'SpinZ_pipi', 'SpinXY_pipi', 'SpinXYZ_pipi'],
    custom_obs=custom_obs, gui='ipy'
)
```

The first argument is a list of directories containing simulations, the second argument specifies which observables to plot, the keyword argument `custom_obs` is needed, when plotting custom observables, e.g. *E_pot_kin* and `gui` specifies which GUI framework to use. With `gui='ipy'`, the function returns a Jupyter Widget, which allows to seamlessly work within Jupyter. Another option would be `gui='tk'`, which opens a separate window using `tkinter`. The latter option might be suitable when working directly from a shell.

The variable `N_skip` can be directly changed in the GUI, which automatically updates the file `parameters`.

```
warmup_widget
```



2.3.3.3 Check rebin

When estimating statistical errors, the analysis assumes different bins to be statistically independent. As a result, one bin must span over enough updates to generate statically independent configurations, or in other words, a bin must be larger than the autocorrelation time. Otherwise the statistical errors will be underestimated. To address this issue, the analysis employs so-called rebinning, which combines `N_rebin` bins into one new bin. The pyALF function *`check_rebin()`* helps in determining the correct `N_rebin`. It plots the errors of the chosen observables against `N_rebin`. With enough statistics, one should see growing errors with increasing `N_rebin` until a saturation point is reached, this saturation point marks a suitable value for `N_rebin`. The usage of *`check_rebin()`* is very similar to *`check_warmup()`*.

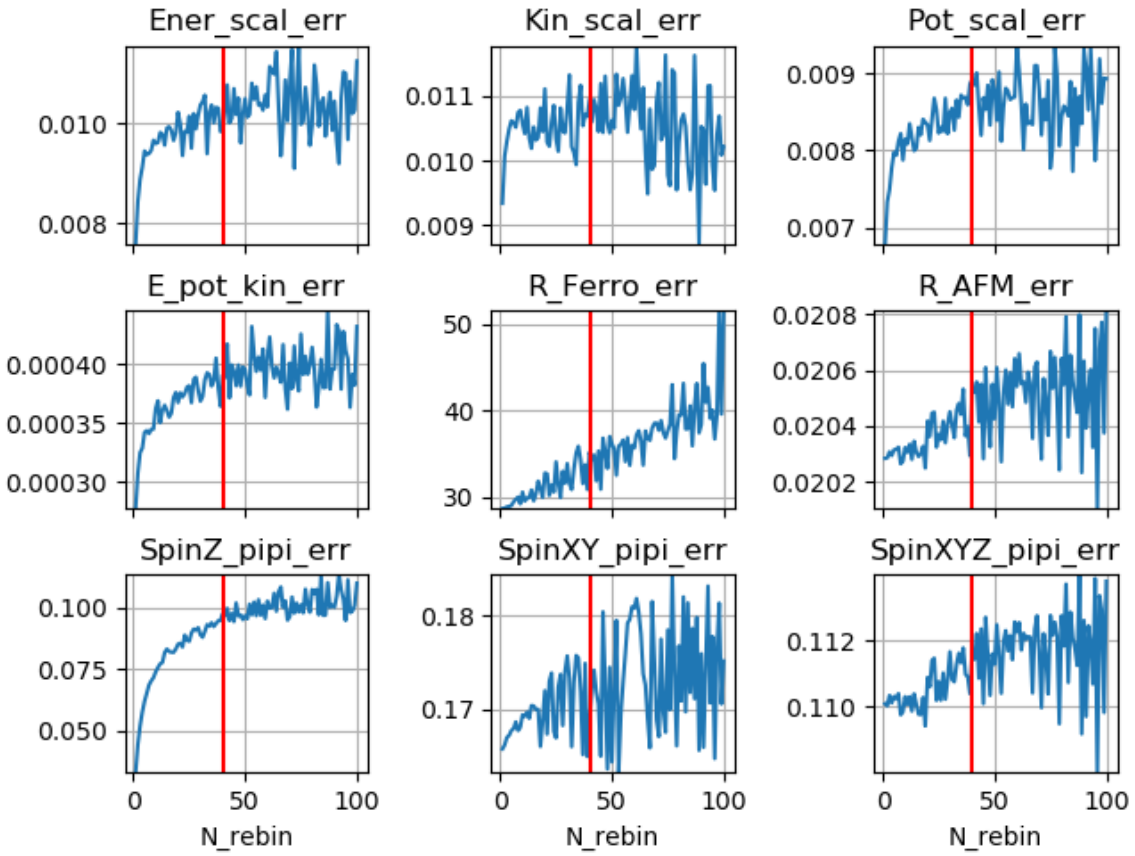
```
rebin_widget = check_rebin(
    dirs,
    ['Ener_scal', 'Kin_scal', 'Pot_scal',
     'E_pot_kin', 'R_Ferro', 'R_AFM',
     'SpinZ_pipi', 'SpinXY_pipi', 'SpinXYZ_pipi'],
    custom_obs=custom_obs, gui='ipy'
)
```

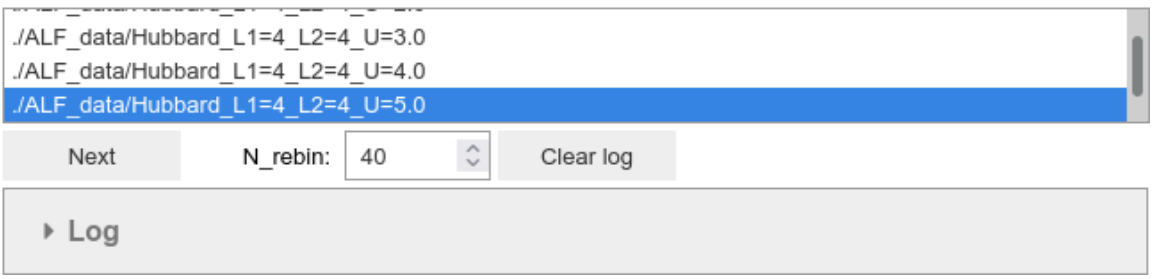
Below, we can see how different observables have different autocorrelation times. While the error of the kinetic energy saturates already at $N_{\text{rebin}} = 3$, the correlations of the z component of the spin at (π, π) (`SpinZ_pipi`) need $N_{\text{rebin}} \sim 40$. For the correlations of the total spin (`SpinXYZ_pipi`), on the other hand, $N_{\text{rebin}} = 1$ is sufficient.

The improvement from `SpinZ_pipi` to `SpinXYZ_pipi` is a good example for the concept of an improved estimator: The simulated Hubbard model is SU(2) symmetric, therefore correlations of the x, y and z components of the spin are equivalent, but with the chosen parameter `Mz=True` (cf. Section 2.2) the auxiliary field couples to the z component of the spin. As a result, the SU(2) symmetry is broken for an individual auxiliary field configuration, but restored by sampling the field configurations. Therefore, measuring the spin correlations through the z component, the x-y plane, or the full spin are in principle equivalent. But the latter option produces the most precise results and has the shortest autocorrelation times, because it explicitly restores the SU(2) symmetry instead of "waiting" for the sampling to do that.

Furthermore, the ferromagnetic correlation ratio R_{Ferro} doesn't seem to converge at all in the considered range of N_{rebin}. This is connected to the fact that the system is not close the ferromagnetic order and therefore R_{Ferro} is a bad observable.

```
rebin_widget
```





The next section will also show options for an improved estimator by employing symmetry operations on the Bravais

lattice.

2.3.4 Symmetrization of correlations on the lattice

The pyALF analysis offers an option to symmetrize correlation functions, by averaging over a list of symmetry operations on the Bravais lattice. This feature is meant to be used as an improved estimator, meaning to explicitly restore symmetries of the model lost due to imperfect sampling, which increases the quality of the data.

For this feature, the user has to supply a list of functions f_i, taking as arguments an instance of *py_alf.Lattice* and an integer corresponding to a k-point of the Bravais lattice and returning an integer corresponding to the transformed k-point of the Bravais lattice. The analysis then averages the correlation over all transformations:

$$\tilde{C}(n_k) = \frac{1}{N} \sum_{i=1}^{N} C\left(f_i(latt, n_k)\right)$$

Note

This symmetrization feature does not affect custom observables, but only the default analysis. Improved estimators would have to be included directly in the definition of custom observables.

The demonstration begins, as usual, with some imports:

```
# Enable Matplotlib Jupyter Widget Backend
%matplotlib widget

import matplotlib.pyplot as plt  # Plotting library
import numpy as np  # Numerical libary
from custom_obs import custom_obs  # Custom observable specifications

from py_alf import Lattice  # Defines Bravais lattice object
from py_alf.ana import load_res  # Function for loading analysis results
from py_alf.analysis import analysis  # Analysis function
                                      # from local file custom_obs.py
```

The Hubbard model on a square lattice possesses a fourfold rotation symmetry (= C_4 symmetry). To restore this symmetry, a list of all possible realizations of it has to be handed to the analysis. These are: rotation by 0 or 2π (= identity), rotation by $\pi/2$, rotation by π and rotation by $3\pi/2$.

```
# Define list of transformations (Lattice, i) -> new_i
# Default analysis will average over all listed elements
sym_c4 = [
    lambda latt, i : i,
    lambda latt, i : latt.rotate(i, np.pi*0.5),
    lambda latt, i : latt.rotate(i, np.pi),
    lambda latt, i : latt.rotate(i, np.pi*1.5),
]
```

We set the directory to be analyzed:

```
directory = './ALF_data/Hubbard'
```

We analyze without symmetrization and load results.

```
analysis(directory, symmetry=None, custom_obs=custom_obs, always=True)
res_nosym = load_res([directory]).iloc[0]
```

```
### Analyzing ./ALF_data/Hubbard ###
/home/jonas/Programs/pyALF/doc/source/usage
Custom observables:
custom E_squared ['Ener_scal']
custom E_pot_kin ['Pot_scal', 'Kin_scal']
custom R_Ferro ['SpinT_eq']
custom R_AFM ['SpinT_eq']
custom SpinZ_pipi ['SpinZ_eq']
custom SpinXY_pipi ['SpinXY_eq']
```

```
custom SpinXYZ_pipi ['SpinT_eq']
Scalar observables:
Ener_scal
Kin_scal
Part_scal
Pot_scal
Histogram observables:
Equal time observables:
Den_eq
Green_eq
SpinT_eq
SpinXY_eq
SpinZ_eq
Time displaced observables:
Den_tau
Green_tau
SpinT_tau
SpinXY_tau
SpinZ_tau
./ALF_data/Hubbard
No orbital locations saved.
```

Analyze with symmetrization and load results.

```
analysis(directory, symmetry=sym_c4, custom_obs=custom_obs, always=True)
res_sym = load_res([directory]).iloc[0]
```

```
### Analyzing ./ALF_data/Hubbard ###
/home/jonas/Programs/pyALF/doc/source/usage
Custom observables:
custom E_squared ['Ener_scal']
custom E_pot_kin ['Pot_scal', 'Kin_scal']
custom R_Ferro ['SpinT_eq']
custom R_AFM ['SpinT_eq']
custom SpinZ_pipi ['SpinZ_eq']
custom SpinXY_pipi ['SpinXY_eq']
custom SpinXYZ_pipi ['SpinT_eq']
Scalar observables:
Ener_scal
Kin_scal
Part_scal
Pot_scal
Histogram observables:
Equal time observables:
Den_eq
Green_eq
SpinT_eq
SpinXY_eq
```

```
SpinZ_eq
Time displaced observables:
```

(continues on next page)

(continued from previous page)

```
Den_tau
Green_tau
SpinT_tau
SpinXY_tau
SpinZ_tau
./ALF_data/Hubbard
No orbital locations saved.
```

We now compare results for the points $(\pi, \pi) + b_1$ and $(\pi, \pi) + b_2$, where $b_1 = (2\pi/L, 0)$ and $b_2 = (0, 2\pi/L)$ are the primitive vectors of the Bravais lattice in k-space, with and without symmetrization.

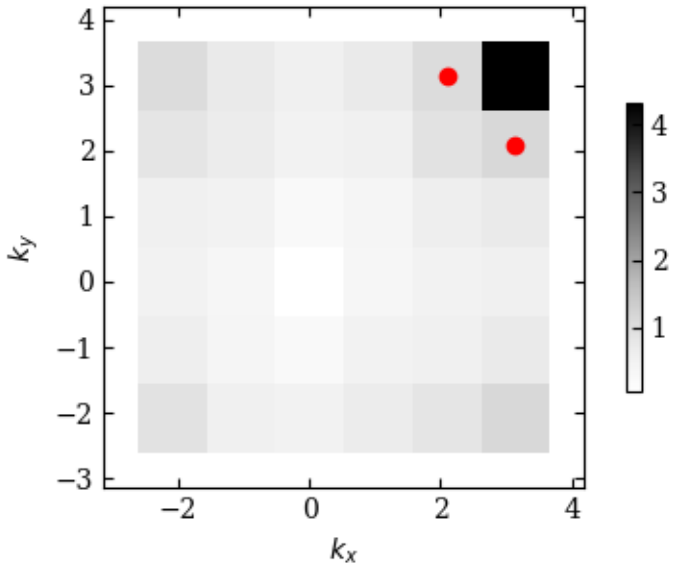
```
latt = Lattice(res_nosym['SpinT_eq_lattice'])
n = latt.k_to_n((np.pi, np.pi))
n1 = latt.nnlistk[n, -1, 0]
n2 = latt.nnlistk[n, 0, -1]
```

```
print(f"""Spin-Spin correlations:
Without symmetrization:
At k={latt.k[n1]}: {res_nosym.SpinT_eqK_sum[n1]:.2f} +- {res_nosym.SpinT_eqK_sum_
 ↪err[n1]:.2f}
At k={latt.k[n2]}: {res_nosym.SpinT_eqK_sum[n2]:.2f} +- {res_nosym.SpinT_eqK_sum_
 ↪err[n2]:.2f}

With symmetrization:
At k={latt.k[n1]}: {res_sym.SpinT_eqK_sum[n1]:.2f} +- {res_sym.SpinT_eqK_sum_
 ↪err[n1]:.2f}
At k={latt.k[n2]}: {res_sym.SpinT_eqK_sum[n2]:.2f} +- {res_sym.SpinT_eqK_sum_
 ↪err[n2]:.2f}
""")
```

```
Spin-Spin correlations:
Without symmetrization:
At k=[2.0943951  3.14159265]: 1.07 +- 0.04
At k=[3.14159265 2.0943951 ]: 1.14 +- 0.07

With symmetrization:
At k=[2.0943951  3.14159265]: 1.10 +- 0.05
At k=[3.14159265 2.0943951 ]: 1.10 +- 0.05
```

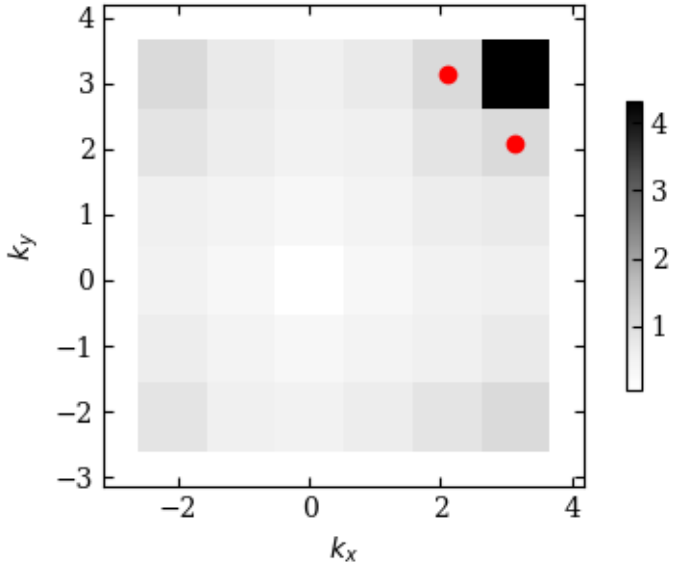
```
latt.plot_k(res_nosym.SpinT_eqK_sum)
plt.plot(*latt.k[n1], 'or')
plt.plot(*latt.k[n2], 'or')
```

```
[<matplotlib.lines.Line2D at 0x7f988d137b10>]
```




```
latt.plot_k(res_sym.SpinT_eqK_sum)
plt.plot(*latt.k[n1], 'or')
plt.plot(*latt.k[n2], 'or')
```

```
[<matplotlib.lines.Line2D at 0x7f988cf12990>]
```



2.4 Command line tools

In addition to the Python objects presented in previous sections, pyALF offers a set of scripts that make it easy to leverage pyALF from a Unix shell (e.g. Bash or zsh). They are located in the folder `py_alf/cli`, but installation via `pip` adds entry points to conveniently use them from the Unix shell without further configuration (notably, by trimming the trailing `.py`).

The list of all command line tools can be found in the *reference*. Out of those, this section will only introduce two more elaborate scripts, namely `alf_run` and `alf_postprocess`.

When starting a code line in Jupyter with an exclamation mark, the line will be interpreted as a shell command. We will use this feature to demonstrate the shell tools.

2.4.1 alf_run

The script `alf_run.py` enables most of the features displayed in Section 2.2 to be used directly from the shell. The help text lists all possible arguments:

```
!alf_run -h
```

```
usage: alf_run [-h] [--alfdir ALFDIR] [--sims_file SIMS_FILE]
               [--branch BRANCH] [--machine MACHINE] [--mpi] [--n_mpi N_MPI]
               [--mpiexec MPIEXEC] [--mpiexec_args MPIEXEC_ARGS]
               [--do_analysis]

Helper script for compiling and running ALF.

options:
  -h, --help            show this help message and exit
  --alfdir ALFDIR       Path to ALF directory. (default: os.getenv('ALF_DIR',
                        './ALF')
  --sims_file SIMS_FILE
                        File defining simulations parameters. Each line starts
                        with the Hamiltonian name and a comma, after wich
                        follows a dict in JSON format for the parameters. A
                        line that says stop can be used to interrupt.
                        (default: './Sims')
  --branch BRANCH       Git branch to checkout.
  --machine MACHINE     Machine configuration (default: 'GNU')
  --mpi                 mpi run
  --n_mpi N_MPI         number of mpi processes (default: 4)
  --mpiexec MPIEXEC     Command used for starting a MPI run (default:
                        'mpiexec')
  --mpiexec_args MPIEXEC_ARGS
                        Additional arguments to MPI executable.
  --do_analysis, --ana  Run default analysis after each simulation.
```

For example, to run a series of four different simulations of the Kondo model, the first step is to create a file specifying the parameters, with one line per simulation:

```
!cat Sims_Kondo
```

```
Kondo, {"L1": 4, "L2": 4, "Ham_JK": 0.5}
Kondo, {"L1": 4, "L2": 4, "Ham_JK": 1.0}
Kondo, {"L1": 4, "L2": 4, "Ham_JK": 1.5}
Kondo, {"L1": 4, "L2": 4, "Ham_JK": 2.0}
```

Then, one can execute `alf_run.py` with options as desired, the script automatically recompiles ALF for each simulation. For understanding some of the options, Section 2.2 might help.

```
!alf_run --sims_file ./Sims_Kondo --mpi --n_mpi 4
```

2.4.2 alf_postprocess

The script `alf_postprocess.py` enables most of the features discussed in Section 2.3, except for plotting capabilities, to be used directly from the shell. The help text lists all possible arguments:

```
!alf_postprocess -h
```

```
usage: alf_postprocess [-h] [--check_warmup] [--check_rebin]
                       [-l CHECK_LIST [CHECK_LIST ...]] [--do_analysis]
                       [--always] [--gather] [--no_tau]
```

(continues on next page)

(continued from previous page)

```
                      [--custom_obs CUSTOM_OBS] [--symmetry SYMMETRY]
                      [directories ...]

Script for postprocessing Monte Carlo bins.

positional arguments:
  directories           Directories to analyze. If empty, analyzes all
                        directories containing file "data.h5" it can find,
                        starting from the current working directory.

options:
  -h, --help            show this help message and exit
  --check_warmup, --warmup
                        Check warmup. Opens new window.
  --check_rebin, --rebin
                        Check rebinning for controlling autocorrelation. Opens
                        new window.
  -l, --check_list CHECK_LIST [CHECK_LIST ...]
                        List of observables to check for warmup and rebinning.
  --do_analysis, --ana  Do analysis.
  --always              Do not skip analysis if parameters and bins are older
                        than results.
  --gather              Gather all analysis results in one file named
                        "gathered.pkl", representing a pickled pandas
                        DataFrame.
  --no_tau              Skip time displaced correlations.
  --custom_obs CUSTOM_OBS
                        File that defines custom observables. This file has to
                        define the object custom_obs, needed by
                        py_alf.analysis. (default: os.getenv("ALF_CUSTOM_OBS",
                        None))
  --symmetry, --sym SYMMETRY
                        File that defines lattice symmetries. This file has to
                        define the object symmetry, needed by py_alf.analysis.
                        (default: None))
```

To use the symmetrization feature, one needs a file defining the object `symmetry`, similar to the already used file `custom_obs.py` defining `custom_obs`.

```
!cat sym_c4.py
```

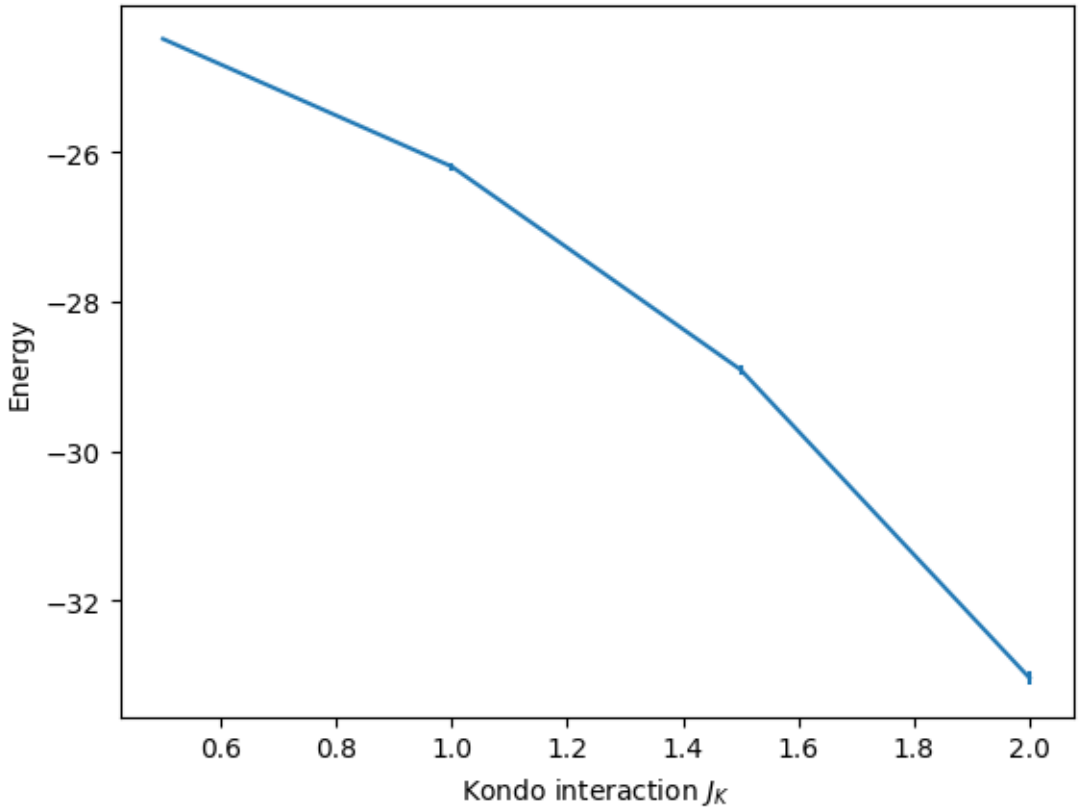
```
"""Define C_4 symmetry (=fourfold rotation) for pyALF analysis."""
from math import pi


# Define list of transformations (Lattice, i) -> new_i
# Default analysis will average over all listed elements
def sym_c4_0(latt, i): return i
def sym_c4_1(latt, i): return latt.rotate(i, pi*0.5)
def sym_c4_2(latt, i): return latt.rotate(i, pi)
def sym_c4_3(latt, i): return latt.rotate(i, pi*1.5)

symmetry = [sym_c4_0, sym_c4_1, sym_c4_2, sym_c4_3]
```

To analyze the results from the Kondo model and gather them all in one file `gathered.pkl`, we execute the following command.

```
!alf_postprocess --custom_obs custom_obs.py --symmetry sym_c4.py --ana --gather␣
↪ALF_data/Kondo*
```

The data from `gathered.pkl` can, for example, be read and plotted like this:

```
# Import modules
import matplotlib.pyplot as plt
import pandas as pd

# Load pickled DataFrame
res = pd.read_pickle('gathered.pkl')

# Create figure with axis labels
fig, ax = plt.subplots()
ax.set_xlabel(r'Kondo interaction $J_K$')
ax.set_ylabel(r'Energy')

# Plot data
ax.errorbar(res.ham_jk, res.Ener_scal0, res.Ener_scal0_err);
```



CHAPTER

THREE

REFERENCE

This is a reference of pyALF's features, most of the information in this section, except for the ones on the *Command line tools*, are also accessible through the Python builtin `help()`.

Table of contents

3.1 Class ALF_source

class py_alf.**ALF_source**(*alf_dir=None*, *branch=None*, *url='https://github.com/ALF-QMC/ALF.git'*)

Objet representing ALF source code.

Parameters

alf_dir
[path-like object, default=os.getenv('ALF_DIR', './ALF')] Directory containing the ALF source code. If the directory does not exist, the source code will be fetched from a server. Defaults to environment variable $ALF_DIR if defined, otherwise to './ALF'.

branch
[str, optional] If specified, this will be checked out by git.

url
[str, default='https://github.com/ALF-QMC/ALF.git'] Address from where to clone ALF if alf_dir does not exist.

get_default_params(*ham_name*, *include_generic=True*)

Return full set of default parameters for hamiltonian.

get_ham_names()

Return list of Hamiltonians.

get_params_names(*ham_name*, *include_generic=True*)

Return list of parameter names for hamiltonian, transformed in all uppercase.

3.2 Class Simulation

class py_alf.**Simulation**(*alf_src*, *ham_name*, *sim_dict*, ***kwargs*)

Object corresponding to an ALF simulation.

Parameters

alf_src
[ALF_source] Objet representing ALF source code.

ham_name
[str] Name of the Hamiltonian.

sim_dict
[dict or list of dicts] Dictionary specfying parameters owerwriting defaults. Can be a list of dictionaries to enable parallel tempering.

sim_dir
[path-like object, optional] Directory in which the Monte Carlo will be run. If not specified, sim_dir is generated from sim_dict.

sim_root
[path-like object, default="ALF_data"] Directory to prepend to sim_dir.

mpi
[bool, default=False] Employ MPI.

parallel_params
[bool, default=False] Run independent parameter sets in parallel. Based on parallel tempering, but without exchange steps.

n_mpi
[int, default=2] Number of MPI processes if mpi is true.

n_omp
[int, default=1] Number of OpenMP threads per process.

mpiexec
[str, default="mpiexec"] Command used for starting a MPI run. This may have to be adapted to fit with the MPI library used at compilation. Possible candidates include 'orterun', 'mpiexec.hydra'.

mpiexec_args
[list of str, optional] Additional arguments to MPI executable. E.g. the flag `--hostfile /path/to/file` is specified by `mpiexec_args=['--hostfile', '/path/to/file']`

machine
[{"GNU", "INTEL", "PGI", "Other machines defined in configure.sh"}] Compiler and environment.

stab
[str, optional] Stabilization strategy employed by ALF. Possible values: "STAB1", "STAB2", "STAB3", "LOG". Not case sensitive.

devel
[bool, default=False] Compile with additional flags for development and debugging.

hdf5
[bool, default=True] Whether to compile ALF with HDF5. Full postprocessing support only exists with HDF5.

analysis(*python_version=True*, ***kwargs*)

Perform default analysis on Monte Carlo data.

Calls `py_alf.analysis()`, if run with *python_version=True*.

Parameters

python_version
[bool, default=True] Use Python version of analysis. The non-Python version is legacy and does not support all postprocessing features.

****kwargs**
[dict, optional] Extra arguments for `py_alf.analysis()`, if run with *python_version=True*.

check_rebin(*names*, *gui='tk'*, ***kwargs*)

Plot error vs n_rebin to control autocorrelation.

Parameters

names
[list of str] Names of observables to check.

gui
[{'tk', 'ipy'}] Whether to use Tkinter or Jupyter Widget for GUI. default: 'tk'

****kwargs**
[dict, optional] Extra arguments for `py_alf.check_rebin_tk()` or `py_alf.check_rebin_ipy()`.

check_warmup(*names*, *gui='tk'*, ***kwargs*)

Plot bins to determine n_skip.

Parameters

names
[list of str] Names of observables to check.

gui
[{'tk', 'ipy'}] Whether to use Tkinter or Jupyter Widget for GUI. default: 'tk'

****kwargs**
[dict, optional] Extra arguments for `py_alf.check_warmup_tk()` or `py_alf.check_warmup_ipy()`.

compile(*verbosity=0*)

Compile ALF.

Parameters

verbosity
[int, default=0] 0: Don't echo make reciepes. 1: Echo make reciepes. else: Print make tracing information.

get_directories()

Return list of directories connected to this simulation.

get_obs(*python_version=True*)

Return Pandas DataFrame containing anaysis results from observables.

The non-python version is legacy and does not support all postprocessing features, e.g. time-displaced observables.

print_info_file()

Print info file(s) that get generated by ALF.

run(*copy_bin=False*, *only_prep=False*, *bin_in_sim_dir=False*)

Prepare simulation directory and run ALF.

Parameters

copy_bin
[bool, default=False] Copy ALF binary into simulation directory.

only_prep
[bool, default=False] Do not run ALF, only prepare simulation directory.

bin_in_sim_dir
[bool, default=False] Assume that the ALF binary is already present in simultation directory and use this.

3.3 High-level analysis functions

py_alf.analysis.**analysis**(*directory*, *symmetry=None*, *custom_obs=None*, *do_tau=True*, *always=False*)

Perform analysis in the given directory.

Results are written to the pickled dictionary *res.pkl* and in plain text in the folder *res/*.

Parameters

directory
[path-like object] Directory containing Monte Carlo bins.

symmetry
[list of functions, optional] List of functions reppresenting symmetry operations on lattice, including unity. It is used to symmetrize lattice-type observables.

custom_obs
[dict, default=None] Defines additional observables derived from existing observables. The key of each entry is the observable name and the value is a dictionary with the format:

```
{'needs': some_list,
 'kwargs': some_dict,
 'function': some_function,}
```

some_list contains observable names to be read by *py_alf.ana.ReadObs*. Jackknife bins and kwargs from *some_dict* are handed to *some_function* with a separate call for each bin.

do_tau
[bool, default=True] Analyze time-displaced correlation functions. Setting this to False speeds up analysis and makes result files much smaller.

always
[bool, default=False] Do not skip if parameters and bins are older than results.

py_alf.**check_warmup**(**args*, *gui='tk'*, ***kwargs*)

Plot bins to determine n_skip.

Calls either `py_alf.check_warmup_tk()` or `py_alf.check_warmup_ipy()`.

Parameters

***args**
gui
[{“tk”, “ipy”}]

****kwargs**

py_alf.check_warmup_tk.**check_warmup_tk**(*directories*, *names*, *custom_obs=None*)

Plot bins to determine n_skip. Opens a new window.

Parameters

directories
[list of path-like objects] Directories with bins to check.

names
[list of str] Names of observables to check.

custom_obs
[dict, default=None] Defines additional observables derived from existing observables. See `py_alf.analysis()`.

`py_alf.check_warmup_ipy.`**`check_warmup_ipy`**(*directories*, *names*, *custom_obs=None*, *ncols=3*)

Plot bins to determine n_skip in a Jupyter Widget.

Parameters

directories
[list of path-like objects] Directories with bins to check.

names
[list of str] Names of observables to check.

custom_obs
[dict, default=None] Defines additional observables derived from existing observables. See `py_alf.analysis()`.

Returns

Jupyter Widget
A graphical user interface based on ipywidgets

`py_alf.`**`check_rebin`**(**args*, *gui='tk'*, ***kwargs*)

Plot error vs n_rebin in a Jupyter Widget.

Calls either `py_alf.check_rebin_tk()` or `py_alf.check_rebin_ipy()`.

Parameters

***args**
gui
[{"tk", "ipy"}]

****kwargs**

`py_alf.check_rebin_tk.`**`check_rebin_tk`**(*directories*, *names*, *Nmax0=100*, *custom_obs=None*)

Plot error vs n_rebin. Opens a new window.

Parameters

directories
[list of path-like objects] Directories with bins to check.

names
[list of str] Names of observables to check.

Nmax0
[int, default=100] Biggest n_rebin to consider. The default is 100.

custom_obs
[dict, default=None] Defines additional observables derived from existing observables. See `py_alf.analysis()`.

`py_alf.check_rebin_ipy.`**`check_rebin_ipy`**(*directories*, *names*, *custom_obs=None*, *Nmax0=100*, *ncols=3*)

Plot error vs n_rebin in a Jupyter Widget.

Parameters

directories
[list of path-like objects] Directories with bins to check.

names
[list of str] Names of observables to check.

Nmax0
[int, default=100] Biggest n_rebin to consider. The default is 100.

custom_obs
[dict, default=None] Defines additional observables derived from existing observables. See `py_alf.analysis()`.

Returns

Jupyter Widget
A graphical user interface based on ipywidgets

3.4 Class Lattice

class py_alf.**Lattice**(**args*, *force_python_init=False*)

Finite size Bravais lattice object, mirroring ALF's Lattice type.

Parameters

***args**
[dict, tuple, or list] if dict: {'L1': L1, 'L2': L2, 'a1': a1, 'a2': a2}.

if tuple or list: [L1, L2, a1, a2].

L1, L2: 2d vectors defining periodic boundary conditions.

a1, a2: 2d primitive vectors.

force_python_init
[bool, default=False] Force the usage of Python version of the initialization. Default behavior is to first try compiled Fortran and fall back to Python if that fails.

Attributes

L1, L2
[arrays of floats] 2d vectors defining periodic boundary conditions in real space.

a1, a2
[arrays of floats] 2d primitive vectors in real space.

BZ1, BZ2
[arrays of floats] 2d vectors defining periodic boundary conditions in k space.

b1, b2
[arrays of floats] 2d primitive vectors in k space.

N
[int] Number of unit cells

r
[array of floats, shape=(N, 2)] Real-space coordinates.

k
[array of floats, shape=(N, 2)] K-space coordinates.

b1_perp
[array, shape=(2,)]

b2_perp
[array, shape=(2,)]

L
[int]

listr
[array of ints, shape=(N, 2)] *r[i] = listr[i, 0]*a1 + listr[i, 1]*a2*

listk
[array of ints, shape=(N, 2)] *k[i] = listk[i, 0]*a1 + listk[i, 1]*b2*

invlistr
[array of ints, shape=(2*L+1, 2*L+1)]

invlistk
[array of ints, shape=(2*L+1, 2*L+1)]

nnlistr
[array of ints, shape=(N, 3, 3)]

nnlistk
[array of ints, shape=(N, 3, 3)]

imj
[array of ints, shape=(N, N)]

fourier_K_to_R(*X*)

Fourier transform from k to r space.

Last index of input has to run over all lattice points in k space.

Last index of output runs over all lattice points in r space.

fourier_R_to_K(*X*)

Fourier transform from r to k space.

Last index of input has to run over all lattice points in r space.

Last index of output runs over all lattice points in k space.

k_to_n(*k*)

Map vector in k space to integer running over all lattice points.

periodic_boundary_k(*k*)

Apply periodic boundary conditions on vector in k space.

periodic_boundary_r(*r*)

Apply periodic boundary conditions on vector in r space.

plot_k(*data*)

Plot data in k space.

Parameters

data
[iterable] Index corresponds to coordinates.

plot_r(*data*)

Plot data in r space.

Parameters

data
[iterable] Index corresponds to coordinates.

r_to_n(*r*)

Map vector in r space to integer running over all lattice points.

rotate(*n*, *theta*)

Rotate vector in k space.

Parameters

n
[int] Index corresponding to input vector.

theta
[float] Angle of rotation.

Returns

int
Index corresponding to output vector.

3.5 Low-level analysis functions

Analysis routines.

class py_alf.ana.**Parameters**(*directory*, *obs_name=None*)

Object representing the "parameters" file.

Parameters

directory
[path-like object] Directory of "parameters" file.

obs_name
[str, optional] Observable name. If this is set, the object tries to get a parameters not from the namelist 'var_errors', but from a namelist called *obs_name*, while 'var_errors' is the fallback options. Parameters will be written to namelist *obs_name*.

N_min()

Get minimal number of bins, given the parameters in this object.

N_rebin()

Get N_rebin.

N_skip()

Get N_skip.

set_N_rebin(*parameter*)

Update N_rebin.

set_N_skip(*parameter*)

Update N_skip.

write_nml()

Write namelist to file. Preseves comments.

class py_alf.ana.**ReadObs**(*directory*, *obs_name*, *bare_bins=False*, *substract_back=True*)

Read, skip, rebin and jackknife scalar-type bins.

Bins get skipped and rebinned according to N_skip an N_rebin retrieved through *Parameters*, then jackknife resampling is applied. Saves jackknife bins.

Cf. *read_scal()*, *read_latt()*, *read_hist()*.

Parameters

directory
[path-like object] Directory containing the observable.

obs_name
[str] Name of observable.

bare_bins
[bool, default=False] Do not perform skipping, rebinning, or jackknife resampling.

substract_back
[bool, default=True] Substract background. Applies to correlation functions.

all()

Return all bins.

jack(*N_rebin*)

Return jackknife bins. Object has to be created with *bare_bins=True*.

Parameters

N_rebin
[int] Overwrite N_rebin from parameters.

slice(*n*)

Return n-th bin.

py_alf.ana.**ana_eq**(*directory*, *obs_name*, *sym=None*)

Analyze given equal-time correlators.

If sym is given, it symmetrizes the bins prior to calculating the error. Cf. *symmetrize()*.

py_alf.ana.**ana_hist**(*directory*, *obs_name*)

Analyze given histogram observables.

py_alf.ana.**ana_scal**(*directory*, *obs_name*)

Analyze given scalar observables.

Parameters

directory
[path-like object] Directory containing the observable.

obs_name
[str] Name of the observable.

py_alf.ana.**ana_tau**(*directory*, *obs_name*, *sym=None*)

Analyze given time-displaced correlators.

If sym is given, it symmetrizes the bins prior to calculating the error. Cf. *symmetrize()*.

py_alf.ana.**error**(*jacks*, *imag=False*)

Calculate expectation values and errors of given jackknife bins.

Parameters

jacks
[array-like object] Jackknife bins.

imag
[bool, default=False] Output with imaginary part.

Returns

tuple of numpy arrays
(expectation values, errors).

py_alf.ana.**jack**(*X*, *par*, *N_skip=None*, *N_rebin=None*)

Create jackknife bins out of input bins after skipping and rebinning.

Parameters

X
[array-like object] Input bins. Bins run over first index.

par
[*Parameters*] Parameters object.

N_skip
[int, default=par.N_skip()] Number of bins to skip.

N_rebin
[int, default=par.N_rebin()] Number of bins to recombine into one.

Returns

numpy array
Jackknife bins after skipping and rebinning.

py_alf.ana.**load_res**(*directories*)

Read analysis results from multiple simulations.

Read from pickled dictionaries 'res.pkl' and return everything in a single pandas DataFrame with one row per simulation.

Parameters

directories
[list of path-like objects] Directories containing analyzed simulation results.

Returns

df
[pandas.DataFrame] Contains analysis results and Hamiltonian parameters. One row per simulation.

py_alf.ana.**read_hist**(*directory*, *obs_name*, *bare_bins=False*)

Read, skip, rebin and jackknife histogram-type bins.

Bins get skipped and rebinned according to N_skip an N_rebin retrieved through *Parameters*, then jackknife resampling is applied.

Parameters

directory
[path-like object] Directory containing the observable.

obs_name
[str] Name of the observable.

bare_bins
[bool, default=False] Do not perform skipping, rebinning, or jackknife resampling.

Returns

array
Observables. shape: *(N_bins, N_classes)*.

array
Sign. shape: *(N_bins,)*.

array
Proportion of observations above upper bound. shape: *(N_bins,)*.

array
Proportion of observations below lower bound. shape: *(N_bins,)*.

N_classes
[int] Number of classes between upper and lower bound.

upper
[float] Upper bound.

lower
[float] Lower bound.

py_alf.ana.**read_latt**(*directory*, *obs_name*, *bare_bins=False*, *substract_back=True*)

Read, skip, rebin and jackknife lattice-type bins (_eq and _tau).

Bins get skipped and rebinned according to N_skip an N_rebin retrieved through *Parameters*, then jackknife resampling is applied.

Parameters

directory
[path-like object] Directory containing the observable.

obs_name
[str] Name of the observable.

bare_bins
[bool, default=False] Do not perform skipping, rebinning, or jackknife resampling.

substract_back
[bool, default=True] Substract background from correlation functions.

Returns

array
Observables. shape: *(N_bins, N_orb, N_orb, N_tau, latt.N)*.

array
Background. shape: *(N_bins, N_orb)*

array
Sign. shape: *(N_bins,)*.

N_orb
[int] Number of orbitals.

N_tau
[int] Number of imaginary time steps.

dtau
[float] Imaginary time step length.

latt
[Lattice] See *py_alf.Lattice*.

py_alf.ana.**read_scal**(*directory*, *obs_name*, *bare_bins=False*)

Read, skip, rebin and jackknife scalar-type bins.

Bins get skipped and rebinned according to N_skip an N_rebin retrieved through *Parameters*, then jackknife resampling is applied.

Parameters

directory
[path-like object] Directory containing the observable.

obs_name
[str] Name of the observable.

bare_bins
[bool, default=False] Do not perform skipping, rebinning, or jackknife resampling.

Returns

array
Observables. shape: *(N_bins, N_obs)*.

array
Sign. shape: *(N_bins,)*.

N_obs
[int] Number of observables.

py_alf.ana.**rebin**(*X*, *N_rebin*)

Combine each N_rebin bins into one bin.

If the number of bins (=N0) is not an integer multiple of N_rebin, the last N0 modulo N_rebin bins are discarded.

py_alf.ana.**symmetrize**(*latt*, *syms*, *dat*)

Symmetrize a dataset.

Parameters

latt
[Lattice] See *py_alf.Lattice*.

syms
[list] List of symmetry operations, including the identity of the form sym(latt, i) -> i_tranformed

dat
[array-like object] Data to symmetrize. The symmetrization is with respect to the last index of dat.

Returns

dat_sym
[numpy array] Symmetrized data.

3.6 Utility functions

Utility functions for handling ALF HDF5 files.

py_alf.utils.**bin_count**(*filename*)

Count number of bins in the given ALF HDF5 file.

Assumes all observables have the same number of bins.

Parameters

filename: str
Name of HDF5 file.

py_alf.utils.**del_bins**(*filename*, *N0*, *N*)

Delete N bins in all observables of the specified HDF5-file.

Parameters

filename: str
Name of HDF5 file.

N0: int
Number of first N0 bins to keep.

N: int
Number of bins to remove after first N0 bins.

py_alf.utils.**find_sim_dirs**(*root_in='.'*)

Find directories containing a file named 'data.h5'.

Parameters

root_in
[path-like object, default='.'] Root directory from where to start searching.

Returns

list of directory names.

py_alf.utils.**show_obs**(*filename*)

Show observables and their number of bins in the given ALF HDF5 file.

Parameters

filename: str
Name of HDF5 file.

3.7 Command line tools

A number of executable python scripts in the folder `py_alf/cli`. For productive work, it may be suitable to add this folder to the `$PATH` environment variable.

3.7.1 minimal_ALF_run

Extensively commented example script showing the minimal steps for creating and running an ALF simulation in pyALF.

3.7.2 alf_run

Helper script for compiling and running ALF.

```
usage: alf_run [-h] [--alfdir ALFDIR] [--sims_file SIMS_FILE]
               [--branch BRANCH] [--machine MACHINE] [--mpi] [--n_mpi N_MPI]
               [--mpiexec MPIEXEC] [--mpiexec_args MPIEXEC_ARGS]
               [--do_analysis]
```

3.7.2.1 Named Arguments

--alfdir	Path to ALF directory. (default: os.getenv('ALF_DIR', './ALF')
--sims_file	File defining simulations parameters. Each line starts with the Hamiltonian name and a comma, after wich follows a dict in JSON format for the parameters. A line that says stop can be used to interrupt. (default: './Sims')
--branch	Git branch to checkout.
--machine	Machine configuration (default: 'GNU')
--mpi	mpi run
--n_mpi	number of mpi processes (default: 4)
--mpiexec	Command used for starting a MPI run (default: 'mpiexec')
--mpiexec_args	Additional arguments to MPI executable.
--do_analysis, --ana	Run default analysis after each simulation.

3.7.3 alf_postprocess

Script for postprocessing Monte Carlo bins.

```
usage: alf_postprocess [-h] [--check_warmup] [--check_rebin]
                       [-l CHECK_LIST [CHECK_LIST ...]] [--do_analysis]
                       [--always] [--gather] [--no_tau]
                       [--custom_obs CUSTOM_OBS] [--symmetry SYMMETRY]
                       [directories ...]
```

3.7.3.1 Positional Arguments

directories Directories to analyze. If empty, analyzes all directories containing file "data.h5" it can find, starting from the current working directory.

3.7.3.2 Named Arguments

--check_warmup, --warmup Check warmup. Opens new window.

Default: `False`

--check_rebin, --rebin Check rebinning for controlling autocorrelation. Opens new window.

Default: `False`

-l, --check_list List of observables to check for warmup and rebinning.

--do_analysis, --ana Do analysis.

Default: `False`

--always Do not skip analysis if parameters and bins are older than results.

Default: `False`

--gather Gather all analysis results in one file named "gathered.pkl", representing a pickled pandas DataFrame.

Default: `False`

--no_tau Skip time displaced correlations.

Default: `False`

--custom_obs File that defines custom observables. This file has to define the object custom_obs, needed by py_alf.analysis. (default: os.getenv("ALF_CUSTOM_OBS", None))

--symmetry, --sym File that defines lattice symmetries. This file has to define the object symmetry, needed by py_alf.analysis. (default: None))

3.7.4 alf_bin_count

Count number of bins in ALF HDF5 file(s), assuming all observables have the same number of bins.

```
usage: alf_bin_count [-h] [filenames ...]
```

3.7.4.1 Positional Arguments

filenames Name of HDF5 files. If no arguments are supplied, all files named "data.h5" in the current working directory and below are taken.

3.7.5 alf_show_obs

Show observables and their number of bins in ALF HDF5 file(s).

```
usage: alf_show_obs [-h] [filenames ...]
```

3.7.5.1 Positional Arguments

filenames Name of HDF5 files. If no arguments are supplied, all files named "data.h5" in the current working directory and below are taken.

3.7.6 alf_del_bins

Delete N bins in all observables of the specified HDF5-file.

```
usage: alf_del_bins [-h] --N N [--N0 N0] filename
```

3.7.6.1 Positional Arguments

filename Name of HDF5 file.

3.7.6.2 Named Arguments

--N Number of bins to remove after first N0 bins.

--N0 Number of first N0 bins to keep. (default=0)

3.7.7 alf_test_branch

Script for testing two branches against one another.The test succeeds if analysis results for both branches are exactly the same.

```
usage: alf_test_branch [-h] [--sim_pars SIM_PARS] [--alfdir ALFDIR]
                       [--branch_R BRANCH_R] [--branch_T BRANCH_T]
                       [--machine MACHINE] [--devel] [--mpi] [--n_mpi N_MPI]
                       [--mpiexec MPIEXEC] [--mpiexec_args MPIEXEC_ARGS]
                       [--no_prep] [--no_sim] [--no_analyze]
```

3.7.7.1 Named Arguments

--sim_pars	JSON file containing parameters for testing. (default: './test_pars.json')
--alfdir	Path to ALF directory. (default: os.getenv('ALF_DIR', './ALF'))
--branch_R	Reference branch. (default: master)
--branch_T	Branch to test. (default: master)
--machine	Machine configuration. (default: "GNU")
--devel	Compile with additional flags for development and debugging.
--mpi	Do MPI run(s). (default: False)
--n_mpi	Number of MPI processes. (default: 4)
--mpiexec	Command used for starting an MPI run. (default: "mpiexec")
--mpiexec_args	Additional arguments to MPI executable.
--no_prep	Do not prepare runs, i.e. Compiling and creating directories.
--no_sim	Do not run ALF binary.
--no_analyze	Do not analyze and compare results.

CHAPTER

FOUR

ACKNOWLEDGMENTS

I would like to acknowledge Jefferson Stafusa Portela for proofreading this documentation.

The development of pyALF has been indirectly supported by funding for research projects from the Deutsche Forschungsgemeinschaft through the SFB1170 and FOR1807 and by direct funding from the Unitary Fund.

Furthermore, I would like to acknowledge the Gauss Centre for Supercomputing e.V. for providing computing time for research projects on the GCS Supercomputer SUPERMUC-NG at Leibniz Supercomputing Centre, which also contributed to the development of pyALF.

BIBLIOGRAPHY

[1] Martin Bercx, Florian Goth, Johannes S. Hofmann, and Fakher F. Assaad. The ALF (Algorithms for Lattice Fermions) project release 1.0. Documentation for the auxiliary field quantum Monte Carlo code. *SciPost Phys.*, 3:013, 2017. URL: https://scipost.org/10.21468/SciPostPhys.3.2.013, doi:10.21468/SciPostPhys.3.2.013.

[2] ALF Collaboration, F. F. Assaad, M. Bercx, F. Goth, A. Götz, J. S. Hofmann, E. Huffman, Z. Liu, F. Parisen Toldin, J. S. E. Portela, and J. Schwab. The ALF (Algorithms for Lattice Fermions) project release 2.0. Documentation for the auxiliary-field quantum Monte Carlo code. *arXiv:2012.11914*, 2020. URL: https://arxiv.org/abs/2012.11914.

[3] Michelle Cotton, Lars Eggert, Dr. Joseph D. Touch, Magnus Westerlund, and Stuart Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335, August 2011. URL: https://www.rfc-editor.org/info/rfc6335, doi:10.17487/RFC6335.

[4] C. J. Geyer. Markov chain monte carlo maximum likelihood. In *Computing Science and Statistics: Proceedings of the 23rd Symposium on the Interface*, 156–163. New York, 1991. American Statistical Association. URL: https://hdl.handle.net/11299/58440.

[5] Koji Hukushima and Koji Nemoto. Exchange monte carlo method and application to spin glass simulations. *Journal of the Physical Society of Japan*, 65(6):1604–1608, 1996. URL: http://dx.doi.org/10.1143/JPSJ.65.1604, arXiv:http://dx.doi.org/10.1143/JPSJ.65.1604, doi:10.1143/JPSJ.65.1604.

[6] B. Efron and C. Stein. The Jackknife Estimate of Variance. *The Annals of Statistics*, 9(3):586 – 596, 1981. URL: https://doi.org/10.1214/aos/1176345462, doi:10.1214/aos/1176345462.

INDEX

S

W