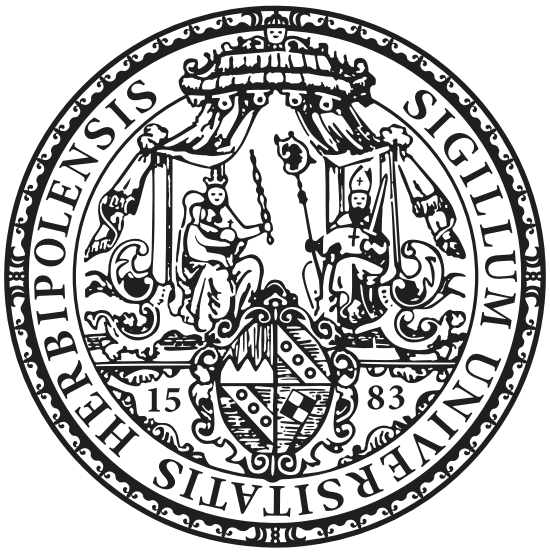
Julius-Maximilians-Universität Würzburg
Graduate School of Science and Technology

Doctoral thesis / *Dissertation*
for the doctoral degree / *zur Erlangung des Doktorgrads*
Doctor rerum naturalium (Dr. rer. nat.)

Phases and phase transitions in SU(N) spin and Dirac systems: Auxiliary field quantum Monte Carlo studies

Phasen und Phasenübergänge in SU(N)-Spin- und Dirac-Systemen: Hilfsfeld Quanten Monte Carlo Studien



Submitted by / *Vorgelegt von*
Jonas Schwab
from / *aus*
Thüngersheim
Würzburg, 2024

Git commit: 1801464

Submitted on / *Eingereicht am:* .....

Stamp / *Stempel* Graduate School

Members of thesis committee / *Mitglieder des Promotionskomitees*

Chairperson / *Vorsitz*: .....

1. Reviewer and Examiner / *1. Gutachter und Prüfer*: .....

2. Reviewer and Examiner / *2. Gutachter und Prüfer*: .....

3. Examiner / *3. Prüfer*: .....

Additional Examiners / *Weitere Prüfer*: .....

.....

Day of thesis defense / *Tag des Promotionskolloquiums:* .....

Abstract

This thesis presents three interrelated projects centered around Quantum Monte Carlo (QMC) simulations and the development of computational tools to study strongly correlated quantum systems. Two of these projects leverage the QMC package *Algorithms for Lattice Fermions* (ALF) to investigate critical phenomena in Dirac fermions and SU(N)-symmetric antiferromagnetic spin models, respectively. The third project introduces *pyALF*, a Python package that simplifies and enhances the use of ALF, making advanced simulations more accessible and efficient.

The first project explores nematic quantum phase transitions in Dirac fermions. The focus lies on two models with either C_{2v} or C_{4v} lattice point-group symmetry, respectively, which are spontaneously broken in the ordered phase, allowing for meandering Dirac points. These models are specifically designed to be free of the negative sign problem, allowing for efficient QMC simulations. My numerically obtained results, complemented by a collaborator's ϵ-expansion renormalization group study, show that both models undergo continuous phase transitions. In contrast to generic Gross-Neveu dynamical mass generation, the quantum critical regime is characterized by large velocity anisotropies, with fixed-point values being approached very slowly. Due to this slow renormalization group flow, both experimental and numerical investigations will not be representative of the infrared fixed point, but of a quasiuniversal regime where the drift of the exponents tracks the velocity anisotropy. Notably, even though the ϵ-expansion finds qualitatively distinct fixed points for the two investigated models, the numerical investigation finds no distinction in the respective exponents. Therefore, it seems that the quasiuniversial regime is at least close to the ultraviolet beginning identical for both models, even though their infrared universality is different.

The second project investigates the ground state phase diagram of an SU(N)-symmetric antiferromagnetic spin model on a square lattice. Each site hosts an irreducible representation of SU(N) described by a square Young tableau with $N/2$ rows and $2S$ columns. Negative sign-free QMC simulations are feasible for all values of S and even values of N, allowing for a comprehensive exploration of the phase diagram. In the large-N limit, the saddle point approximation favors a four-fold degenerate valence bond solid (VBS) phase, while in the large-S limit, semi-classical approximations predict Néel order. Along a line defined by $N = 8S+2$ in the S versus N phase diagram, we observe a rich variety of phases. For $S = 1/2$ and $3/2$, the system forms a four-fold degenerate VBS state, while for $S = 1$, we identify a two-fold degenerate spin nematic state that breaks the C_4 lattice symmetry down to C_2. At $S = 2$, we observe a unique symmetry-protected topological state, characterized by a dimerized SU(18) boundary state, reminiscent of the two-dimensional Affleck-Kennedy-Lieb-Tasaki (AKLT) state. These phases proximate to the Néel state align with the theoretical framework of monopole condensation of the antiferromagnetic order parameter, with degeneracies following a mod$(4, 2S)$ rule.

The third project documents the development of pyALF, a Python-based package designed to lower the barrier of entry for users new to ALF and enhance the productivity of experienced ALF users by providing a streamlined workflow for setting up and analyzing QMC simulations. Through easily reproducible examples, the documentation introduces key concepts such as preparing, executing and postprocessing simulations. The postprocessing tools in pyALF leverage Python's dynamic capabilities, enabling users to quickly define custom observables, such as correlation ratios and other complex order parameters. The package also allows for interactive checks of warmup and autocorrelation times, with a convenient way to adjust corresponding analysis parameters. Furthermore, the analysis tools allow researchers the implementation of improved estimators with minimal effort, leveraging e.g. lattice point-group symmetries to improve the accuracy of their results. By utilizing Python's extensive libraries for data analysis and visualization, pyALF enhances the workflow, with data conveniently stored in *pandas DataFrames* for easy access.

Together, these three projects offer new insights into the study of highly correlated quantum systems and provide powerful computational tools that significantly advance numerical approaches in condensed matter physics.

Zusammenfassung

Diese Dissertation präsentiert drei miteinander verbundene Projekte, die sich um Quantum-Monte-Carlo-Simulationen (QMC-Simulationen) und die Entwicklung von Rechenwerkzeugen zur Untersuchung stark korrelierter Quantensysteme drehen. Zwei dieser Projekte nutzen das QMC-Paket *Algorithms for Lattice Fermions* (ALF) zur Untersuchung kritischer Phänomene in Dirac-Fermionen bzw. SU(N)-symmetrischen antiferromagnetischen Spin-Modellen. Das dritte Projekt stellt pyALF vor, ein Python-Paket, das die Nutzung von ALF vereinfacht und verbessert, um komplexe Simulationen zugänglicher und effizienter zu machen.

Das erste Projekt untersucht nematische Quantenphasenübergänge in Dirac-Fermionen. Der Fokus liegt auf zwei Modellen mit C_{2v}- bzw. C_{4v}-Gitterpunktgruppensymmetrie, die in der geordneten Phase spontan gebrochen werden, was zu mäandernden Dirac-Punkten führt. Diese Modelle sind speziell so entworfen, dass sie frei vom negativen Vorzeichenproblem sind, was effiziente QMC-Simulationen ermöglicht. Meine numerische Analyse, ergänzt durch eine ϵ-Entwicklung-Renormierungsgruppenstudie eines Kollaborators, zeigt, dass beide Modelle kontinuierliche Phasenübergänge durchlaufen. Im Gegensatz zur generischen dynamischen Massengenerierung nach Gross-Neveu ist das quantenkritische Regime durch große Geschwindigkeitsanisotropien gekennzeichnet, wobei die Fixpunktwerte nur sehr langsam erreicht werden. Aufgrund dieses langsamen Flusses in der Renormierungsgruppe sind sowohl experimentelle als auch numerische Untersuchungen nicht repräsentativ für den infraroten Fixpunkt, sondern für ein quasi-universelles Regime, in dem das Exponenten-Driftverhalten der Geschwindigkeitsanisotropie folgt. Bemerkenswert ist, dass die ϵ-Entwicklung qualitativ unterschiedliche Fixpunkte für die beiden untersuchten Modelle findet, die numerische Untersuchung jedoch innerhalb der Fehlergrenzen keinen Unterschied in den Exponenten erkennt. Es scheint daher, dass das quasi-universelle Regime zumindest nahe dem ultravioletten Beginn für beide Modelle identisch ist, obwohl ihre infrarote Universalität unterschiedlich ist.

Das zweite Projekt untersucht das Grundzustands-Phasendiagramm eines SU(N)-symmetrischen antiferromagnetischen Spin-Modells auf einem quadratischen Gitter. Jeder Gitterplatz trägt eine irreduzible Darstellung von SU(N), die durch ein Young-Tableau mit $N/2$ Reihen und $2S$ Spalten beschrieben wird. Vorzeichenproblem-freie QMC-Simulationen sind für alle Werte von S und gerade Werte von N möglich, was eine umfassende Erforschung des Phasendiagramms erlaubt. Im Grenzfall großer N begünstigt die Sattelpunkt-Näherung eine vierfach entartete Valenzbindungsfestkörperphase (VBS), während im Grenzfall großer S semi-klassische Näherungen eine Néel-Ordnung vorhersagen. Entlang einer Linie, definiert durch $N = 8S + 2$ im S-gegen-N-Phasendiagramm, beobachten wir eine Vielzahl von Phasen. Für $S = 1/2$ und $3/2$ bildet das System einen vierfach entarteten VBS-Zustand, während wir für $S = 1$ einen zweifach entarteten Spin-nematischen Zustand identifizieren, der die C_4-Gittersymmetrie auf C_2 reduziert. Bei $S = 2$ beobachten wir einen einzigartigen symmetriegeschützten topologischen Zustand, der durch einen dimerisierten SU(18)-Randzustand gekennzeichnet ist, der an den zweidimensionalen Affleck-Kennedy-Lieb-Tasaki (AKLT)-Zustand erinnert. Diese Phasen in der Nähe des Néel-Zustands stimmen mit dem theoretischen Rahmen der Monopolkondensation des antiferromagnetischen Ordnungsparameters überein, wobei die Entartungen einer $\mathrm{mod}(4, 2S)$-Regel folgen.

Das dritte Projekt dokumentiert die Entwicklung von pyALF, einem Python-basierten Paket, das die Einstiegshürde für ALF-Nutzer:innen senkt und die Produktivität erfahrener ALF-Nutzer:innen erhöht, indem es einen optimierten Arbeitsablauf für die Einrichtung und Analyse von QMC-Simulationen bietet. Durch leicht reproduzierbare Beispiele führt die Dokumentation in zentrale Konzepte ein, wie die Vorbereitung von Simulationen, deren Ausführung und die Auswertung der Ergebnisse. Die Auswerungstools in pyALF nutzen die dynamischen Fähigkeiten von Python, wodurch Benutzer schnell benutzerdefinierte Observable wie Korrelationsbrüche (correlation ratios) und andere komplexe Ordnungsparameter definieren können. Das Paket ermöglicht auch interaktive Überprüfungen von Aufwärm- und Autokorrelationszeiten sowie eine einfache Anpassung der entsprechenden Analyseparameter. Darüber hinaus erlauben die Analysetools Forscher:innen die Implementierung verbesserter Schätzer (improved estimators) mit minimalem Aufwand, indem beispielsweise Gitterpunktgruppensymmetrien genutzt werden, um die Genauigkeit der Ergebnisse zu verbessern. Durch die Nutzung der umfangreichen Bibliotheken von Python für Datenanalyse und -visualisierung verbessert pyALF den Arbeitsablauf, wobei die Daten bequem in pandas DataFrames gespeichert werden, um einen einfachen Zugriff zu ermöglichen.

Diese drei Projekte bieten neue Einblicke in die Untersuchung stark korrelierter Quantensysteme und stellen leistungsstarke Rechenwerkzeuge bereit, die numerische Ansätze in der Festkörperphysik erheblich voranbringen.

CONTENTS

CHAPTER

ONE

INTRODUCTION

Traditionally, there are two major paradigms in science: The theoretical and the experimental approach. But one might argue that with the advent of computers, a third paradigm, namely computation, has emerged. These "bicycles for our minds", as Steve Jobs used to call computers[1] [1, 2], opened up completely new possibilities in research. With the development of computational tools, theoreticians have gained the ability to generate data themselves, an approach I utilized extensively in the research presented in this thesis.

Both research projects investigate the ground states of interacting quantum systems, but solving such systems is a hard problem. Generically, one needs to consider all possible configurations to solve the system.

Many common problems in computational science amount to summation over a high-dimensional space, which leads to a hard challenge: The volume of a space scales exponentially with the number of dimensions and therefore (when utilizing a direct approach) the computational effort as well. For bigger systems, this quickly becomes unfeasible, as the following example shows: Consider N_S spins, each having two possible states, up or down. This amounts to the simplest N_S-dimensional space, with size 2^{N_S}. The method for directly solving a corresponding model numerically is called *exact diagonalization* (ED) and amounts to calculating the eigenvalues (and -vectors) of a $2^{N_\mathrm{S}} \times 2^{N_\mathrm{S}}$ matrix[2] [3]. Doing this for $N_S = 14$ needs about 2 GB of RAM and one minute on a current laptop, but becomes quickly very expensive. To my knowledge, the biggest spin system solved via ED comprises 50 spins and took 15.5 TB of RAM to solve [4].

Rather than integrating over the whole space, one can average over a random set of its elements. This does not produce an absolutely exact result, but for a large number N of statistically independent samples in a "well-behaved" problem one can use the central limit theorem to show that the result is exact up to a statistical error proportional to $1/\sqrt{N}$ [5]. As a result, many problems that would need an exponential amount of processing power can be solved with polynomial effort instead. This method, known as statistical sampling, is older than the Monte Carlo method, but due to the tedious work of obtaining the samples it has not had significant scientific impact prior to the existence of computers.

John von Neumann and Stanislaw Ulam were possibly the first to realize the possibilities arising from combining statistical sampling with the tireless work of computers. They used the approach to simulate neutron diffusion processes in fissionable material at Los Alamos National Laboratory in the late 1940s. The secret project needed a code name and they used Monte Carlo, after the Casino in Monaco, a label that stuck [6].

The initial and ongoing exponential growth of computational power, described by Gordon E. Moore in 1965 [7] that has since become known as "Moore's law", enables new simulations leveraging the Monte Carlo method that push the boundaries of what has previously been possible.

There exist countless algorithms based on the statistical sampling approach, the method I used throughout my research is the BSS algorithm [8, 9, 10], which is a specific auxiliary field QMC algorithm. Section 1.1 will give a brief introduction to the method. More specifically, all QMC results in this thesis are produced with the program package *Algorithms for Lattice Fermions* (ALF) [11, 12], which implements the BSS algorithm in a very generic way, making it easy to implement new models. During my doctoral work, I added some significant contributions to ALF, like support for HDF5[3], a better separation/encapsulation between model definitions and the QMC algorithm, and various usability improvements [13]. The source code for ALF is publicly available at https://git.physik.uni-wuerzburg.de/ALF/ALF.

[1] Which is because human locomotion on foot is not extraordinarily energy efficient, compared to other animals, but a human on a bicycle soars to the top [23]. In that sense, the computer is a tool for the mind, as the bicycle is for locomotion. Unfortunately, most people use much less efficient modes of transport.

[2] One can usually use symmetries to split the matrix into smaller blocks, which helps immensely but does not solve the problem of exponential scaling.

[3] https://www.hdfgroup.org/solutions/hdf5/

My first doctoral research project is discussed in Chapter 2, consisting in a collaboration with Lukas Janssen, Kai Sun, Zi Yang Meng, Igor F. Herbut, Matthias Vojta, and Fakher F. Assaad whose results are published in [14]. We studied nematic quantum criticality of Dirac fermions, where the lattice rotation symmetry of the system is spontaneously broken, such that the Dirac points, which are pinned in the disordered phase, begin to meander at the critical point. Such transitions are of experimental importance, especially in the realm of d-wave superconductors [15, 16]. I provide the first numerical simulations that tackle this problem in form of QMC simulations of two distinct models which reveal that this kind of transition is continuous. One key aspect of the transition is the lack of Lorentz invariance encoded in a Fermi velocity anisotropy. Comparison with an ϵ-expansion conducted by Lukas Janssen reveals that the flow to the fixed point is extremely slow. Hence numerics as well as experiments will be characterized by a crossover regime with drifting exponents.

Chapter 3 covers the second research project, consisting in a collaboration with Francesco Parisen Toldin and Fakher F. Assaad published in [17]. Here, we treat the long-standing problem of the ground state phase diagram of the SU(N) antiferromagnet of spin S on a square lattice [18]. We simulate a fermionic representation of an SU(N)-symmetric antiferromagnetic spin model on a square lattice. Each site hosts an irreducible representation of SU(N) described by a square Young tableau of $N/2$ rows and $2S$ columns. For any S and even N, our Quantum Monte Carlo (QMC) simulations are sign problem free, which enables us to generate the first ever exact ground state phase diagram for this model with $N \in \{2, 4, \dots, 20\}$, $S \in \{1/2, 1, 3/2, 2\}$. In the large-N limit, the saddle point approximation favors a four-fold degenerate valence bond solid (VBS) phase. In the large S-limit, the semi-classical approximation points to Néel order. On a line set by $N = 8S + 2$ in the S versus N phase diagram, we observe a variety of phases proximate to the Néel state. At $S = 1/2$ and $3/2$ we observe the aforementioned four fold degenerate VBS state. At $S = 1$, a two fold degenerate spin nematic state, in which the C_4 lattice symmetry is broken down to C_2, emerges. Finally, at $S = 2$ we observe a unique ground state that pertains to a two-dimensional version of the Affleck-Kennedy-Lieb-Tasaki (AKLT) state [18, 19, 20, 21]. For our specific realization, this symmetry protected topological state is characterized by an SU(18), $S = 1/2$ boundary state, that has a dimerized ground state. These phases which are proximate to the Néel state are consistent with the notion of monopole condensation of the antiferromagnetic order parameter. In particular, one expects spin disordered states with degeneracy set by $\mathrm{mod}(4, 2S)$.

While working on these two projects, I optimized my workflows of using ALF. This resulted both in significant contributions to ALF and a set of Python scripts to streamline in particular the post-processing of data produced by ALF. These scripts eventually lead to the development of *pyALF*, a Python package built on top of ALF that is meant to simplify the different steps of working with ALF. Chapter 4 contains the documentation for pyALF. The source code for pyALF is publicly available at https://git.physik.uni-wuerzburg.de/ALF/pyALF.

Note

This work is also available as a website found at https://purl.org/diss-jschwab. This might represent a more convenient read, especially for the pyALF documentation in Chapter 4. The document is built with *Jupyter Book* [22], aiming to achieve a more interactive experience.

1.1 A brief (Auxiliary Field Quantum) Monte Carlo primer

In the following section, I provide a brief introduction of the Monte Carlo method. The aim is not to offer an exhaustive description but rather to convey a fundamental understanding of the method and its underlying concepts. The majority of the discussion focuses on classical Markov Chain Monte Carlo (MCMC) techniques, illustrated with easily reproducible examples. The section concludes with a brief exploration of how this approach can be extended to quantum models, highlighting the key challenges involved.

For more in-depth coverage, I recommend [5] for MCMC in general, [8, 9, 10] for a detailed description of our specific QMC algorithm and [12] for our actual implementation.

All examples are self-contained, meaning that they can be reproduced with the Python code included in the website version[4] of this document [24]. To run the code, in addition to Python 3, the additional libraries NumPy [25],

[4] https://purl.org/diss-jschwab

Numba [26] and Matplotlib [27] and SciPy [28] are needed. For generating the shown results, the following software versions have been used, but most other versions should also work.

```
Software versions:
Python 3.12.7, NumPy 2.0.2, Numba 0.60.0, Matplotlib 3.9.2, SciPy 1.14.1
```

1.1.1 Stochastic integration

As previously mentioned, the Monte Carlo method refers to a computer-based stochastic sampling technique. It is commonly employed to compute the expectation value of a function $f(x)$

$$\langle f \rangle = \int dx\ f(x)\rho(x).$$

Usually, x spans a high-dimensional space[5] and $f(x)$ corresponds to what we later refer to as an observable. In general, the probability distribution $\rho(x)$ is only accessible through a weight $w(x) \propto \rho(x)$. In order to obtain the probability distribution, one needs to normalize the weight:

$$\rho(x) = \frac{w(x)}{\int dx'\ w(x')}.$$

Which results in

$$\langle f \rangle = \frac{\int dx\ f(x)w(x)}{\int dx\ w(x)}.$$

In a naive approach for approximating this integral, one draws N values for x from a uniform distribution and just replaces the integral over all possible values of x with a sum over this sample:

$$\text{Naive sampling:} \qquad \langle f \rangle \approx \frac{\sum_{x \in \text{Sample}} f(x)w(x)}{\sum_{x \in \text{Sample}} w(x)}. \tag{1.1}$$

In principle this approach works, but in higher dimensions it becomes extremely inefficient and therefore unfeasible: Most of the values within the sample will be from regions with very small weight. A more practicable approach is the so-called importance sampling, where the random values are drawn from a non-uniform distribution according to their weight $w(x)$. As a result, given the existence of a suitable sample, our equation for estimating the expectation value simplifies a bit:

$$\text{Importance sampling:} \qquad \langle f \rangle \approx \frac{1}{N} \sum_{x \in \text{Sample} \propto w} f(x). \tag{1.2}$$

A key component for the success of stochastic integration is the central limit theorem (CLT). It states that the mean of N independent values drawn from the same distribution with expectation value μ and variance σ^2 converge in the limit $N \to \infty$ to a normal (i.e. Gaussian) distribution with expectation value μ and variance σ^2/N. This guarantees that statistical sampling will give us the correct answer with a statistical error scaling asymptotically as $1/\sqrt{N}$. There is a caveat that some probability distributions do not have a well-defined expectation value and therefore do not follow the CLT, for example fat-tailed distributions, which pose a big challenge for a significant subset of QMC simulations [29, 30]. I will show a minimal example with a fat-tailed distribution in Section 1.1.1.2.

[5] Reminder: The advantage of stochastic sampling is its superior dimensional scaling (polynomial) compared to a brute-force approach leading to an exponential scaling in computational effort, as will be demonstrated in Fig. 1.3. Low-dimensional integrals can usually just be solved with a "brute force" approach.

1.1.1.1 Example: Stochastically calculating a 1d integral

In the following, I will demonstrate stochastic integration on a simple one-dimensional example, by evaluating:

$$\mu = \langle x^2 \rangle = \frac{\int_{-5}^{5} dx\ x^2 \exp\left(-\frac{x^2}{2}\right)}{\int_{-5}^{5} dx\ \exp\left(-\frac{x^2}{2}\right)}. \tag{1.3}$$

Which means we're evaluating the function $f(x) = x^2$ with the weight $w(x) = \exp\left(-\frac{x^2}{2}\right)$ in the range $x \in [-5, 5]$. Note that the restriction on the range of x values is somewhat artificial and an integration over all real numbers would probably be closer to a realistic scenario, but I chose the restriction to be able to compare importance sampling with the naive sampling approach and the latter would not have been applicable for integrating over all real numbers. This restriction is also the reason why $w(x)$ is not equal to $\rho(x)$ and a normalization is necessary.

The plots in Fig. 1.1 are produced with a few lines of Python, embedded below this paragraph in the website version[6] of this document. By comparing Fig. 1.1(b) and Fig. 1.1(c), one can see how the importance sampling focuses mainly on the peak of the normal distribution, while the naive approach samples indiscriminately over the whole parameter space. After approximately the first ten samples, the two approaches perform similarly well (c.f. Fig. 1.1(d,e)), which is because the parameter space is very small, contrary to common Monte Carlo applications. If we increase the integration interval from $x \in [-5, 5]$ to $x \in \mathbb{R}$, the naive sampling (Eq. (1.1)) would already fail, while importance sampling (Eq. (1.2)) would perform nearly the same.

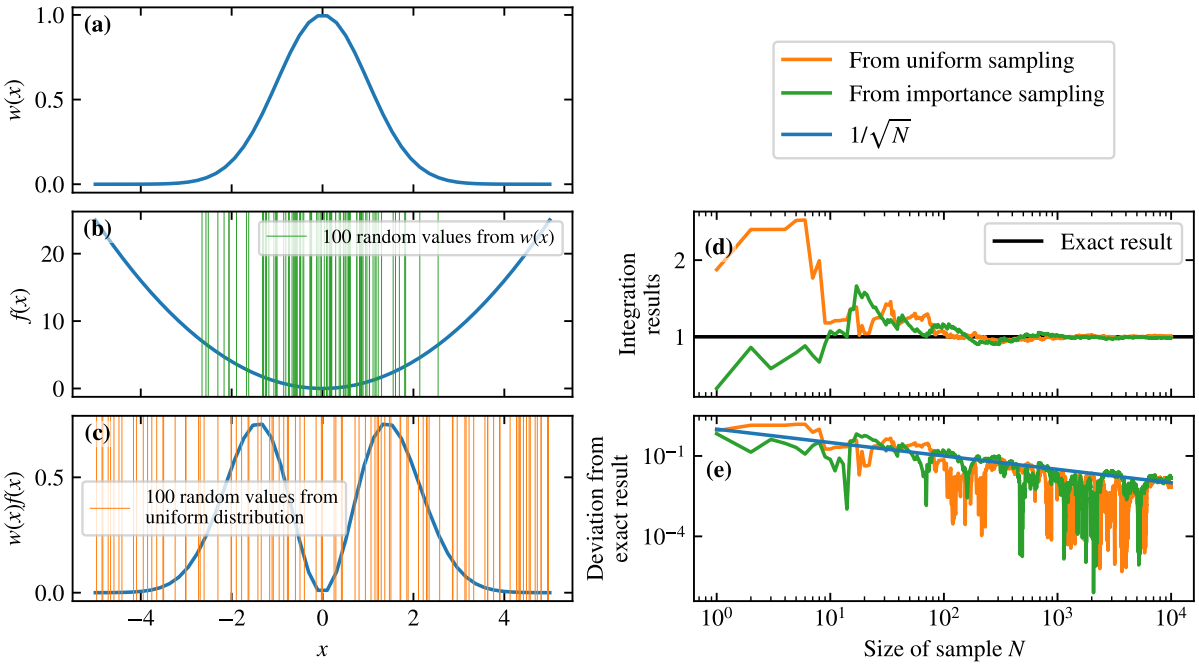


Fig. 1.1: Demonstration of naive sampling (Eq. (1.1)) and importance sampling (Eq. (1.2)) through numerical integration of Eq. (1.3). a: Probability density $w(x) = \exp(-x^2/2)$. b: Function $f(x) = x^2$ whose expectation value is to be calculated, with $x \in [-5, 5]$. The vertical lines demonstrate the importance sampling approach. c: Intergrand $w(x)f(x)$. The vertical lines demonstrate the naive sampling approach. d: Cumulative integration results over samples of size N. e: Deviation of the integration results from the exact result. The deviation scales as $1/\sqrt{N}$, as predicted by the CLT.

To look at the effect of increasing the parameter space, we generalize Eq. (1.3) to more dimensions, resulting in:

$$\mu = \frac{\int_{-5}^{5} dx_1 \ldots \int_{-5}^{5} dx_{N_{\text{dim}}}\ \boldsymbol{x}^2 \exp\left(-\frac{\boldsymbol{x}^2}{2}\right)}{\int_{-5}^{5} dx_1 \ldots \int_{-5}^{5} dx_{N_{\text{dim}}}\ \exp\left(-\frac{\boldsymbol{x}^2}{2}\right)}. \tag{1.4}$$

Simulating for $N_{\text{dim}} \in \{1, 2, 3, 4\}$ results in Fig. 1.2, which shows the decreasing performance of uniform sampling compared to importance sampling. To better compare the performance between these approaches, it is helpful to

[6] https://purl.org/diss-jschwab

compare the computing time necessary to achieve a certain precision. In order to obtain these times, we fit the deviation from the exact result to $a/\sqrt{N}$, where N is the number of random numbers and a a fitting parameter.[7] From this, we can estimate the computing time t_σ needed to achieve a precision of e.g. $\sigma = 10^{-2}$ as

$$t_\sigma = t_1 \left(\frac{a}{\sigma}\right)^2, \tag{1.5}$$

where t_1 is the time needed for a single sample.

In Fig. 1.3 these estimates and the actual computing time to generate the results of Fig. 1.2 are plotted. For both uniform sampling and exactly calculating[8] the integral Eq. (1.4) the time t scales exponentially with N_{dim}. Therefore, importance sampling is needed to beat the exponential scaling of a direct integral and achieve polynomial scaling.

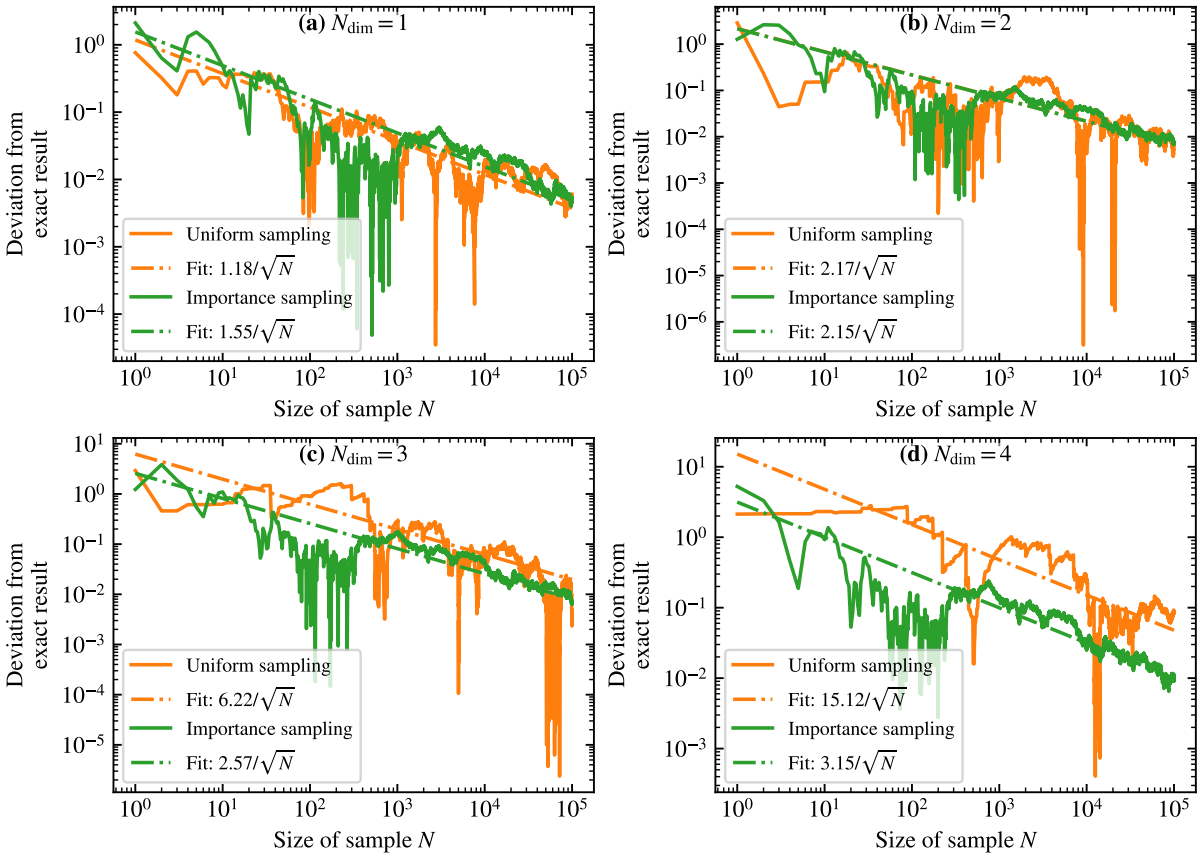


Fig. 1.2: Comparison of uniform and importance sampling through numerical evaluation of Eq. (1.4) for diffrent number of dimensions N_{dim}. With increasing parameter space, one observes decreasing performance of the uniform sampling compared to importance sampling.

[7] Instead of fitting to $a/\sqrt{N}$ one could also estimate the standard error through the spread of values, which is usually the approach since the exact value is not known.

[8] The reference values have been created through exact numeric integration using `scipy.integrate.quad()`[11].

[11] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad

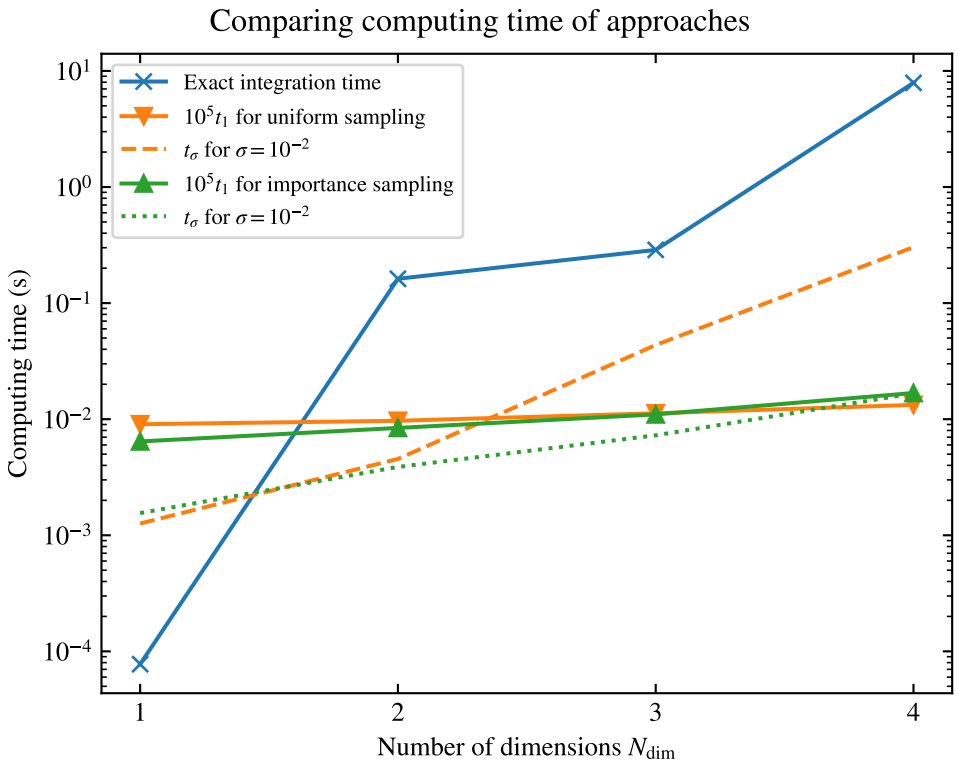


Fig. 1.3: Computing time needed for evaluating Eq. (1.4) with different approaches (exact integration, importance sampling, uniform sampling) to produce Fig. 1.2. The dashed (dotted) line estimates the time needed to achieve a precision of 10^{-2} with uniform (importance) sampling (cf. Eq. (1.5)). Both non-stochastic (= exact) numerical integration and uniform sampling scale exponentially in computational effort when desiring a set precision, therefore only with importance sampling a polynomial scaling is achievable.

1.1.1.2 Example: Fat tails

Here, I will give a very simple demonstration of the fat tail problem by trying to stochastically estimate the mean of a Lorentz distribution.

The Lorentz distribution

$$P(x) = \frac{1}{\pi(1+x^2)}$$

is symmetric around $x = 0$, therefore its median is zero. So one might expect the mean to be also zero. But actually the integral

$$\langle x \rangle = \int_{-\infty}^{\infty} dx\, xP(x)$$

does not converge, because $P(x)$ falls off so slowly. It has a long or fat tail, see the comparison to a Gauss distribution in Fig. 1.4(a). Therefore, the Lorentz distribution does not have a mean and the CLT can not be applied.

In Fig. 1.4(b,c), we can see the attempt to calculate the mean of this distribution through stochastic sampling. An endeavor that seems to be successful if we focus on some parts of the sequence, but occasionally the long tails will create a big outlier, also called a spike, that throws off the result. If you find such spikes in your Monte Carlo time sequence, it means you have fat tails and the CLT is not applicable any more[9].

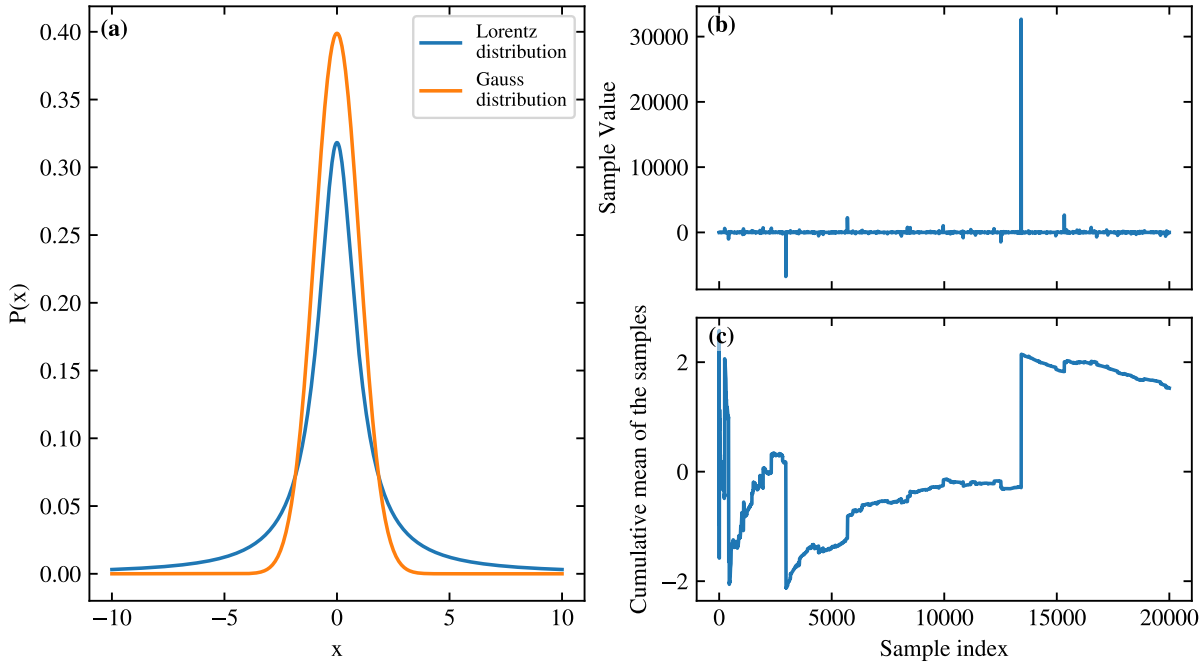


Fig. 1.4: Demonstration of fat-tailed distributions: Trying to estimate the mean of a Lorentz distribution through stochastic sampling fails. a: Comparison of Lorentz distribution to Gauss distribution: The Lorentz distribution decays extremely slowly (i.e. with a power law). b: 20×10^3 random numbers drawn from a Lorentz distribution. One can see a few of the characteristic outliers, also called "spikes", indicating a fat-tailed distribution. c: Cumulative mean of the random numbers. Periodically, it seems to converge to the median $x_0 = 0$, until a spike throws the value off.

[9] Reminder: The CLT is a fundamental cornerstone of Monte Carlo methods.

1.1.2 Markov chain Monte Carlo

The previous examples used normal distributions as probability weights, which can easily be sampled directly. But our actual problems are too complicated to directly generate samples according to their probability weights. Instead, we only have access to the weight of one specific given configuration. The established stochastic sampling approach for solving this problem is called Markov chain Monte Carlo (MCMC). A Markov chain describes a sequence of configurations $(C_1, C_2, C_3, \dots)$, where (starting from an initial configuration C_1) a new configuration C_{i+1} is derived from its previous configuration C_i plus a random component. C_{i+1} does not explicitly depend on configurations earlier than C_i. The propagation to the next configuration of the chain —from now on referred to as an *update*— is designed such that the configurations on the chain correspond to the desired importance samples. Notably, propagation along the chain does (usually) not correspond to any physical time (except for CPU time during the simulation).

Analogous to Eq. (1.2), an expectation value $\langle f \rangle \equiv \sum_C f(C)P(C)$ can then be approximated as

$$\langle f \rangle \approx \frac{1}{N} \sum_{i=1}^{N} f(C_i),$$

with some caveats that I will address in a few paragraphs.

For a successful MCMC simulation, the updating rules have to fulfill two criteria: *Stationarity* with regards to the probability distribution and *ergodicity*.

Stationarity means that, statistically, an update maps the desired probability distribution $P(C)$ onto the same distribution, which means that all "probability flows" from and to any given configuration balance out:

$$P(C) = \sum_{C'} P(C|C')P(C') \quad \forall C,$$

where $P(C'|C)$ is the probability that the next configuration is C' if the current configuration is C. The extension to non-normalized weights W is straightforward:

$$W(C) = \sum_{C'} P(C|C')W(C') \quad \forall C.$$

Many MCMC algorithms implement a stronger criterion called detailed balance, where the "probability flow" between two configurations is symmetric:

$$P(C'|C)P(C) = P(C|C')P(C') \quad \forall C, C',$$

and with non-normalized weights

$$P(C'|C)W(C) = P(C|C')W(C') \quad \forall C, C'.$$

Fig. 1.5 demonstrates both detailed balance and a weaker version of stationarity through an example with three states A, B and C with statistical weights of $w(A) = 1$, $w(B) = 2$ and $w(C) = 3$, corresponding to probabilities of $P(A) = 1/6$, $P(B) = 1/3$ and $P(C) = 1/2$, respectively. The probabilities are achieved by normalizing the weights: $P(X) = W(X)/\sum_i W(i)$.

Ergodicity means that the updates are actually able to reach the stationary distribution. This is equivalent to saying that all relevant parts of the configuration space are covered by the simulation, or that any configuration C is reachable from any other configuration C' within a finite number of updates. Note that simulations who are in principle ergodic can be non-ergodic in practice, if the simulation moves through the whole configuration space, but too slowly for any practical computing times. Ergodicity issues are one of the major challenges in many Monte Carlo simulations, for example in low-temperature simulations of spin glasses, where algorithms tend to get stuck in local minima of the energy (= local maxima of the Monte Carlo weight).

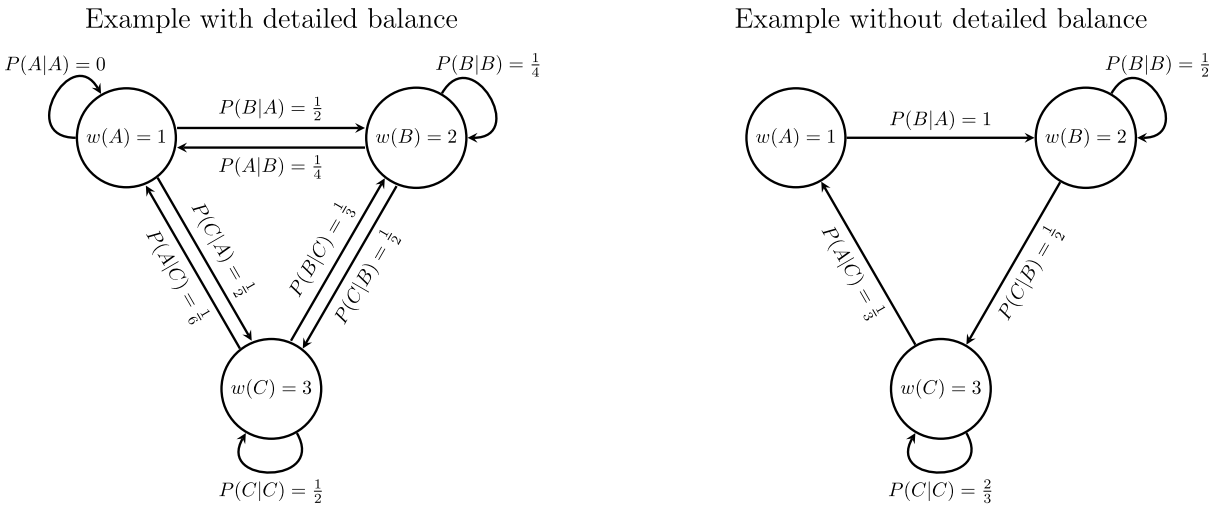


Fig. 1.5: Simple example of Markov chain processes with and without detailed balance. Both Markov processes have possible states A, B and C with statistical weights of 1, 2 and 3, corresponding to probabilities of $1/6$, $1/3$ and $1/2$, respectively.

1.1.2.1 Caveats: Autocorrelation and Warmup

Contrary to the direct sampling approach as shown in Section 1.1.1.1, the configurations created by MCMC are not independent, but C_i is usually strongly correlated to C_{i+1}, a bit weaker to C_{i+2} and so on. Therefore, the CLT does not hold and a naive estimation of the errors will grossly underestimate them. This correlation between different configurations on the Markov chain can be quantified by the autocorrelation γ_τ of an observable O

$$\gamma_\tau(O) = \frac{\sum_i \langle (O_i - \mu)(O_{i+\tau} - \mu) \rangle}{\sum_i 1}$$

and the normalized autocorrelation

$$\bar{\gamma}_\tau(O) = \frac{\gamma_\tau(O)}{\gamma_0(O)}. \tag{1.6}$$

Usually, the autocorrelation declines asymptotically exponentially with τ:

$$\bar{\gamma}_\tau(O) \sim \exp\left(-\frac{\tau}{\tau_\gamma(O)}\right), \tag{1.7}$$

where we call $\tau_\gamma(O)$ the autocorrelation time of observable O.

One established method of dealing with autocorrelation is to combine multiple measurements from adjacent Markov chain configurations in what we refer to as a bin.

$$\bar{O}_i = \frac{1}{M_{\text{bin}}} \sum_{n=(i-1)M_{\text{bin}}}^{iM_{\text{bin}}} O_n$$

Here, we combine M_{bin} measurements into one bin. If M_{bin} is large enough (order of $\tau_\gamma(O)$), the bins are virtually uncorrelated and we can use the central limit theorem again.

Similarly to autocorrelation, there is often also what we refer to as "*warmup*" period, sometimes also called "burn-in" or "equilibration". It refers to a period at the beginning of a simulation whose measurements should be dismissed to achieve the correct results. This stems from the fact that finding good initial configurations for a Markov chain is hard and most initial configurations are very untypical, meaning they are far from the region of the configuration space making up the bulk of the configurations traversed during the simulation.

I will show in an example in Section 1.1.2.3 how to control for autocorrelation and warmup.

Note

Both autocorrelation and warmup times are observable dependent, meaning that different observables like magnetisation and energy of the same simulation have generally different autocorrelation and warmup times.

1.1.2.2 Metropolis-Hastings algorithm

Finally, we need an algorithm that implements MCMC. A very common one is the *Metropolis algorithm* [31], developed in 1953 by Metropolis et al. and generalized to the *Metropolis-Hastings algorithm* [32] by Hastings in 1970.

The algorithm stands out for its simplicity:

After the creation of an initial state C_1, a step that all MCMC simulations have in common, the algorithm comprises of a loop, where each iteration consist of these four steps:

Algorithm 1.1.1 (Metropolis-Hastings)

1. **Propose update:** The system is currently in state $C = C_i$ and a potential new state C' is proposed. How C' is generated exactly is beyond the scope of this algorithm and can often be the most challenging aspect of designing a Monte Carlo simulation, since an update should ideally move quickly and efficiently through the phase space to reduce autocorrelation times and ensure ergodicity. This includes that updates should have acceptance probabilities large enough to be effective.
2. **Calculate the acceptance probability:**
$$A(C'|C) = \min\left(1, \frac{W(C')g(C|C')}{W(C)g(C'|C)}\right), \tag{1.8}$$
where $W(C)$ is the probability weight of the configuration C and $g(C|C')$ $[g(C'|C)]$ is the probability that an update to C $[C']$ is proposed, if in step 1 the system is in state C' $[C]$.[10]
3. **Accept / reject update:** Generate random number $p \in [0, 1[$. If $A(C'|C) > p$, then accept the update: $C_{i+1} = C'$. Else, reject the update: $C_{i+1} = C$.
4. **Go back to 1.**

1.1.2.3 Example: One-dimensional Ising chain

As a simple demonstration of the Metropolis algorithm, I am going to simulate the one-dimensional Ising chain with periodic boundary conditions. It consists of L classical spins with a ferromagnetic nearest-neighbor interaction J:

$$H(C) = -J\sum_{i=1}^{L} s_i s_{i+1} \qquad s_{L+1} = s_1 \qquad C = (s_1, s_2, ..., s_L) \qquad s_i \in \{-1, 1\}. \tag{1.9}$$

Where C is one of the 2^L possible configurations. This model can be quickly solved analytically. We calculate the thermodynamic partition function

$$Z = \sum_C e^{-\beta H(C)} = \prod_{i=1}^{N}\left(\sum_{s_i s_{i+1}\in\{-1,1\}} \exp(\beta J s_i s_{i+1})\right) = \left(2\cosh(\beta J)\right)^L$$

and the energy

$$E = \langle H\rangle = \frac{1}{Z}\sum_C H(C)e^{-\beta H(C)} = -\frac{\partial}{\partial\beta}\ln(Z) = -JL\tanh(\beta J),$$

which we will use as a reference to check the accuracy of the simulation.

[10] In the pre-Hastings Metropolis algorithm, $g(C|C') = g(C'|C)$.

Instead of calculating the sum over C analytically, one can approximate it through importance sampling,

$$E = \frac{\sum_C H(C) e^{-\beta H(C)}}{\sum_C e^{-\beta H(C)}}$$

$$E \approx \frac{1}{N} \sum_{C \propto W(C)} H(C) \qquad W(C) = e^{-\beta H(C)}, \tag{1.10}$$

where $C \propto W(C)$ means that C is sampled according to the weight $W(C)$ and N is the number of samples.

To evaluate Eq. (1.10) with the Metropolis-Hastings algorithm, it is now necessary to generate an initial configuration C_1 and define an algorithm for proposing updates. C_1 will be generated randomly by setting each spin individually to either -1 or 1 with equal probability. The proposal of an update is done by choosing a random spin and flipping it.

In the website version[11] of this document, you can view the source code defining a function `run_1d_ising_wrap`, which simulates the Ising chain. We measure the energy E, magnetization $m = \left\langle \sum_i s_i \right\rangle$ and its second and fourth moment $m_2 = \left\langle (\sum_i s_i)^2 \right\rangle$ and $m_4 = \left\langle (\sum_i s_i)^4 \right\rangle$ after every update. Measurements from L updates are averaged into one bin.

Here, we simulate with $L = 200$ spins, $J = 1.0$ and $\beta = 2.0$. We simulate for 50×10^3 bins, resulting in a total of $LN_{\text{bins}} = 10 \times 10^6$ updates.

```
L = 200           # Number of Ising spins
J = 1.0           # Ferromagnetic interaction strength
beta = 2.0        # Reciprocal temperature
N_bins = 50000    # Number of Monte Carlo bins

energy_exact = energy_1d_ising(J, beta, L)
observables, final_state = run_1d_ising_wrap(L, J, beta, N_bins)
```

We first look at the time series of the measured bins in Fig. 1.6. The definition of the used function `plot_bins` is again viewable in the website version. This time series already gives a good first overview of the simulation. We can see what might be a good number of bins to skip for the warmup periods and also get an idea about autocorrelation times. The energy shows a drift for about the first 300 bins, after which it seems to oscillate around the mean. This means dismissing the first 300 bins is the correct approach for accurately estimating the energy. On the other hand, the magnetization m seems to be "warmed up" right from the beginning, but it oscillates on a time scale of several thousand bins, suggesting a similarly big autocorrelation time.

To more accurately estimate autocorrelation times, we can now calculate the autocorrelation by dismissing the non-equilibrated bins and applying Eq. (1.6) on the rest. The result is then fitted to an exponential decay, using Eq. (1.7) to estimate the autocorrelation time τ_{auto}.

Furthermore or alternatively, one can calculate the standard error σ, assuming N independent measurements x_i for different rebinning values N_{rebin}:

$$\sigma^2 = \frac{\sum_{i=1}^{N} (\bar{O}_i - \mu)^2}{N(N-1)}; \qquad \mu = \frac{\sum_{i=1}^{N} \bar{O}_i}{N}; \qquad \bar{O}_i = \frac{1}{N_{\text{rebin}}} \sum_{n=(i-1)N_{\text{rebin}}}^{iN_{\text{rebin}}} O_n.$$

If N_{rebin} is too small, the error will be underestimated and increasing N_{rebin} will increase the error estimate, until it saturates signaling the correct value for N_{rebin}.

Both approaches are shown in Fig. 1.7. One can see greatly differing autocorrelation times τ_{auto} between the four observables. While the energy has the smallest τ_{auto} of less than 100 bins, the magnetization m has that more than ten times. The second and fourth moment of the magnetization lay in between those two extremes with approximately 350 and 270, respectively.

Notably, the strong autocorrelation of m is in a sense an artifact of how the updates are designed. Since inversion of all Ising spins ($s_i \to -s_i \forall i$) is a symmetry of the model, follows $m = 0$. On the Monte Carlo side, this is expressed by the fact that the update $s_i \to -s_i \forall i$ would always be accepted. Therefore, we could simply set $m = 0$, which would be called an *improved estimator*.

[11] https://purl.org/diss-jschwab

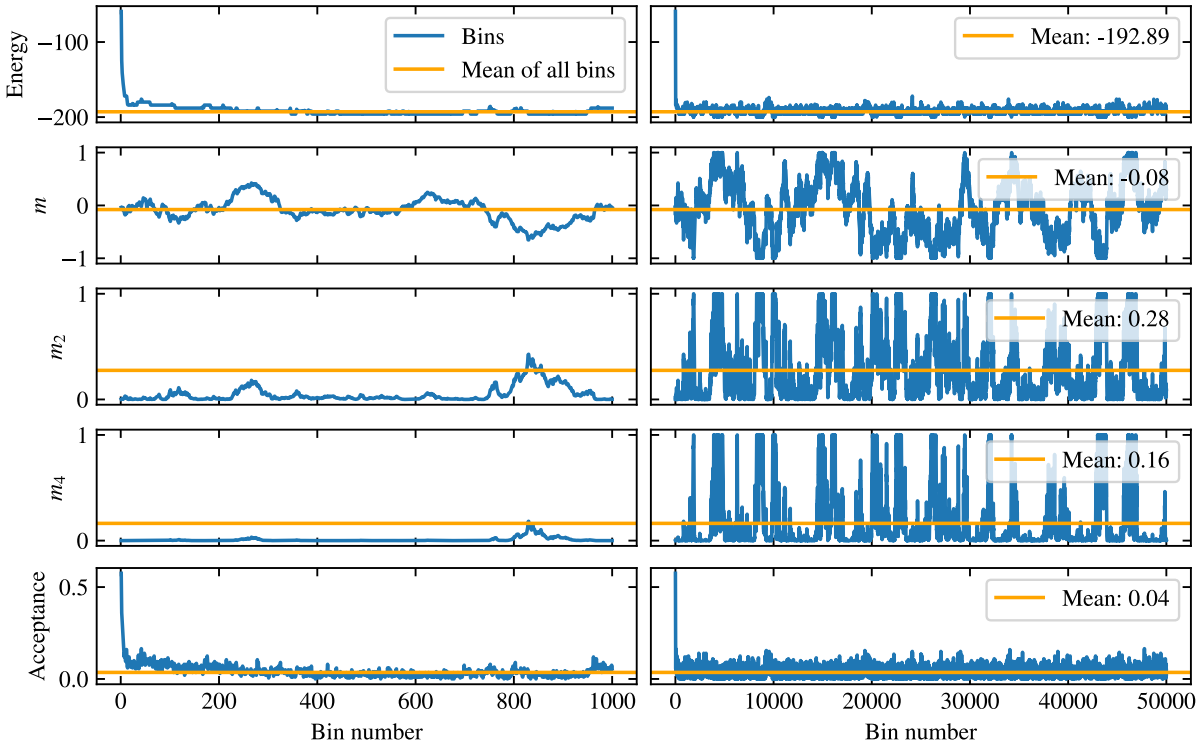


Fig. 1.6: Time series of bins measured while simulating the one-dimensional Ising chain. Left: Zoomed in to the first 1000 bins. Right: all 50×10^3 bins.

Now that we are aware of the warmup and autocorrelation times, we can determine the final results of the simulation. Out of convenience, we use for all observables the same parameters, skipping the first 1000 bins and rebinning 2000 original bins into one. The results are shown below. The exact values $E \approx -192.805516015$ and $m = 0$ are reproduced within margin of errors.

$$\begin{aligned} E &= -192.9 \pm 0.2 \\ m &= -0.08 \pm 0.08 \\ m_2 &= 0.28 \pm 0.03 \\ m_4 &= 0.17 \pm 0.02 \end{aligned}$$

1.1.2.4 Example in two dimensions: Critical slowing down

After finishing our first proper Monte Carlo simulation, I want to show a phenomenon called critical slowing down. For this, we go from one to two dimensions: Simulating the two-dimensional Ising model on a square lattice allows us to investigate a continuous phase transition. The model was analytically solved in 1944 by Onsager [33], therefore we know it has a paramagnetic-ferromagnetic phase transition at reciprocal temperature $\beta_c = \frac{\ln(1+\sqrt{2})}{2J} \approx 0.44069/J$. Meaning that a high temperature above the critical point $\beta < \beta_c$, the system is disordered and forms a paramagnet. However, when lowering the temperature $\beta > \beta_c$ the system orders and becomes ferromagnetic.

Simulating at $J = 1$ for system sizes 12×12 and 24×24 in the range $\beta \in [0.35, 0.5]$, we find the autocorrelation times and acceptance ratios displayed in Fig. 1.8. Once again, the code for producing this results is found below this paragraph on the website version[12].

We use the same single spin flip updates as in the previous example. We find that the autocorrelation times increase dramatically around the critical point β_c. This effect is related to the nature of critical points: At a critical point, the system exhibits fluctuations at all energy and length scales, which is very hard to capture efficiently through Monte Carlo updates. In this case, the single spin flip updates struggle with long-ranged modes, increasing the

[12] https://purl.org/diss-jschwab

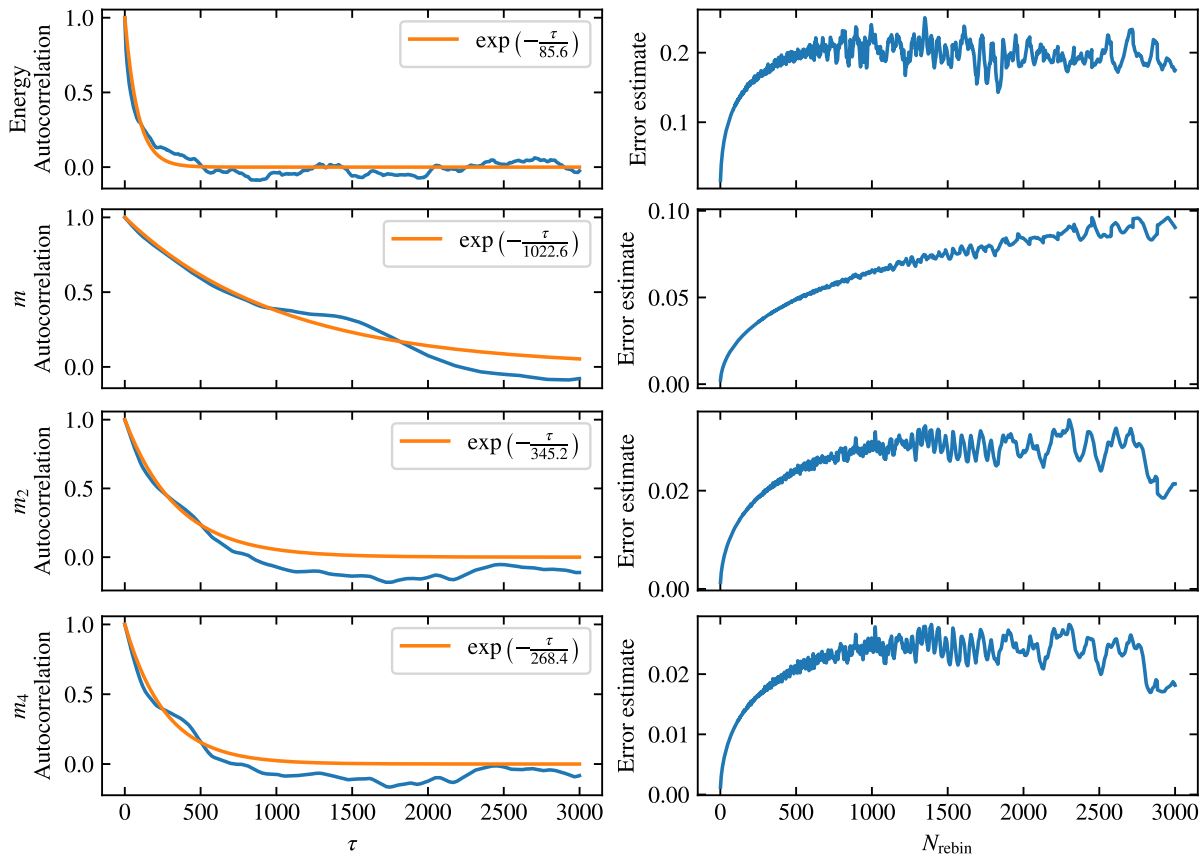


Fig. 1.7: Autocorrelation and error estimates for a simulation of the one-dimensional Ising chain. Left: Normalized autocorrelation vs distance between bins, with fits to an exponential decay for estimating the autocorrelation times τ_{auto}. Right: Error estimate, assuming independent bins, vs rebinning value N_{rebin}. It saturates approximately at $N_{\text{rebin}} = \tau_{\text{auto}}$.

autocorrelation times. Furthermore, the acceptance ratio (i.e. the probability that a proposed update gets accepted) diminishes continuously with increasing β, additionally impacting autocorrelation times.

Despite the long autocorrelation times, the simulations produce accurate results. We can use the Binder ratio B [34]

$$B = \left(1 - \frac{m_4}{(m_2)^2}\right)/2$$

to detect the critical point. B is a renormalization group (RG) invariant quantity, which means it is either 0 or 1 in the thermodynamic limit, depending on whether the system is in an unordered or ordered phase. B, plotted as a function of the control parameter (in this case β), is expected to intersect at the critical point for different system sizes. Exactly this behavior is shown in Fig. 1.9, reproducing Onsager's value for β_c within the margin of error. Still, due to critical slowing down, the precision of the numerical results is not very high, a limitation, we are going to address in the next step by using Wolff cluster updates.

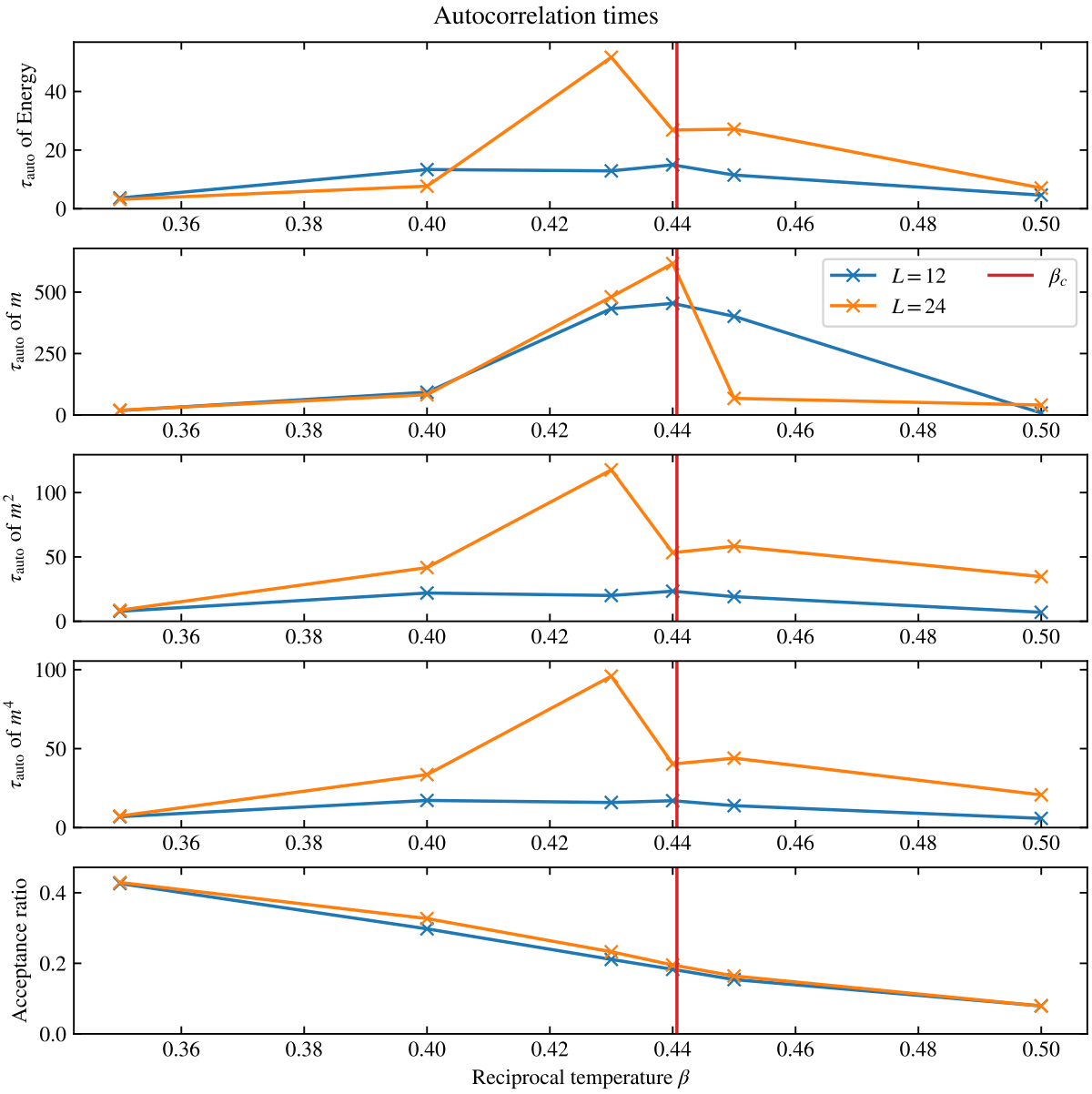


Fig. 1.8: Autocorrelation times and acceptance ratio for 2d Ising model for two lattice sizes, simulated using single spin flip updates. The autocorrelation times increase dramatically around the critical point β_c, a phenomenon that is called critical slowing down. The acceptance ratio is high in the disordered phase at high temperature and decreases when approaching the ordered phase by decreasing the temperature.

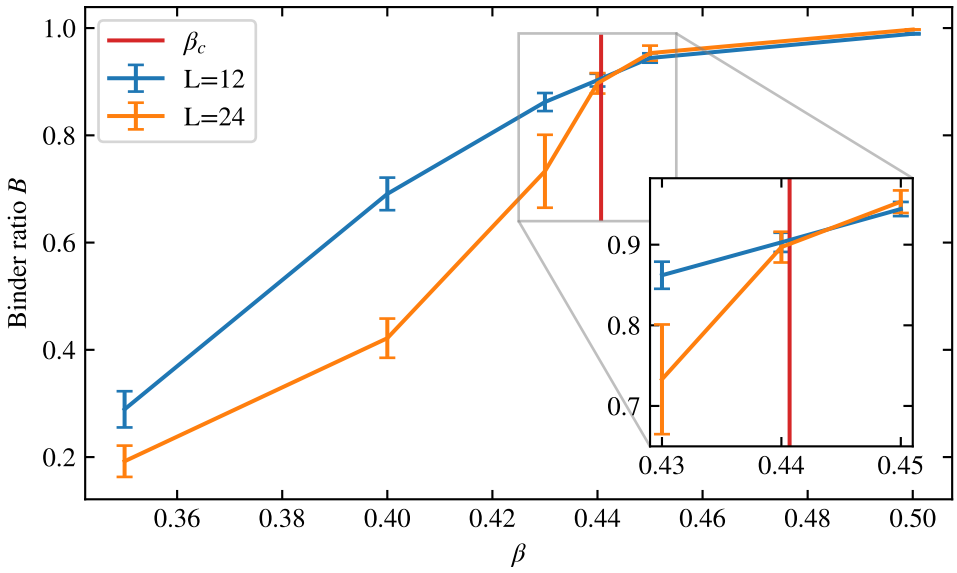


Fig. 1.9: Binder ratio for 2d Ising model at $J = 1$. The crossing around $\beta \approx 0.44$ signals a phase transition, a result which is in accordance with the exact result by Onsager from 1944 [33].

1.1.2.4.1 Solving critical slowing down with the Wolff algorithm

A strategy for addressing critical slowing down, or ergodicity challenges in general, is to go from atomistic (e.g. single spin flip) updates to bigger correlated updates, which for example flip an entire cluster of spins. Updates like these are generally very hard to design: Most states in the phase space have high energy, therefore a random configuration change $C \to C'$ is very likely to increase the energy, moving the system away from the higher-weighted low-energy states. The positive energy mismatch $E(C') - E(C)$ reduces the weight —and therefore the acceptance probability of that state— exponentially (cf. Eq. (1.8)). Or in other words: Proposing an update that moves the configuration fast through the configuration space, will most likely result in a very low acceptance probability, except if significant knowledge about the model was used to design the update.

An example of a well-designed algorithm generating such updates is the Wolff algorithm [35]. It is specifically designed for simulating the Ising model, by building clusters of spins that get flipped collectively. Notably, each cluster is built in such a way that $\frac{W(C')g(C|C')}{W(C)g(C'|C)} = 1$ (cf. Eq. (1.8)), hence flipping a cluster is accepted with probability 1.

Algorithm 1.1.2 (Wolff cluster)

1. **Choose initial spin:** Randomly chose one spin as the seed for the cluster.
2. **Try to add neighboring spins:** For all neighboring spins that are not already part of the cluster and have the same orientation as the spins in the cluster: Try to add this spin to the cluster with probability $1 - \exp(-2\beta J)$.
3. If a spin gets added, perform step 2 for that spin.
4. Flip cluster.

Simulating the two-dimensional Ising model with the help of *Algorithm 1.1.2* leads to autocorrelation times and average cluster sizes as displayed in Fig. 1.10. The algorithm solves critical slowing down by scaling the average cluster size with the correlation length in the system. With this approach, we manage to create more precise data in less time.[13] The result is displayed in Fig. 1.11.

[13] In this concrete example, the simulations for Fig. 1.9 took my computer about 64 seconds to complete, while the simulations for Fig. 1.11 took about 21 seconds, even though the former only computed system sizes 12×12 and 24×24, while the latter went up to sizes of 36×36.

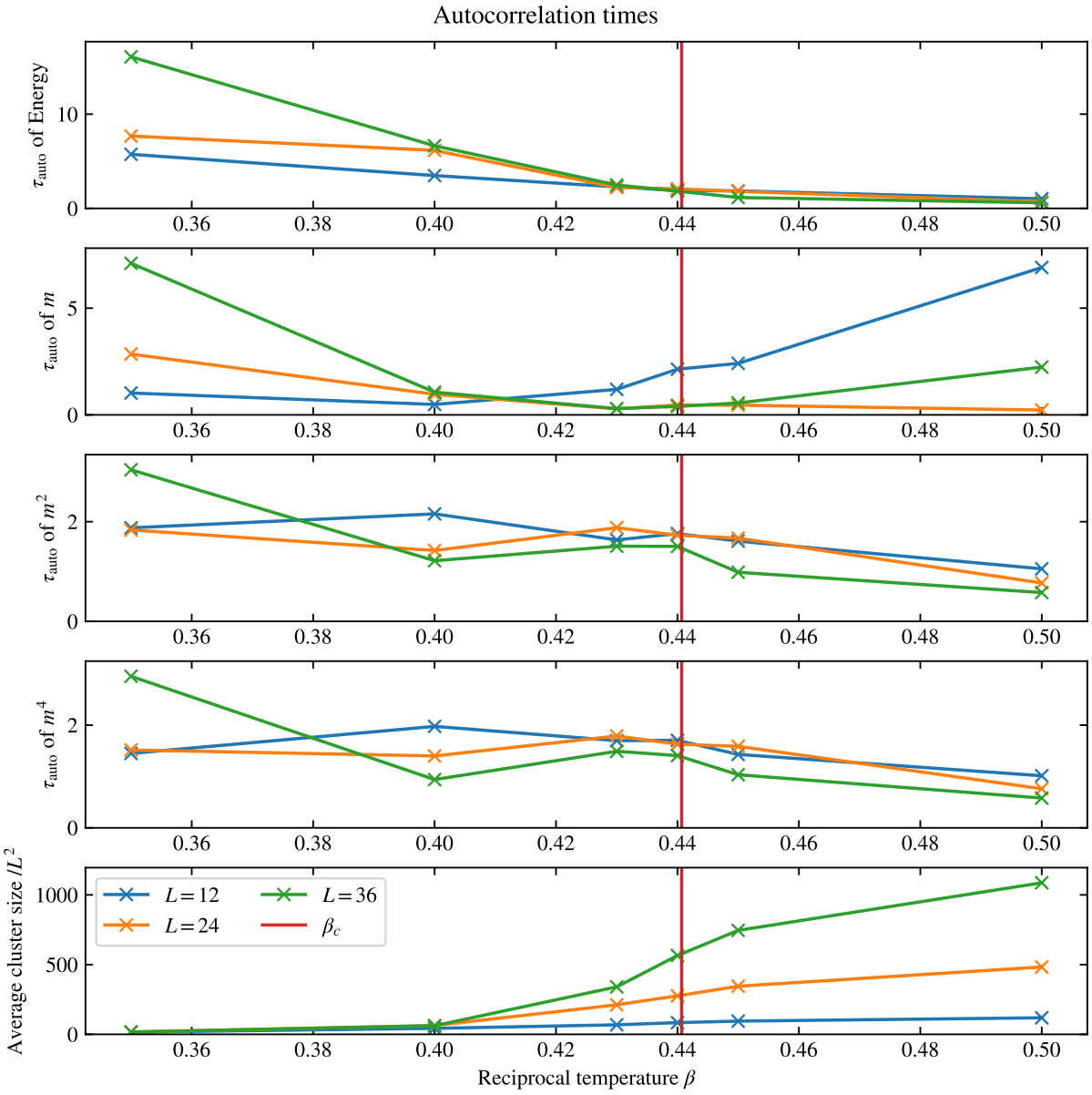


Fig. 1.10: Autocorrelation times and average cluster size for 2d Ising model for three lattice sizes, simulated with Wolff cluster updates. The Wolff algorithm (*Algorithm 1.1.2*) solves critical slowing down. The cluster size grows dynamically when going from the disordered to the ordered phase, mirroring the correlation length in the system.

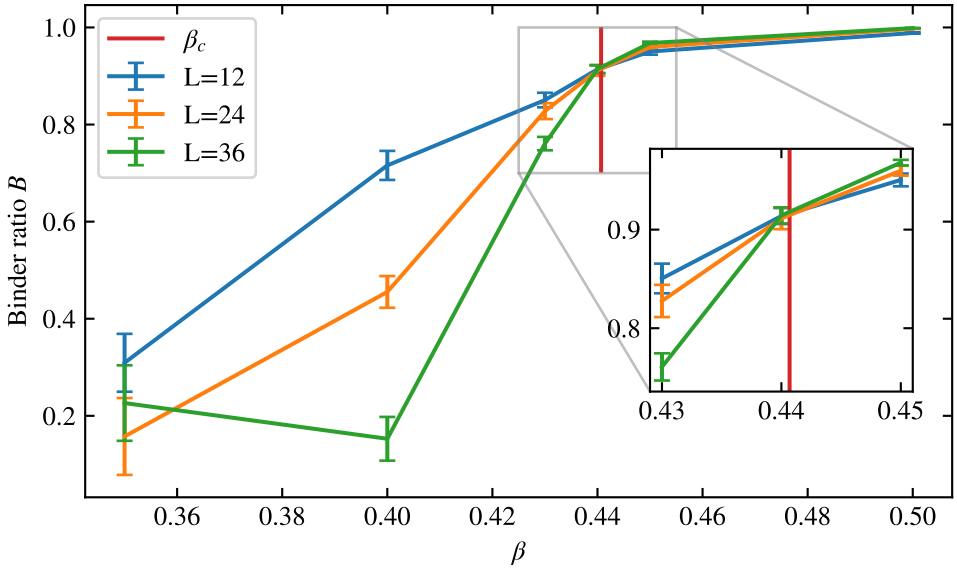


Fig. 1.11: Binder ratio for 2d Ising model at $J = 1$ for three lattice sizes, simulated with Wolff cluster updates. The crossing around $\beta \approx 0.44$ signals a phase transition, a result which is in accordance with the exact result by Onsager from 1944 [33].

1.1.3 Making a classical computer understand quantum models

Now that we know how to simulate classical models, the next challenge will be to apply the same approach to quantum models. In this context, the significant difference between classical and quantum models is that for classical models the energy of the system is a function of the configuration, while in a quantum model, the energy is determined through an operator. Meaning that for a classical model, each system configuration has a well-defined energy and statistical weight, while for a quantum model this is not the case.

As far as I am aware, there are generally two strategies for solving this problem, which both essentially start at the partition function $Z = \mathrm{Tr}(\exp(-\beta\hat{H}))$. Both express the partition function as a high-dimensional integral over a classical weight $w(C)$, that can be calculated through classical MCMC. This does not absolve the system from quantum-mechanical quirks (e.g. the famous sign problem can occur), but it makes the problem amenable to a classical computer.

The first strategy is based on a path integral formulation, while the other one uses the series expansion of the exponential function. The latter approach can be extremely efficient, if the operators are idempotent w.r.t. some power of the operator, as is e.g. the case for spin systems. The former approach, on the other hand, is more flexible and the one that ALF employs, which is why I will be focusing on it.

At its core, the path integral (also known as world line) approach maps the d dimensional quantum model to a $d+1$ dimensional (quasi) classical model, on which classical MCMC can be applied. The additional dimension is referred to as imaginary time, since translation along this axis is, up to a Wick rotation, like a time evolution in e.g. the Schödinger equation.

To demonstrate the concept on a relatively simple model, I am taking the already familiar Ising model and extend it to a quantum model by adding a field h perpendicular to the Ising spin orientation

$$\hat{H} = \underbrace{-J\sum_{\langle i,j\rangle}\hat{s}_i^z\hat{s}_j^z}_{\hat{H}_0} \underbrace{-h\sum_i \hat{s}_i^x}_{\hat{H}_h}.$$

Here, the operators $\hat{s}^z$ and $\hat{s}^x$ correspond to the Pauli matrices $s^z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}, s^x = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$, in the basis $|\uparrow\rangle = \begin{pmatrix}1\\0\end{pmatrix}, |\downarrow\rangle = \begin{pmatrix}0\\1\end{pmatrix}$.

This is (one of) the simplest interacting quantum models, known as the transverse field Ising model or the quantum Ising model. The transverse field introduces a tunneling between up and down spins and therefore disorder.

Since we are interested in thermodynamic quantities, we want to calculate the partition function

$$Z = \mathrm{Tr}\left(e^{-\beta \hat{H}}\right).$$

The first step of representing the partition function in terms of a classical model, is to rewrite the imaginary time evolution of length β into N_τ steps of length $\Delta_\tau = \beta / N_\tau$. By introducing $N_\tau - 1$ identities $\mathbb{1} = \sum_\Phi |\Phi\rangle \langle\Phi|$, the configuration space gets enhanced $\Phi \to C = (\Phi_1, \dots, \Phi_{N_\tau})$, to $d+1$ dimensions. We call the additional dimension imaginary time.

$$\begin{aligned}
Z &= \mathrm{Tr}\left(\exp\left(-\Delta_\tau \hat{H}\right)^{N_\tau}\right) \\
&= \sum_\Phi \langle\Phi| \exp\left(-\Delta_\tau \hat{H}\right) |\Phi\rangle && |\phi\rangle \equiv \bigotimes_i |s_i^z\rangle \\
&= \sum_C \prod_{n=1}^{N_\tau} \langle\Phi_n| \exp\left(-\Delta_\tau \hat{H}\right) |\Phi_{n+1}\rangle && \Phi_{N_\tau+1} = \Phi_1
\end{aligned}$$

The next step is a Trotter decomposition to separate the classical Ising part $\hat{H}_0$ from the transverse field $\hat{H}_h$, thereby introducing a systematic error $\mathcal{O}({\Delta_\tau}^2)$ [36].

$$\exp\left(-\Delta_\tau(\hat{H}_0 + \hat{H}_h)\right) = \exp\left(-\Delta_\tau \hat{H}_0\right) \exp\left(-\Delta_\tau \hat{H}_h\right) + \mathcal{O}({\Delta_\tau}^2)$$

$$Z = \sum_C \prod_{n=1}^{N_\tau} \langle\Phi_n| \exp\left(-\Delta_\tau \hat{H}_0\right) \exp\left(-\Delta_\tau \hat{H}_h\right) |\Phi_{n+1}\rangle + \mathcal{O}({\Delta_\tau}^2) \tag{1.11}$$

We now apply the classical Hamiltonian to the bra on the left, turning the operator back into a function:

$$Z = \sum_C \prod_{n=1}^{N_\tau} \exp\left(-\Delta_\tau H_0(\Phi_n)\right) \langle\Phi_n| \exp\left(-\Delta_\tau \hat{H}_h\right) |\Phi_{n+1}\rangle + \mathcal{O}({\Delta_\tau}^2).$$

The transverse field part factorizes into on-site terms whose matrix elements can be mapped to the exponential of an Ising interaction:

$$\langle\Phi_n| \exp\left(-\Delta_\tau \hat{H}_h\right) |\Phi_{n+1}\rangle = \prod_i \langle s_{i,n}^z| \exp\left(\Delta_\tau h \hat{s}_i^x\right) |s_{i,n+1}^z\rangle \tag{1.12}$$

$$\langle s_i^z| \exp\left(\Delta_\tau h \hat{s}^x\right) |s_j^z\rangle = \alpha \exp(\gamma s_i^z s_j^z) \qquad \begin{array}{l} \alpha = \sqrt{\sinh(h\Delta_\tau)\cosh(h\Delta_\tau)} \\ \gamma = -\frac{1}{2}\ln(\tanh h\Delta_\tau) \end{array}$$

Inserting this result back into the partition function, we can identify it as an anisotropic classical Ising model [up to a systematic error $\mathcal{O}({\Delta_\tau}^2)$] in $d+1$ dimensions with interaction strength $\Delta_\tau J/\beta$ in the original dimension and $-\frac{1}{2\beta}\ln(\tanh h\Delta_\tau)$ in the new dimension.

$$\begin{aligned}
Z &= \alpha^{NN_\tau} \sum_C \prod_{n=1}^{N_\tau} \left[\exp\left(-\Delta_\tau H_0(\Phi_n)\right) \exp\left(\gamma \sum_i s_{i,n}^z s_{i,n+1}^z\right)\right] + \mathcal{O}({\Delta_\tau}^2) \\
&= \alpha^{NN_\tau} \sum_C \exp(-\beta' H'(C)) + \mathcal{O}({\Delta_\tau}^2) \qquad \text{where} \qquad \beta' H' = -\sum_{\langle x,x'\rangle} J_{x,x'} s_x^z s_{x'}^z
\end{aligned}$$

$$J_{x,x'} = \begin{cases} \Delta_\tau J & \text{If } x \text{ and } x' \text{ are neighbors in space} \\ -\frac{1}{2}\ln(\tanh h\Delta_\tau) & \text{If } x \text{ and } x' \text{ are neighbors in } \tau \end{cases}$$

Omitting the systematic error from the Trotter decomposition, the model can now be simulated with classical MCMC, by sampling configurations C according to the weight $w(C) = \exp(-\beta' H'(C))$. This is a relatively simple example on how to simulate quantum models, but the general approach of enhancing the configuration space to get a statistical weight that is a function of the configuration is common to all QMC methods. Most methods include a Trotterisation that introduces a —usually well-controllable— systematic error. Note: There are also so-called continuous time methods that do not generate such a systematic error.

1.1.4 Negative sign problem

Most of the times, the resulting weight $w(C)$ from the step above can not be identified with a physical classical Hamiltonian $H'(C)$, but only some abstract action. More severely, in many models $w(C)$ can become non-positive, which results in the (in)famous sign problem.

The sign problem produces two problems: Firstly, we cannot interpret w as a probability weight any more, which prevents the direct use of importance sampling. This problem can be solved with reweighting, as I will show in a moment. More heavily weighs the second problem: As we will see after introducing reweighting, the sign problem leads to an exponential scaling of the computational effort with system size.

To recap, we want to use importance sampling to calculate something like

$$\langle O \rangle_w = \frac{\sum_C O(C) w(C)}{\sum_C w(C)}. \tag{1.13}$$

But if $w(C)$ can become negative or complex, a direct application is not possible. To solve this problem, one can absorb the sign of the weight in a function $\sigma(C)$, resulting in a modified weight $\tilde{w}(C) = \frac{w(C)}{\sigma(C)}$. This is a process known as *reweighting*. Usually, the reweighting chosen such that $\tilde{w}(C) = |w(C)|$ and therefore $\sigma(C) = \frac{w(C)}{|w(C)|}$, but principally any other choice resulting in $\tilde{w}(C) > 0$ works also.

With this, we can rewrite the expectation value Eq. (1.13) as an integration with respect to the new weight $\tilde{w}$:

$$\langle O \rangle_w = \frac{\sum_C O(C)\sigma(C)\tilde{w}(C)}{\sum_C \sigma(C)\tilde{w}(C)} = \frac{\langle \sigma O \rangle_{\tilde{w}}}{\langle \sigma \rangle_{\tilde{w}}} \tag{1.14}$$

The average sign $\langle \sigma \rangle_{\tilde{w}}$ measures the severity of the sign problem: The smaller it is, the larger are the fluctuations of Eq. (1.14), decreasing the precision of results. In the common choice $|\tilde{w}(C)| = 1$, the absolute value of the average sign ranges from zero to one, where one is a sign problem free simulation and zero is not feasible.

The average sign can be understood as the ratio of two partition functions

$$\langle \sigma \rangle_{\tilde{w}} = \frac{\sum_C w(C)}{\sum_C \tilde{w}(C)} = \frac{Z}{\tilde{Z}}.$$

Since a partition function scales asymptotically as an exponential of the $d+1$ dimensional volume of the simulated model, the same is also true for the average sign. Meaning $\langle \sigma \rangle_{\tilde{w}} \sim \exp(-f\beta V)$, where f has the unit of an energy density, V is the d dimensional volume and β the reciprocal temperature and thereby the size of the additional dimension. Therefore, to keep the same precision, the computational effort scales exponentially with the system size in the presence of a sign problem. But all is not lost! The sign problem is basis dependent, i.e. in some simulations with a sign problem another representation might exist that solves the sign problem. Furthermore, the sign problem can be more and less severe so that systems with a relatively big sign, larger systems sizes could be simulated with reasonable effort.

1.1.5 Auxiliary field QMC

In the last part of this section, I will take a cursory glance at simulating fermions with a class of methods called *auxiliary field quantum Monte Carlo*.

The idea in auxiliary field QMC is to express the interacting model as free fermions interacting with a bosonic field. This way, one can integrate out the fermions analytically and the bosonic field can be sampled with Monte Carlo. Most commonly, the auxiliary field is generated through a Hubbard-Stratonovich transformation, but it can also be designed directly into the model, as is done for example through the Yukawa coupling in Chapter 2.

A Hubbard-Stratonovich transformation is essentially just a Gauss integral, but backwards:

$$\exp\left(\hat{A}^2\right) = \frac{1}{\sqrt{4\pi}} \int_{-\infty}^{\infty} dx \exp\left(-\frac{x^2}{4} - x\hat{A}\right),$$

where $\hat{A}^2$ might for example be a Hubbard interaction $\hat{A}^2 = \left[\sum_{\sigma=\uparrow,\downarrow}(\hat{c}_\sigma^\dagger \hat{c}_\sigma - 1/2)\right]^2$.[14]

The transformation works with the exponential of an operator, so to apply it on the partition function, we have to separate each operator through a Trotter decomposition. This introduces a systematic error for non-commuting operators. The decomposition is executed in analogous way to the transverse-field Ising model in Section 1.1.3. After doing that and mapping the system to $d+1$ dimensions in the process, the partition function can be written as

$$Z = \mathrm{Tr}\left[\exp(-\beta \hat{H})\right] = \mathrm{Tr}_{C,\hat{\boldsymbol{c}}^\dagger,\hat{\boldsymbol{c}}}\left[\exp\left(S_\mathrm{B}(C) + \hat{\boldsymbol{c}}^\dagger V(C)\hat{\boldsymbol{c}}\right)\right].$$

Here, C is the $d+1$ dimensional bosonic configuration, $\hat{\boldsymbol{c}}^\dagger$ ($\hat{\boldsymbol{c}}$) are vectors containing all fermionic creation (annihilation) operators in $d+1$ dimensional space. S_B is the purely bosonic contribution to the action and V is a matrix encoding the structure of the fermions. Since the fermions are non-interacting, they can be integrated out by performing the trace over the Fock space, leaving a purely bosonic theory. This results in a determinant:

$$Z = \mathrm{Tr}_C\left[\exp\left(S_\mathrm{B}(C)\right)\det(V(C))\right].$$

The "only" challenge remaining is to find an efficient way to sample C. Then we can calculate observables as:

$$\langle O \rangle = \frac{\sum_C O(C)w(C)}{\sum_C w(C)} \qquad w(c) = \exp\left(S_\mathrm{B}(C)\right)\det(V(C))$$

In practice, the execution is of course not that trivial. In particular because the determinant is very expensive to calculate. One very versatile algorithm has been developed by Blankenbecler, Scalapino and Sugar [8], which is the algorithm ALF is using. This has been my introduction to the Monte Carlo method. For more details on the algorithm and implementation used throughout this work, see [8, 9, 10, 12].

The next chapter starts with the presentation of my actual research projects.

---

[14] (a) This Hubbard term is not in the more well-known form $\hat{c}_\uparrow^\dagger \hat{c}_\uparrow \hat{c}_\downarrow^\dagger \hat{c}_\downarrow$, but if we perform the square, we get this term plus a chemical potential (and a constant), which can be subtracted in an additional single particle term.

(b) ALF uses a discrete equivalent of this transformation, summarizing over four field values instead of a real numbers. This introduces a systematic error $\mathcal{O}(\Delta_\tau^4)$, but it is negligible to the systematic Trotter error $\mathcal{O}(\Delta_\tau^2)$.

Projects

CHAPTER

TWO

NEMATIC QUANTUM CRITICALITY IN DIRAC SYSTEMS

The results presented in this chapter are the outcome of a collaboration with Lukas Janssen, Kai Sun, Zi Yang Meng, Igor F. Herbut, Matthias Vojta, and Fakher F. Assaad. These findings have been published in [14], with significant portions reproduced here verbatim. My contributions to the project comprise the quantum Monte Carlo (QMC) and mean field calculations, including the implementation of the models in code and the creation of corresponding figures. The ϵ-expansion has been performed by L. J., the models are designed by F. F. A. and K. S., while the interpretation of data and written text is a combined work of all authors.

2.1 Introduction

In a strongly correlated electron system, global symmetries, such as spin rotation, point group, or translational symmetries, can be spontaneously broken as a function of some external tuning parameter. This challenging problem has been studied extensively numerically and experimentally over the last years and impacts our understanding of quantum criticality [37] in cuprates [38] and heavy fermions [39]. The problem greatly simplifies when the Fermi surface reduces to isolated Fermi points in $2+1$ dimensions and the critical point features emergent Lorentz symmetry. In this context, spin, time reversal, and translational symmetry breaking generically correspond to the dynamical generation of mass terms [40], and the semimetal-to-insulator transition belongs to one of the various Gross-Neveu universality classes [41, 42, 43, 44, 45, 46, 47].

Across nematic transitions, rotational symmetry is spontaneously broken [48, 49]. For continuum Dirac fermions with Hamiltonian $H(\boldsymbol{k}) = v\left(k_x\sigma_x + k_y\sigma_y\right)$ in momentum space, where $\boldsymbol{\sigma}$ are Pauli spin matrices and v is the Fermi velocity, nematic transitions correspond to the dynamical generation of nonmass terms, such as $m\sigma_x$. They shift the position of the Dirac cone and as such break rotational, and therewith also Lorentz, symmetries. Such nematic transitions have been studied theoretically in the past in the context of d-wave superconductors [50, 51, 52, 53, 54] and bilayer graphene [55]. Fundamental questions pertaining to the very nature of the transition remain open: While initial renormalization group (RG) calculations based on the ϵ expansion suggested a first-order transition [50, 51], a continuous transition has been found in large-N analyses [52, 53]. In this work, we use quantum Monte Carlo (QMC) simulations and a revised ϵ-expansion analysis to study these transitions. We introduce two different models of Dirac fermions with twofold and fourfold lattice rotational symmetries, respectively. They are designed to feature a nematic transition, tuned by a parameter h. The fermion dispersion of these models and the meandering of their Dirac cones are shown in Fig. 2.1(a). We demonstrate numerically and analytically that for both models this transition is continuous, realizing a new family of quantum universality classes in Dirac systems *without* emergent Lorentz invariance.

This chapter is organized as follows. In Sec. 2.2 we define the models and derive their symmetries. In Sec. 2.3 we perform a mean-field analysis of the models. In Sec. 2.4 we derive the continuum-field theories of the models. In Sec. 2.5 we perform an ϵ-expansion analysis of these field theories, finding a continuous transition for both models, where *App. A.1* contains more details on the analysis. In Sec. 2.6 we introduce the numerical method used for the QMC simulations and prove the absence of the negative sign problem. In Sec. 2.8 we define the QMC observables and show numerical results confirming a continuous transition for both models. In Sec. 2.9 we summarize our results. Furthermore, *App. A.2* demonstrates how to use pyALF (cf. Section 4) to reproduce the QMC results of this chapter. *App. A.3* and *App. A.4* display the source code used for data collapse and retrieving the fermion dispersions, respectively. Finally, *App. A.5* shows that the results from Sec. 2.8 do not change qualitatively when varying the number of spin degrees of freedom N_σ or the coupling parameter ξ.

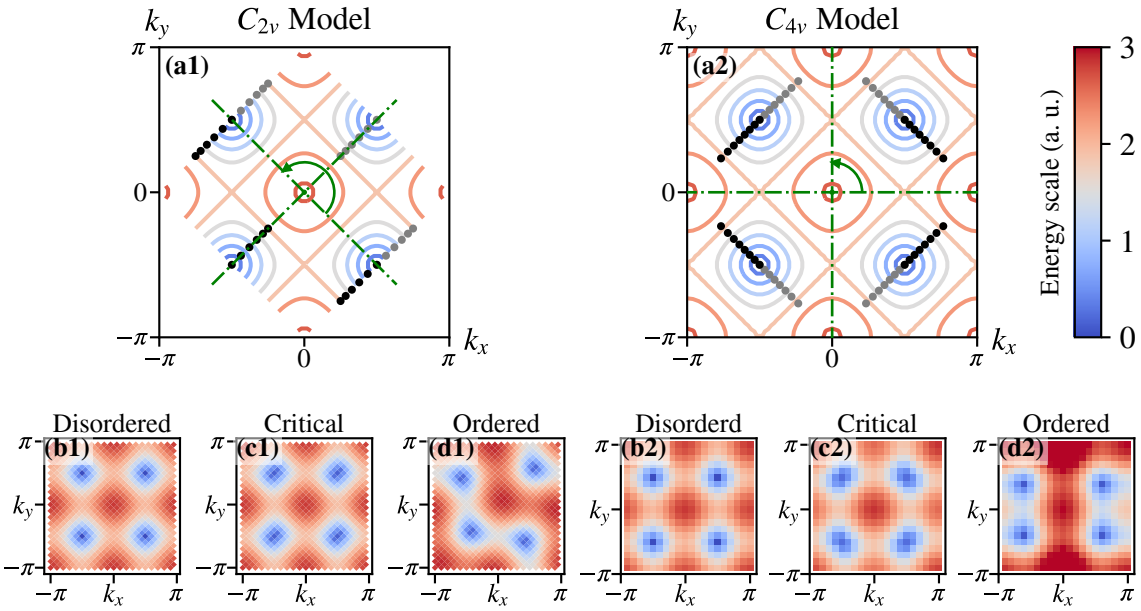


Fig. 2.1: (a) Contour plot of the fermion dispersion in the disordered phase from mean-field theory. The plotted area indicates the Brillouin zone. The green lines and arrows indicate the point group symmetries. Black (gray) dots sketch the meandering of the Dirac cones in the nematic phase for $\langle \hat{s}^z_{\boldsymbol{R}} \rangle > 0$ (< 0). (b)-(d) Fermion dispersion from QMC at $L = 20$ for (b) $h = 5.0 > h_c$ featuring isotropic Dirac cones (c) $h \simeq h_c$, $h \approx 3.27$ (left) and $h \approx 3.65$ (right) featuring anisotropic Dirac cones, and (d) at $h = 1.0 < h_c$ featuring broken point-group symmetries. Color scale applies to all plots.

2.2 Models

Inspired from Refs. [49, 56, 57], we design two models of (2+1)-dimensional Dirac fermions, $\mathcal{H}_0$, coupled to a transverse-field Ising model (TFIM),

$$\mathcal{H}_{\text{Ising}} = -J \sum_{\langle \boldsymbol{R}, \boldsymbol{R}' \rangle} \hat{s}^z_{\boldsymbol{R}} \hat{s}^z_{\boldsymbol{R}'} - h \sum_{\boldsymbol{R}} \hat{s}^x_{\boldsymbol{R}}, \tag{2.1}$$

where $\boldsymbol{R}$ denotes a unit cell and $\langle \boldsymbol{R}, \boldsymbol{R}' \rangle$ runs over adjacent unit cells. A Yukawa coupling, $\mathcal{H}_{\text{Yuk}}$, between the Ising field and a nematic fermion bilinear yields the desired models, $\mathcal{H} = \mathcal{H}_0 + \mathcal{H}_{\text{Ising}} + \mathcal{H}_{\text{Yuk}}$, that correspond to one of many possible lattice regularizations of the continuum field theories of Eqs. (2.18) and (2.19).

In the C_{2v} model, depicted in Fig. 2.2(a), we employ a π-flux Hamiltonian on the square lattice as

$$\begin{aligned} \mathcal{H}_0^{C_{2v}} = -t \sum_{\boldsymbol{R}} \sum_{\sigma=1}^{N_\sigma} \hat{a}^\dagger_{\boldsymbol{R},\sigma} \Big(& \hat{b}_{\boldsymbol{R},\sigma} e^{-i\frac{\pi}{4}} + \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_-,\sigma} e^{i\frac{\pi}{4}} \\ & + \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_- - \boldsymbol{e}_+,\sigma} e^{-i\frac{\pi}{4}} + \hat{b}_{\boldsymbol{R}-\boldsymbol{e}_+,\sigma} e^{i\frac{\pi}{4}} \Big) + \text{H.c.}, \end{aligned} \tag{2.2}$$

where $\hat{a}$ and $\hat{b}$ with spin index σ are fermion annihilation operators on the two sublattices, t is the hopping parameter, and $N_\sigma = 2$ is the number of spin degrees of freedom. $\mathcal{H}_0$ features two inequivalent Dirac points per spin component in the Brillouin zone (BZ). The Ising spins $\hat{s}_{\boldsymbol{R}}$ couple, with the sign structure indicated in Fig. 2.2(a), to the nearest-neighbor fermion hopping terms,

$$\begin{aligned} \mathcal{H}_{\text{Yuk}}^{C_{2v}} = -\xi \sum_{\boldsymbol{R}} \sum_{\sigma=1}^{N_\sigma} \hat{s}^z_{\boldsymbol{R}} \hat{a}^\dagger_{\boldsymbol{R},\sigma} \Big(& \hat{b}_{\boldsymbol{R},\sigma} e^{-i\frac{\pi}{4}} - \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_-,\sigma} e^{i\frac{\pi}{4}} \\ & - \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_- - \boldsymbol{e}_+,\sigma} e^{-i\frac{\pi}{4}} + \hat{b}_{\boldsymbol{R}-\boldsymbol{e}_+,\sigma} e^{i\frac{\pi}{4}} \Big) + \text{H.c.}, \end{aligned} \tag{2.3}$$

where ξ denotes the coupling strength.

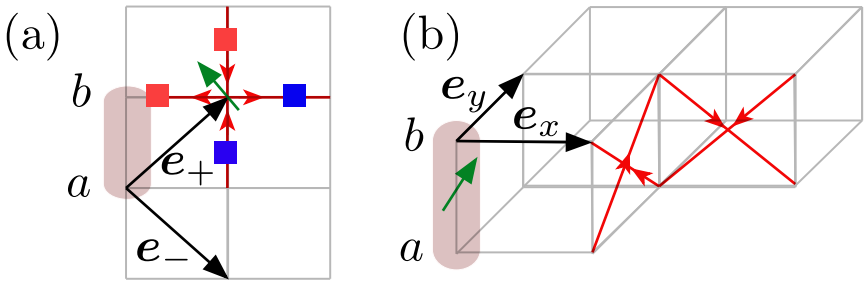


Fig. 2.2: Sketch of (a) C_{2v} and (b) C_{4v} models, defined on π-flux single-layer and bilayer square lattices, with lattice vectors $\boldsymbol{e}_{+/-}$ and $\boldsymbol{e}_{x/y}$, respectively. Dark pink regions indicate unit cells, containing two orbitals (a and b) and one Ising spin (green arrow) in both cases. Fermions hop along the red lines and acquire a phase factor $e^{i\pi/4}$ when following the direction of the arrow. Red and blue squares in (a) indicate the sign structure in the Yukawa coupling of the C_{2v} model.

The model has a C_{2v} point group symmetry, composed of reflections, $\hat{T}_\pm$ on the $\boldsymbol{e}_\pm = \boldsymbol{e}_x \pm \boldsymbol{e}_y$ axis. $\hat{T}_\pm$ pins the Dirac cones to the $\boldsymbol{K}_\pm = (\pi/2, \pm\pi/2)$ points in the BZ. Aside from the above reflections, π rotations about the z axis are obtained as $\hat{T}_+\hat{T}_-$. Further, the model exhibits an explicit SU(N_σ) spin symmetry that is enlarged to O($2N_\sigma$) (cf. Section 2.6.1.1).

The C_{4v} model corresponds to a bilayer π-flux model, in which the Ising spins are located on the rungs, Fig. 2.2(b). The fermion hopping Hamiltonian is

$$\begin{aligned}\mathcal{H}_0^{C_{4v}} = -t\sum_{\boldsymbol{R}}\sum_{\sigma=1}^{N_\sigma}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\Big(&\hat{b}_{\boldsymbol{R}+\boldsymbol{e}_x,\sigma}e^{i\frac{\pi}{4}} + \hat{b}_{\boldsymbol{R}-\boldsymbol{e}_x,\sigma}e^{i\frac{\pi}{4}} \\ &+ \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_y,\sigma}e^{-i\frac{\pi}{4}} + \hat{b}_{\boldsymbol{R}-\boldsymbol{e}_y,\sigma}e^{-i\frac{\pi}{4}}\Big) + \text{H.c.},\end{aligned} \tag{2.4}$$

featuring four Dirac cones per spin component. The Yukawa coupling reads

$$\mathcal{H}_{\text{Yuk}}^{C_{4v}} = -\xi\sum_{\boldsymbol{R}}\sum_{\sigma=1}^{N_\sigma} i\hat{s}^z_{\boldsymbol{R}}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\hat{b}_{\boldsymbol{R},\sigma} + \text{H.c.}, \tag{2.5}$$

amounting to a coupling of the Ising spins to the interlayer fermion current. The C_{4v} Hamiltonian commutes with $\hat{T}_{\pi/2}$, corresponding to $\pi/2$ rotation about the z axis. The model is invariant under reflections $\hat{T}_x$ and $\hat{T}_y$ along the x and y axes, respectively. Reflections along $\boldsymbol{e}_\pm = \boldsymbol{e}_x \pm \boldsymbol{e}_y$, denoted by $\hat{T}_\pm$, can be derived from $\hat{T}_{\pi/2}$, $\hat{T}_x$, and $\hat{T}_y$, and therefore also leave the model invariant. The model hence has a C_{4v} symmetry. Particle-hole symmetry, imposes $A(\boldsymbol{k},\omega) = A(-\boldsymbol{k}+\boldsymbol{Q},-\omega)$, where $\boldsymbol{Q} = (\pi,\pi)$ such that alongside with the C_{4v} symmetry the Dirac cones are pinned to the $\pm\boldsymbol{K}_\pm$ points in the BZ (cf. Section 2.2.2).

2.2.1 Fourier transformed models

We define the Fourier transformation as:

$$\begin{aligned}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{R},\sigma}\end{pmatrix} &= \frac{1}{\sqrt{N}}\sum_{\boldsymbol{k}} e^{-i\boldsymbol{k}\boldsymbol{R}}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{k},\sigma}\end{pmatrix} \\ s^z_{\boldsymbol{R}} &= \frac{1}{N}\sum_{\boldsymbol{q}} s^z_{\boldsymbol{q}}e^{i\boldsymbol{q}\boldsymbol{R}}\end{aligned} \tag{2.6}$$

with this definition, both models take the form

$$\mathcal{H} = -\sum_{\sigma=1}^{N_\sigma}\sum_{\boldsymbol{k}}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\left(\hat{b}_{\boldsymbol{k},\sigma}Z_0(\boldsymbol{k}) + \frac{\xi}{N}\sum_{\boldsymbol{q}}\hat{b}_{\boldsymbol{k}-\boldsymbol{q},\sigma}s^z_{\boldsymbol{q}}Z_{\text{Yuk}}(\boldsymbol{k})\right) + \text{H.c.} + H_{\text{Ising}} \tag{2.7}$$

with

$$Z_0(\boldsymbol{k}) = \begin{cases} 2t\left(e^{i\frac{\pi}{4}}\cos k_x + e^{-i\frac{\pi}{4}}\cos k_y\right)e^{-i\boldsymbol{k}_y} & C_{2v}\text{ model} \\ 2t\left(e^{i\frac{\pi}{4}}\cos k_x + e^{-i\frac{\pi}{4}}\cos k_y\right) & C_{4v}\text{ model} \end{cases}$$
$$Z_{\text{Yuk}}(\boldsymbol{k}) = \begin{cases} i2\left(-e^{i\frac{\pi}{4}}\sin k_x + e^{-i\frac{\pi}{4}}\sin k_y\right)e^{-i\boldsymbol{k}_y} & C_{2v}\text{ model} \\ i & C_{4v}\text{ model.} \end{cases}$$

2.2.2 Symmetries

2.2.2.1 The C_{2v} model

The First model has a C_{2v} symmetry, consisting of two reflections: T_+ and T_- on $\boldsymbol{e}_\pm = \boldsymbol{e}_x \pm \boldsymbol{e}_y$, the π rotation needed by the point group can be obtained as $T_\pi = T_+ \cdot T_-$. The T_- invariance hinges on the Z_2 Ising symmetry, $s^z \to -s^z$, and is therefore broken in the ordered phase.

The C_{2v} symmetry pins the Dirac points (up to a gauge choice) to $(\pi/2, \pm\pi/2)$, while in the ordered phase, meandering parallel to $\boldsymbol{e}_+$ is possible.

To show this symmetry, we expand the momentum- and real-space vectors as: $\boldsymbol{k} = k_+\boldsymbol{e}_+ + k_-\boldsymbol{e}_-$ and $\boldsymbol{R} = R_+\boldsymbol{e}_+ + R_-\boldsymbol{e}_-$.

The first reflection T_+ reads:

$$\begin{gathered}\hat{T}_+^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{k},\sigma}\end{pmatrix}\hat{T}_+ = \begin{pmatrix}\hat{b}^\dagger_{(k_+,-k_-),\sigma}\\ \hat{a}^\dagger_{(k_+,-k_-),\sigma}e^{-ik_+}\end{pmatrix} \\ \hat{T}_+^{-1}s^z_{\boldsymbol{q}}\hat{T}_+ = s^z_{(q_+,-q_-)}\end{gathered} \tag{2.8}$$

Inserting the above in Eq. (2.7), we obtain:

$$\begin{aligned}&\hat{T}_+^{-1}\mathcal{H}^{C_{2v}}\hat{T}_+ = \\ &= -\sum_{\boldsymbol{k}} e^{ik_+}\hat{b}^\dagger_{(k_+,-k_-),\sigma}\left(\hat{a}_{(k_+,-k_-),\sigma}Z_0^{C_{2v}}(\boldsymbol{k}) + \xi\sum_{\boldsymbol{q}}\hat{a}_{(k_+-q_+,-k_-+q_-),\sigma}\hat{s}^z_{(q_+,-q_-)}Z_{\text{Yuk}}^{C_{2v}}(\boldsymbol{k})\right) + \text{H.c.} + H_{\text{Ising}} \\ &= -\sum_{\boldsymbol{k}} e^{-ik_+}\hat{a}^\dagger_{(k_+,k_-),\sigma}\left(\hat{b}_{(k_+,k_-),\sigma}\bar{Z}_0^{C_{2v}}(k_+,-k_-) + \xi\sum_{\boldsymbol{q}}\hat{b}_{(k_+-q_+,k_--q_-),\sigma}\hat{s}^z_{(q_+,q_-)}\bar{Z}_{\text{Yuk}}^{C_{2v}}(k_+,-k_-)\right) + \text{H.c.} + H_{\text{Ising}} \\ &\quad\begin{bmatrix}\bar{Z}_0^{C_{2v}}(k_+,-k_-) & = Z_0^{C_{2v}}(\boldsymbol{k})e^{ik_+}\\ \bar{Z}_{\text{Yuk}}^{C_{2v}}(k_+,-k_-) & = Z_{\text{Yuk}}^{C_{2v}}(\boldsymbol{k})e^{ik_+}\end{bmatrix} \\ &= -\sum_{\boldsymbol{k}}\hat{a}^\dagger_{(k_+,k_-),\sigma}\left(\hat{b}_{(k_+,k_-),\sigma}Z_0^{C_{2v}}(\boldsymbol{k}) + \xi\sum_{\boldsymbol{q}}\hat{b}_{\boldsymbol{k}-\boldsymbol{q},\sigma}\hat{s}^z_{\boldsymbol{q}}Z_{\text{Yuk}}^{C_{2v}}(\boldsymbol{k})\right) + \text{H.c.} + H_{\text{Ising}} \\ &= \mathcal{H}^{C_{2v}}\end{aligned}$$

In real space, $\hat{T}_+$ translates to:

$$\begin{gathered}\hat{T}_+^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{R},\sigma}\end{pmatrix}\hat{T}_+ = \begin{pmatrix}\hat{b}^\dagger_{(R_+,-R_-),\sigma}\\ \hat{a}^\dagger_{(R_++1,-R_-),\sigma}\end{pmatrix} \\ \hat{T}_+^{-1}s^z_{\boldsymbol{R}}\hat{T}_+ = s^z_{(R_+,-R_-)}\end{gathered}$$

The second reflection T_- can be expressed as:

$$\begin{gathered}\hat{T}_-^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{k},\sigma}\end{pmatrix}\hat{T}_- = \begin{pmatrix}\hat{b}^\dagger_{(-k_+,k_-),\sigma}\\ \hat{a}^\dagger_{(-k_+,k_-),\sigma}e^{ik_-}\end{pmatrix} \\ \hat{T}_-^{-1}s^z_{\boldsymbol{q}}\hat{T}_- = -s^z_{(-q_+,q_-)}\end{gathered} \tag{2.9}$$

Inserting this into Eq. (2.7), we obtain:

$$
\begin{aligned}
&\hat{T}_-^{-1}\mathcal{H}^{C_{2v}}\hat{T}_- = \\
&= -\sum_{\boldsymbol{k}} e^{-ik_-}\hat{b}^\dagger_{(-k_+,k_-),\sigma}\left(\hat{a}_{(-k_+,k_-),\sigma}Z_0(\boldsymbol{k}) + \xi\sum_{\boldsymbol{q}}\hat{a}_{(-k_++q_+,k_--q_-),\sigma}(-\hat{s}^z_{(-q_+,q_-)})Z_{\mathrm{I}}(\boldsymbol{k})\right) + \mathrm{H.c.} + H_{\mathrm{Ising}} \\
&= -\sum_{\boldsymbol{k}} e^{ik_-}\hat{a}^\dagger_{(k_+,k_-),\sigma}\left(\hat{b}_{(k_+,k_-),\sigma}\bar{Z}_0(-k_+,k_-) + \xi\sum_{\boldsymbol{q}}\hat{b}_{(k_+-q_+,k_--q_-),\sigma}(-\hat{s}^z_{(q_+,q_-)})\bar{Z}_{\mathrm{I}}(-k_+,k_-)\right) + \mathrm{H.c.} + H_{\mathrm{Ising}} \\
&\quad \begin{bmatrix} \bar{Z}_0(-k_+,k_-) & = Z_0^{C_{2v}}(\boldsymbol{k})e^{-ik_-} \\ \bar{Z}_{\mathrm{I}}(-k_+,k_-) & = -Z_{\mathrm{Yuk}}^{C_{2v}}(\boldsymbol{k})e^{-ik_-} \end{bmatrix} \\
&= -\sum_{\boldsymbol{k}} \hat{a}^\dagger_{\boldsymbol{k},\sigma}\left(\hat{b}_{\boldsymbol{k},\sigma}Z_0^{C_{2v}}(\boldsymbol{k}) + \xi\sum_{\boldsymbol{q}}\hat{b}_{\boldsymbol{k}-\boldsymbol{q}}\hat{s}^z_{\boldsymbol{q}}Z_{\mathrm{Yuk}}^{C_{2v}}(\boldsymbol{k})\right) + \mathrm{H.c.} + H_{\mathrm{Ising}} \\
&= \mathcal{H}^{C_{2v}}
\end{aligned}
$$

In real space, $\hat{T}_-$ translates to:

$$
\begin{gathered}
\hat{T}_-^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{R},\sigma}\end{pmatrix}\hat{T}_- = \begin{pmatrix}\hat{b}^\dagger_{(-R_+,R_-),\sigma}\\ \hat{a}^\dagger_{(-R_+,R_--1),\sigma}\end{pmatrix} \\
\hat{T}_-^{-1}s^z_{\boldsymbol{R}}\hat{T}_- = s^z_{(-R_+,R_-)}
\end{gathered}
$$

2.2.2.2 The C_{4v} model

The Second model has a C_{4v} symmetry, consisting of a rotation by $\frac{\pi}{2}$ and reflections on the x- and y-axis.

The corresponding operators in momentum space are:

$$
\begin{gathered}
\hat{T}_{\pi/2}^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{k},\sigma}\end{pmatrix}\hat{T}_{\pi/2} = \begin{pmatrix}\hat{b}^\dagger_{(-k_y,k_x),\sigma}\\ \hat{a}^\dagger_{(-k_y,k_x),\sigma}\end{pmatrix} \quad \hat{T}_x^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{k},\sigma}\end{pmatrix}\hat{T}_x = \begin{pmatrix}\hat{a}^\dagger_{(k_x,-k_y),\sigma}\\ \hat{b}^\dagger_{(k_x,-k_y),\sigma}\end{pmatrix} \quad \hat{T}_y^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{k},\sigma}\end{pmatrix}\hat{T}_y = \begin{pmatrix}\hat{a}^\dagger_{(-k_x,k_y),\sigma}\\ \hat{b}^\dagger_{(-k_x,k_y),\sigma}\end{pmatrix} \\
\hat{T}_{\pi/2}^{-1}\hat{s}^z_{\boldsymbol{q}}\hat{T}_{\pi/2} = -\hat{s}^z_{(-q_y,q_x)} \qquad \hat{T}_x^{-1}\hat{s}^z_{\boldsymbol{q}}\hat{T}_x = \hat{s}^z_{(q_x,-q_y)} \qquad \hat{T}_y^{-1}\hat{s}^z_{\boldsymbol{q}}\hat{T}_y = \hat{s}^z_{(-q_x,q_y)}
\end{gathered}
$$

And in real space:

$$
\begin{gathered}
\hat{T}_{\pi/2}^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{R},\sigma}\end{pmatrix}\hat{T}_{\pi/2} = \begin{pmatrix}\hat{b}^\dagger_{(-R_y,R_x),\sigma}\\ \hat{a}^\dagger_{(-R_y,R_x),\sigma}\end{pmatrix} \quad \hat{T}_x^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{R},\sigma}\end{pmatrix}\hat{T}_x = \begin{pmatrix}\hat{a}^\dagger_{(R_x,-R_y),\sigma}\\ \hat{b}^\dagger_{(R_x,-R_y),\sigma}\end{pmatrix} \quad \hat{T}_y^{-1}\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{R},\sigma}\end{pmatrix}\hat{T}_y = \begin{pmatrix}\hat{a}^\dagger_{(-R_x,R_y),\sigma}\\ \hat{b}^\dagger_{(-R_x,R_y),\sigma}\end{pmatrix} \\
\hat{T}_{\pi/2}^{-1}\hat{s}^z_{\boldsymbol{R}}\hat{T}_{\pi/2} = -\hat{s}^z_{(-R_y,R_x)} \qquad \hat{T}_x^{-1}\hat{s}^z_{\boldsymbol{R}}\hat{T}_x = \hat{s}^z_{(R_x,-R_y)} \qquad \hat{T}_y^{-1}\hat{s}^z_{\boldsymbol{R}}\hat{T}_y = \hat{s}^z_{(-R_x,R_y)}
\end{gathered}
$$

In the Ising ordered phase, the Ising symmetry $\hat{s}^z_{\boldsymbol{R}} \to -\hat{s}^z_{\boldsymbol{R}}$ is broken, which reduces $\hat{T}_{\pi/2}$ to $\hat{T}_\pi$, such that the C_{4v} symmetry is reduced to C_{2v}. This reduced symmetry allows the cones to meander.

Particle-hole symmetry:

The particle-hole symmetry $\hat{T}_{\mathrm{ph}}$ implies that energy eigenstates satisfy $E(\boldsymbol{k}) = -E(-\boldsymbol{k}+\boldsymbol{Q})$, $\boldsymbol{Q} = (\pi,\pi)$.

$$
\begin{gathered}
\hat{T}_{\mathrm{ph}}^{-1}\alpha\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{R},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{R},\sigma}\end{pmatrix}\hat{T}_{\mathrm{ph}} = \bar{\alpha}(-1)^{R_x+R_y}\begin{pmatrix}\hat{a}_{\boldsymbol{R},\sigma}\\ \hat{b}_{\boldsymbol{R},\sigma}\end{pmatrix} \\
\hat{T}_{\mathrm{ph}}^{-1}\alpha s^z_{\boldsymbol{R}}\hat{T}_{\mathrm{ph}} = -\bar{\alpha}s^z_{\boldsymbol{R}}
\end{gathered} \tag{2.10}
$$

$$
\begin{gathered}
\hat{T}_{\mathrm{ph}}^{-1}\alpha\begin{pmatrix}\hat{a}^\dagger_{\boldsymbol{k},\sigma}\\ \hat{b}^\dagger_{\boldsymbol{k},\sigma}\end{pmatrix}\hat{T}_{\mathrm{ph}} = \bar{\alpha}\begin{pmatrix}\hat{a}_{\boldsymbol{Q}-\boldsymbol{k},\sigma}\\ \hat{b}_{\boldsymbol{Q}-\boldsymbol{k},\sigma}\end{pmatrix} \quad \boldsymbol{Q} = \begin{pmatrix}\pi\\ \pi\end{pmatrix} \\
\hat{T}_{\mathrm{ph}}^{-1}\alpha s^z_{\boldsymbol{q}}\hat{T}_{\mathrm{ph}} = -\bar{\alpha}s^z_{-\boldsymbol{q}}
\end{gathered} \tag{2.11}
$$

Inserting this in Eq. (2.7), we obtain:

$$\begin{aligned}
\hat{T}_{\text{ph}}^{-1}\mathcal{H}^{C_{4v}}\hat{T}_{\text{ph}} &= -\sum_{\boldsymbol{k}}\hat{a}_{\boldsymbol{Q}-\boldsymbol{k}}\left(\hat{b}^{\dagger}_{\boldsymbol{Q}-\boldsymbol{k}}\bar{Z}_0^{C_{4v}}(\boldsymbol{k})+\frac{\xi}{N}\sum_{\boldsymbol{q}}\hat{b}^{\dagger}_{\boldsymbol{Q}-\boldsymbol{k}+\boldsymbol{q}}\left(-s^z_{-\boldsymbol{q}}\right)\bar{Z}_{\text{Yuk}}^{C_{4v}}(\boldsymbol{k})\right)+\text{H.c.}+H_{\text{Ising}}\\
&= \sum_{\boldsymbol{k}}\hat{a}^{\dagger}_{\boldsymbol{Q}-\boldsymbol{k}}\left(\hat{b}_{\boldsymbol{Q}-\boldsymbol{k}}Z_0^{C_{4v}}(\boldsymbol{k})+\frac{\xi}{N}\sum_{\boldsymbol{q}}\hat{b}_{\boldsymbol{Q}-\boldsymbol{k}+\boldsymbol{q}}\left(-s^z_{-\boldsymbol{q}}\right)Z_{\text{Yuk}}^{C_{4v}}(\boldsymbol{k})\right)+\text{H.c.}+H_{\text{Ising}}\\
&= \sum_{\boldsymbol{k}}\hat{a}^{\dagger}_{\boldsymbol{k}}\left(\hat{b}_{\boldsymbol{k}}Z_0^{C_{4v}}(\boldsymbol{Q}-\boldsymbol{k})+\frac{\xi}{N}\sum_{\boldsymbol{q}}\hat{b}_{\boldsymbol{k}-\boldsymbol{q}}\left(-s^z_{\boldsymbol{q}}\right)Z_{\text{Yuk}}^{C_{4v}}(\boldsymbol{Q}-\boldsymbol{k})\right)+\text{H.c.}+H_{\text{Ising}}\\
&\quad\left[\begin{array}{ll} Z_0^{C_{4v}}(\boldsymbol{Q}-\boldsymbol{k}) & =-Z_0^{C_{4v}}(\boldsymbol{k})\\ Z_{\text{Yuk}}^{C_{4v}}(\boldsymbol{Q}-\boldsymbol{k}) & =Z_{\text{Yuk}}^{C_{4v}}(\boldsymbol{k})\end{array}\right]\\
&= -\sum_{\boldsymbol{k}}\hat{a}^{\dagger}_{\boldsymbol{k}}\left(\hat{b}_{\boldsymbol{k}}Z_0^{C_{4v}}(\boldsymbol{k})+\frac{1}{N}\sum_{\boldsymbol{q}}\hat{b}_{\boldsymbol{k}-\boldsymbol{q}}\left(s^z_{\boldsymbol{q}}\right)Z_{\text{Yuk}}^{C_{4v}}(\boldsymbol{k})\right)+\text{H.c.}+H_{\text{Ising}}\\
&= \mathcal{H}^{C_{4v}}
\end{aligned}$$

As a result of this symmetry, the single particle spectral function satisfies

$$A(\boldsymbol{k},\omega)=A(-\boldsymbol{k}+\boldsymbol{Q},-\omega),\ \text{with}\ \boldsymbol{Q}=(\pi,\pi).$$

2.3 Lattice mean-field theory

The key point of both models is that the point group and particle-hole symmetries are tied to the flipping of the Ising spin degree of freedom. In the large-h limit, the ground state has the full symmetry of the model Hamiltonian and at the mean-field level, we can set $\langle\hat{s}^z_{\boldsymbol{R}}\rangle=0$. In this limit, the Dirac cones are pinned by symmetry. In the opposite small-h limit, the Ising spins order, i.e. $\langle\hat{s}^z_{\boldsymbol{R}}\rangle\neq 0$. Thereby, the C_{2v} (C_{4v}) symmetry is reduced to $\hat{T}_+$ (C_{2v}). At the mean-field level, this induces a meandering of the Dirac points in the BZ, see Fig. 2.1(a), and an anisotropy in the Fermi velocities. In this section, we present the mean-field calculation. At this level of approximation, the transition turns out to be continuous, in agreement with the large-N analysis [53].

We expand Eq. (2.7) around $\langle\hat{s}^z_{\boldsymbol{R}}\rangle\equiv\phi$. The resulting mean-field Hamiltonian reads:

$$\mathcal{H}_{\text{MF}}=\sum_{\sigma=1}^{N_\sigma}\sum_{\boldsymbol{k}}\hat{K}_{\boldsymbol{k},\sigma}(\phi\xi)+\sum_{\boldsymbol{R}}\hat{I}_{\boldsymbol{R}}(\phi). \tag{2.12}$$

With:

$$\hat{K}_{\boldsymbol{k},\sigma}(\phi\xi)=-\hat{a}^{\dagger}_{\boldsymbol{k},\sigma}\hat{b}_{\boldsymbol{k},\sigma}Z(\boldsymbol{k},\phi\xi)\quad Z(\boldsymbol{k},\phi\xi)=Z_0(\boldsymbol{k})+\phi\xi Z_{\text{Yuk}}(\boldsymbol{k})\quad \hat{I}_{\boldsymbol{R}}(\phi)=-4J\left(\phi\hat{s}^z_{\boldsymbol{R}}-\tfrac{1}{2}\phi^2\right)-h\hat{s}^x_{\boldsymbol{R}}.$$

The fermionic dispersion is

$$\pm\left|Z(\boldsymbol{k},\phi\xi)\right|. \tag{2.13}$$

To determine the nature of the zero-temperature phase transition, we determine the order parameter ϕ for a given transverse field h by minimizing the ground state energy $E_{0,\text{MF}}=\lim_{\beta\to\infty}E_{\text{MF}}=\lim_{\beta\to\infty}\langle\mathcal{H}_{\text{MF}}\rangle_{\text{MF}}$.

$$\begin{aligned}
E_{\text{MF}}=\langle\mathcal{H}_{\text{MF}}\rangle_{\text{MF}}&=\frac{\text{Tr}(\exp(-\beta\mathcal{H}_{\text{MF}})\mathcal{H}_{\text{MF}})}{\text{Tr}(\exp(-\beta\mathcal{H}_{\text{MF}}))}\\
&=N_\sigma\sum_{\boldsymbol{k}}|Z(\boldsymbol{k},\phi\xi)|\frac{1-\exp\left(\beta|Z(\boldsymbol{k},\phi\xi)|\right)}{1+\exp\left(\beta|Z(\boldsymbol{k},\phi\xi)|\right)}+L^2\left(2\phi^2-\sqrt{h^2+16\phi^2}\tanh\left(\beta\sqrt{h^2+16\phi^2}\right)\right)
\end{aligned}$$

$$\begin{aligned}
\lim_{\beta\to\infty}\frac{E_{\mathrm{MF}}}{L^2} &= \underbrace{-\frac{N_\sigma}{L^2}\sum_{\boldsymbol{k}}|Z(\boldsymbol{k},\phi\xi)|}_{\epsilon_{\mathrm{F}}(L,\phi\xi)} + \underbrace{\left(2\phi^2-\sqrt{h^2+16\phi^2}\right)}_{\epsilon_{\mathrm{I}}(\phi,h)} \\
\partial_\phi\epsilon_{\mathrm{F}}(L,\phi) &= -\frac{N_\sigma}{L^2}\sum_{\boldsymbol{k}}\frac{1}{|Z(\boldsymbol{k},\phi)|}\left(\Re\left(Z_0(\boldsymbol{k})\bar{Z}_{\mathrm{Yuk}}(\boldsymbol{k})\right)+\phi|Z_{\mathrm{Yuk}}(\boldsymbol{k})|^2\right) \\
\partial_\phi^2\epsilon_{\mathrm{F}}(L,\phi) &= -\frac{N_\sigma}{L^2}\sum_{\boldsymbol{k}}\left(\frac{|Z_{\mathrm{Yuk}}(\boldsymbol{k})|^2}{|Z(\boldsymbol{k},\phi)|}-\frac{\Re\left(Z_0(\boldsymbol{k})\bar{Z}_{\mathrm{Yuk}}(\boldsymbol{k})\right)}{|Z(\boldsymbol{k},\phi)|^3}\left(\Re\left(Z_0(\boldsymbol{k})\bar{Z}_{\mathrm{Yuk}}(\boldsymbol{k})\right)+\phi|Z_{\mathrm{Yuk}}(\boldsymbol{k})|\right)\right)
\end{aligned} \tag{2.14}$$

Equation (2.14) separates into a fermionic and an Ising part, ϵ_{F} and ϵ_{I}. While ϵ_{I} has a well-behaved, closed form, ϵ_{F} has some non-analytic points for finite lattices (cf. Fig. 2.4). Namely, $\partial_\phi^2\epsilon_{\mathrm{F}}$ diverges, if $Z(\boldsymbol{k},\phi\xi)$ vanishes.

This corresponds to a finite-size artifact which can be qualitatively understood with the help of Fig. 2.3. Essentially, ϕ shifts the single particle energy and produces level crossing reminiscent of those produced when twisting boundary conditions [58, 59]. Fig. 2.3 shows the valence band of a one-dimensional Dirac cone on a lattice with five $\boldsymbol{k}$ points at two different twists. As a function of the twist the $\boldsymbol{k}$-point will cross the Fermi surface and at this crossing point, a singularity in the kinetic energy – corresponding to a level crossing – will appear. This is explicitly shown in Fig. 2.5. This observation means that the thermodynamic limit and the derivative ∂_ϕ do not commute: one should first take the thermodynamic limit prior to carrying out the derivative.

To avoid this artifact, we consider two different approaches:

1. Choose a weaker coupling ξ, such that the Dirac points do not cross the Fermi surface in proximity to the critical point (see Fig. 2.5 vs Fig. 2.6). However, choosing a small ξ may result in a slow flow from the 3d Ising fixed point of the unperturbed Ising model to nematic criticality.
2. Choose antiperiodic boundary conditions in space for the fermions, so as to shift the $\boldsymbol{k}$ points away from the Fermi surface: Fig. 2.7. However, this choice results in large finite size effects presumably due to the boundary-condition-induced finite size gap.

It turns out the first option is the best choice and that ξ can be chosen large enough so as to minimize the aforementioned crossover effects.

The C_{4v} model (Fig. 2.8) shows different behaviors between systems with linear sizes $L = 2 + 4\mathbb{N}$ and $L = 4\mathbb{N}$. At $L = 4\mathbb{N}$ with periodic boundary conditions, the Dirac points in the disordered phase are located at $\boldsymbol{k}$-points resolved by the finite lattice. This is not the case at $L = 2+4\mathbb{N}$ (cf. Fig. 2.9). As a result, the $L = 4\mathbb{N}$ sizes have a smoothed-out phase transition. Nevertheless, both $L = 4\mathbb{N}$ and $L = 2+4\mathbb{N}$ converge to the same result in the thermodynamic limit. The Monte-Carlo simulations also have odd-even effects, as elaborated in Section 2.8.3. It turns out that even system sizes produce nicer numerical results for the phase transition.

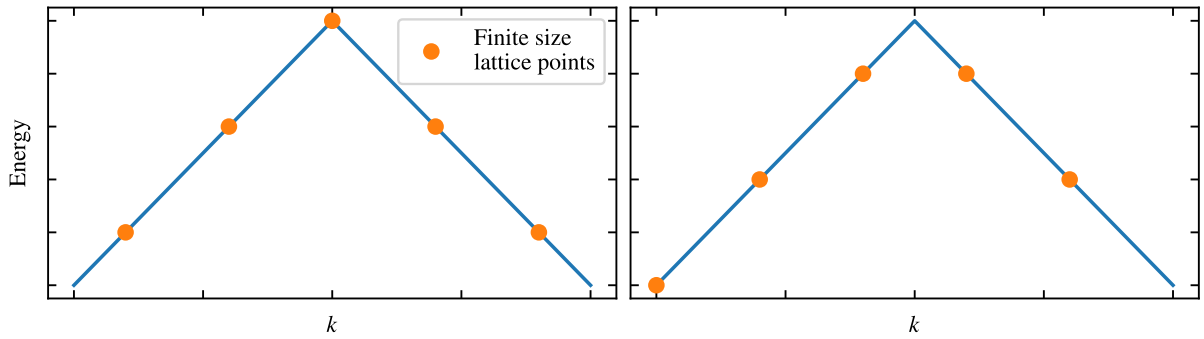


Fig. 2.3: Sketch for understanding finite-size artifacts of Dirac systems as a function of $\boldsymbol{k}$-quantization. Shown is the valence band of a one-dimensional Dirac cone on a lattice of size $L = 5$. We can see that the choice of the momenta quantization (i.e. boundary conditions) affects the energy. Left: Dirac point belongs to the set of finite-size $\boldsymbol{k}$-points. Right: Dirac point is between two finite-size $\boldsymbol{k}$-points. In the quantizations shown on the left, the ground state energy is higher, and a level crossing occurs when a $\boldsymbol{k}$ point crosses the Fermi surface. In nematic transitions, the translation symmetry is not broken such that momenta are well defined and the $\boldsymbol{k}$ quantization for a given lattice size remains unchanged. However, the position of the Dirac point meanders. The energy level crossing, that originates, is reminiscent of that obtained when twisting boundary conditions [58, 59].

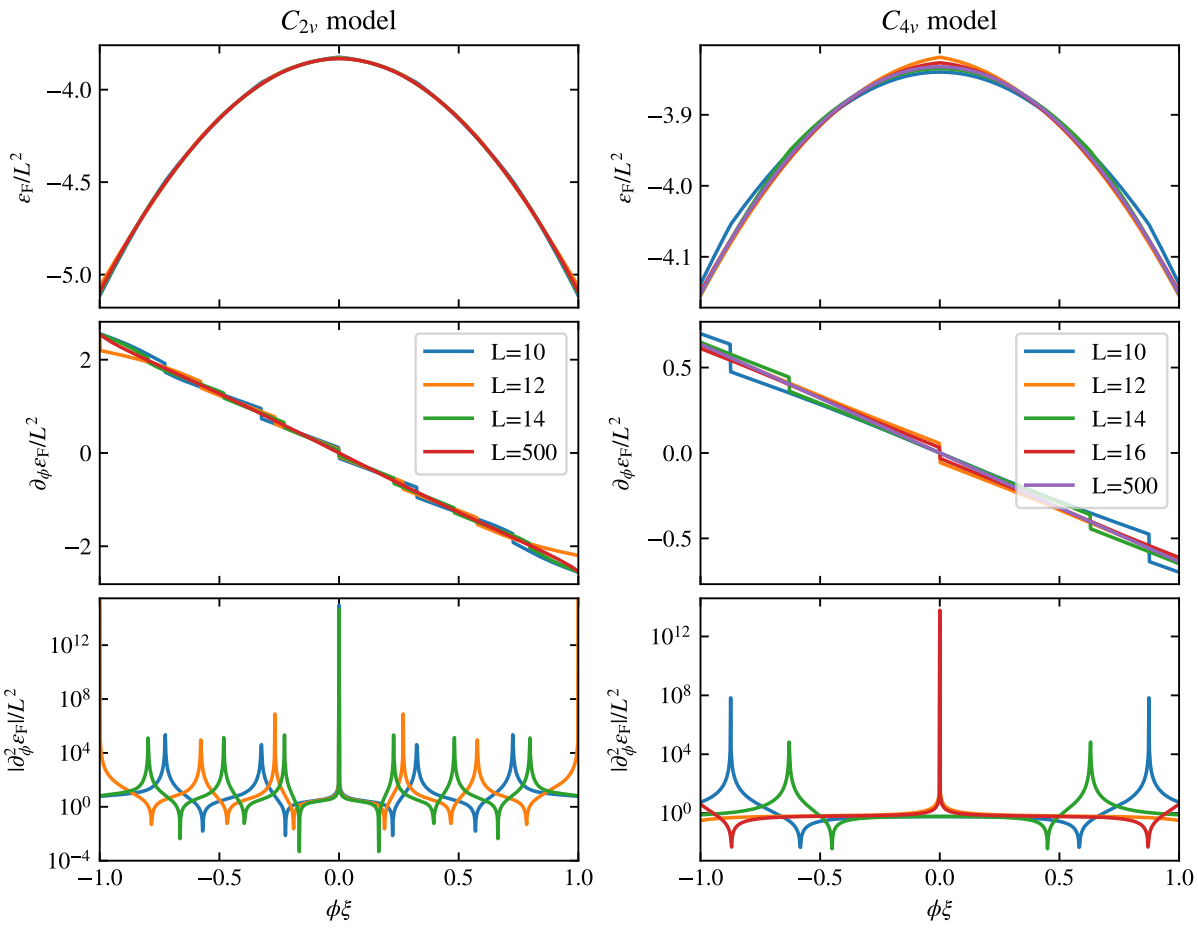


Fig. 2.4: Fermionic part of Mean-field ground state energy. The second derivative with respect to the Ising field ϕ diverges at level crossings as described in the main text.

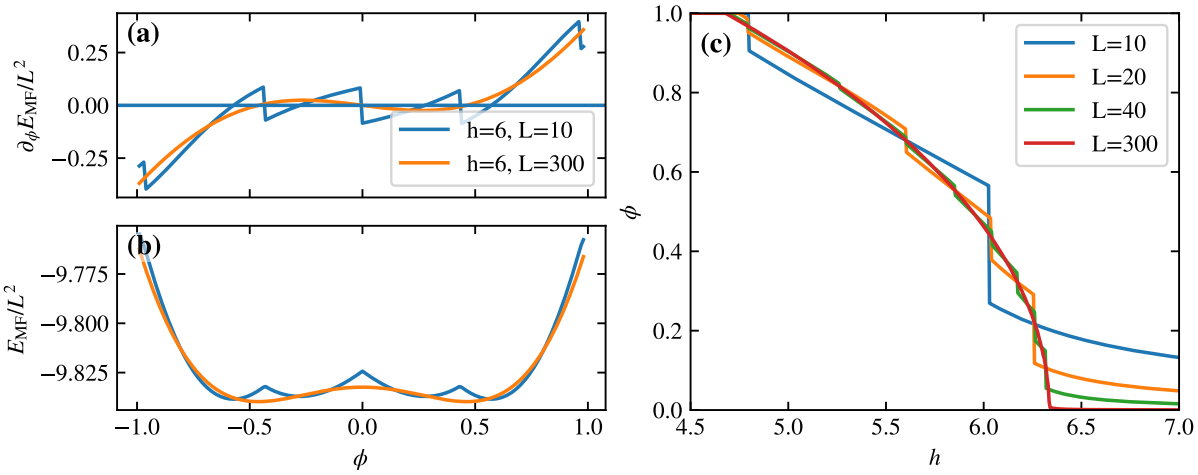


Fig. 2.5: Mean-field results for the C_{2v} model with coupling strength $\xi = 0.75$. There are many discontinuities in the order parameter ϕ when tuning the transverse field h trough the phase transition. These finite size artifacts are disappearing at $L \to \infty$.

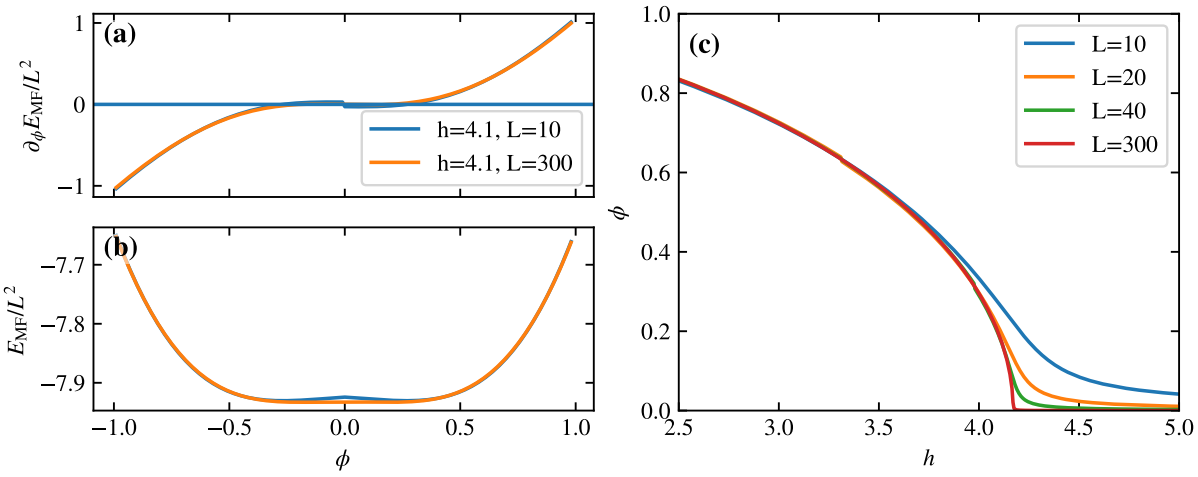


Fig. 2.6: Mean-field results for the C_{2v} model with coupling strength $\xi = 0.25$. Most discontinuities visible for $\xi = 0.75$ (cf. Fig. 2.5) are avoided at this coupling strength.

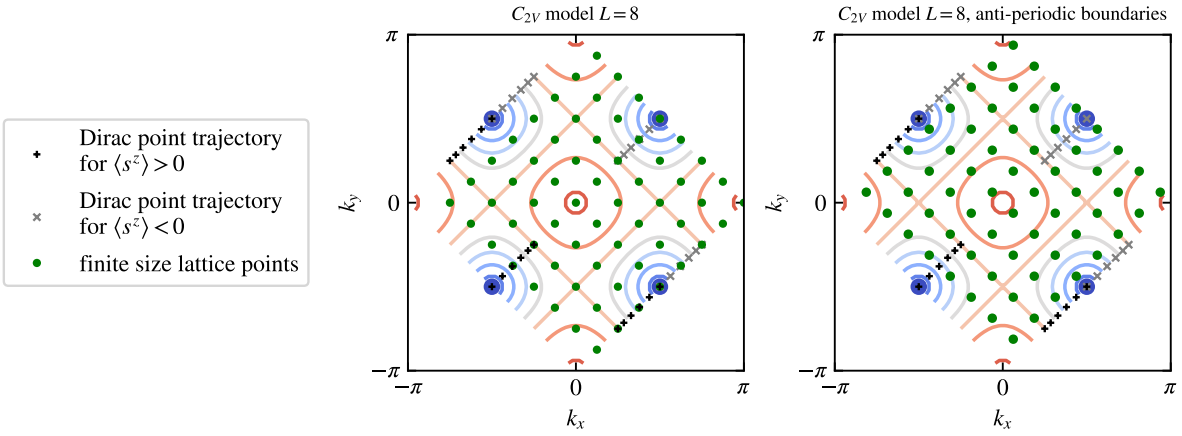


Fig. 2.7: Brillouin zone of C_{2v} model with $\boldsymbol{k}$ points of a $8*8$ lattice. Left: With periodic boundary conditions. Right: With antiperiodic boundary conditions for movement parallel to $(1,-1)$. Also sketched: Dispersion in disordered phase and trajectory of Dirac cones.

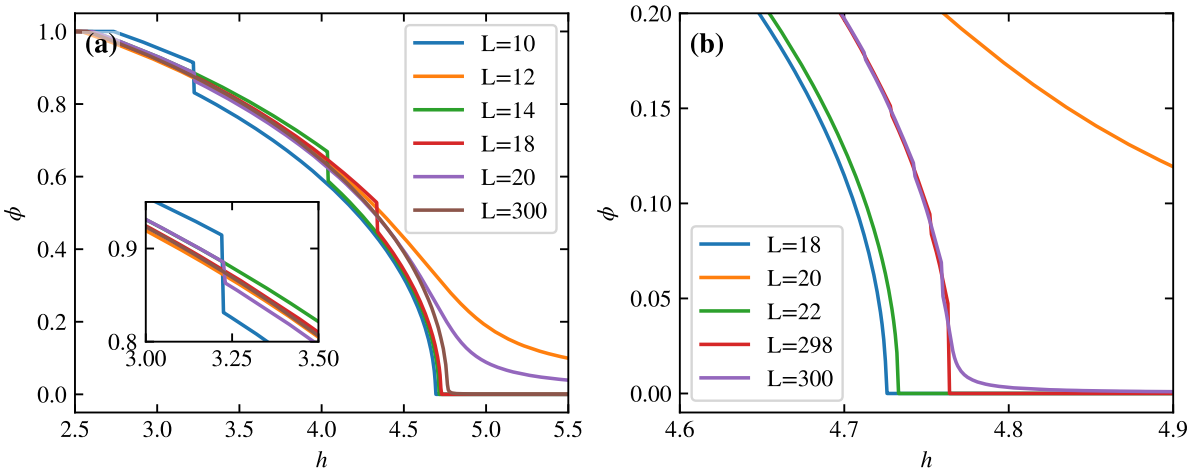


Fig. 2.8: Mean-field results for the C_{4v} model with coupling strength $\xi = 1$. There is an qualitative difference between system sizes $L = 2 + 4\mathbb{N}$ and $L = 4\mathbb{N}$: The former converges faster, but both are still converging to the same result for $L \to \infty$.

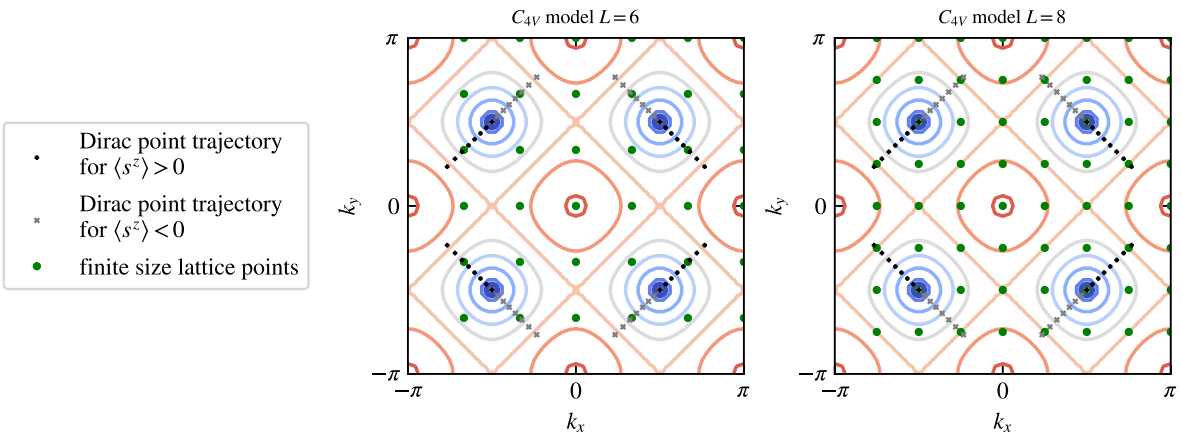


Fig. 2.9: Brillouin zone of C_{4v} model with $\boldsymbol{k}$ points of $6*6$ and $8*8$ lattice. Also sketched: Dispersion in disordered phase and trajectory of Dirac cones. Left: $6*6$ lattice, the Dirac cones in the disordered phase are each centered between four $\boldsymbol{k}$ points. Right: $8*8$ lattice, the Dirac cones in the disordered phase are directly resolved by the $\boldsymbol{k}$ points.

To summarize, the mean-field calculation finds a continuous transitions for both models and to avoid finite size artifacts in the QMC simulations, ξ should not be chosen too large.

2.4 Continuum field theory

In order to investigate whether the above remains true upon the inclusion of order-parameter fluctuations, we derive corresponding continuum field theories, which are amenable to RG analyses.

We derive the low energy model from Eq. (2.7) by expansion in $\boldsymbol{k}$ around the nodal points $\boldsymbol{K}_i$ and for a scalar Ising field $\phi(\boldsymbol{q})$:

$$\begin{aligned}\mathcal{H} = -\sum_{\sigma=1}^{N_\sigma}\sum_i \int \mathrm{d}\boldsymbol{\kappa}\, \hat{a}^\dagger_{\boldsymbol{K}_i+\boldsymbol{\kappa},\sigma}\Bigg(\hat{b}_{\boldsymbol{K}_i+\boldsymbol{\kappa},\sigma} Z_0(\boldsymbol{K}_i+\boldsymbol{\kappa}) \\ + \frac{1}{2\pi}\int \mathrm{d}\boldsymbol{q}\, \hat{b}_{\boldsymbol{K}_i+\boldsymbol{\kappa}-\boldsymbol{q},\sigma}\phi(\boldsymbol{q}) Z_{\mathrm{Yuk}}(\boldsymbol{K}_i+\boldsymbol{\kappa})\Bigg) + \mathrm{H.c.} + H_{\mathrm{Ising}}\{\phi\}\end{aligned}$$

In leading order in $\boldsymbol{\kappa}$ we obtain:

$$\begin{aligned}\mathcal{H} = -\sum_{\sigma=1}^{N_\sigma}\sum_i \int \mathrm{d}\boldsymbol{\kappa}\, \hat{a}^\dagger_{\boldsymbol{K}_i+\boldsymbol{\kappa},\sigma}\Bigg(\hat{b}_{\boldsymbol{K}_i+\boldsymbol{\kappa},\sigma}\boldsymbol{\kappa}\nabla Z_0(\boldsymbol{K}_i) \\ + \frac{1}{2\pi}\int \mathrm{d}\boldsymbol{q}\, \hat{b}_{\boldsymbol{K}_i+\boldsymbol{\kappa}-\boldsymbol{q},\sigma}\phi(\boldsymbol{q}) Z_{\mathrm{Yuk}}(\boldsymbol{K}_i)\Bigg) + \mathrm{H.c.} + H_{\mathrm{Ising}}\{\phi\}.\end{aligned}$$

Introducing the Fourier transformations:

$$\begin{pmatrix}\hat{a}^i_\sigma(\boldsymbol{r}) \\ \hat{b}^i_\sigma(\boldsymbol{r})\end{pmatrix} = \frac{1}{2\pi}\int \mathrm{d}\boldsymbol{\kappa}\, e^{i\boldsymbol{\kappa}\boldsymbol{r}}\begin{pmatrix}\hat{a}_{\boldsymbol{K}_i+\boldsymbol{\kappa},\sigma} \\ \hat{b}_{\boldsymbol{K}_i+\boldsymbol{\kappa},\sigma}\end{pmatrix}$$
$$\phi(\boldsymbol{r}) = \frac{1}{2\pi}\int \mathrm{d}\boldsymbol{\kappa}\, e^{i\boldsymbol{q}\boldsymbol{r}}\phi(\boldsymbol{q}).$$

and defining:

$$\boldsymbol{v}_i \equiv \nabla Z_0(\boldsymbol{K}_i) \qquad\qquad I_i \equiv Z_{\mathrm{Yuk}}(\boldsymbol{K}_i)$$

The Hamiltonian takes the form:

$$\mathcal{H} = -\sum_{\sigma=1}^{N_\sigma}\sum_i \int \mathrm{d}\boldsymbol{r}\, \hat{a}^i_\sigma(\boldsymbol{r})^\dagger\Big(i\boldsymbol{v}_i\cdot\nabla_{\boldsymbol{r}} + \phi(\boldsymbol{r})I_i\Big)\hat{b}^i_\sigma(\boldsymbol{r}) + \mathrm{H.c.} + H_{\mathrm{Ising}}\{\phi\}. \tag{2.15}$$

2.4.1 The C_{2v} model

The C_{2v} model has the nodal points $\boldsymbol{K}_{\pm} = \begin{pmatrix} \pi/2 \\ \pm\pi/2 \end{pmatrix}$. By defining the four-component Dirac spinor

$$\Psi_\sigma(\boldsymbol{r}) = \begin{pmatrix} \hat{a}_\sigma^+(\boldsymbol{r}) & \hat{b}_\sigma^+(\boldsymbol{r}) & \hat{a}_\sigma^-(\boldsymbol{r}) & \hat{b}_\sigma^-(\boldsymbol{r}) \end{pmatrix}^{\mathrm{T}},$$

where $\hat{a}_\sigma^\pm$ and $\hat{b}_\sigma^\pm$ corresponds to hole excitations near $\boldsymbol{K}_\pm$ on the A and B sublattices, respectively, and

$$\tau_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \tau_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \tau_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Eq. (2.15) can be written as

$$\begin{aligned} \mathcal{H}^{C_{2v}} = \sum_{\sigma=1}^{N_\sigma} \int \mathrm{d}\boldsymbol{r}\ \Psi_\sigma^\dagger(\boldsymbol{r}) \Bigg[& 2it \begin{pmatrix} \tau_2 & 0 \\ 0 & -\tau_1 \end{pmatrix} \partial_{r_+} + 2it \begin{pmatrix} \tau_1 & 0 \\ 0 & -\tau_2 \end{pmatrix} \partial_{r_-} + 2\sqrt{2}\xi\phi(\boldsymbol{r}) \begin{pmatrix} -\tau_2 & 0 \\ 0 & -\tau_1 \end{pmatrix} \Bigg] \Psi_\sigma(\boldsymbol{r}) \\ & + H_{\text{Ising}}\{\phi\}. \end{aligned}$$

Introducing the gamma matrices, that realize a four-dimensional representation of the Clifford algebra

$$\gamma_0 = \begin{pmatrix} -\tau_3 & 0 \\ 0 & -\tau_3 \end{pmatrix} \qquad \gamma_1 = \begin{pmatrix} \tau_1 & 0 \\ 0 & \tau_2 \end{pmatrix} \qquad \gamma_2 = \begin{pmatrix} -\tau_2 & 0 \\ 0 & -\tau_1 \end{pmatrix} \qquad \{\gamma_\alpha, \gamma_\beta\} = 2\delta_{\alpha\beta},$$

we can write the action in the form

$$S^{C_{2v}} = \int \mathrm{d}^D x \sum_{\sigma=1}^{N_\sigma} \Big[\Psi_\sigma^\dagger(x) [\mathbb{1}\partial_\tau + v\gamma_0\gamma_1\partial_+ + v\gamma_0\gamma_2\partial_- + g\phi(x)\gamma_2] \Psi_\sigma(x) \Big] + S_{\text{Ising}}(\{\phi\}). \tag{2.16}$$

2.4.2 The C_{4v} model

The C_{4v} model has the nodal points $\boldsymbol{K}_{+\pm} = \begin{pmatrix} \pi/2 \\ \pm\pi/2 \end{pmatrix}$ and $\boldsymbol{K}_{-\pm} = -\boldsymbol{K}_\pm$. By defining the eight-component Dirac spinor

$$\Psi_\sigma(\boldsymbol{r}) = \begin{pmatrix} \hat{a}_\sigma^{++}(\boldsymbol{r}) & \hat{b}_\sigma^{++}(\boldsymbol{r}) & \hat{a}_\sigma^{-+}(\boldsymbol{r}) & \hat{b}_\sigma^{-+}(\boldsymbol{r}) & \hat{a}_\sigma^{+-}(\boldsymbol{r}) & \hat{b}_\sigma^{+-}(\boldsymbol{r}) & \hat{a}_\sigma^{--}(\boldsymbol{r}) & \hat{b}_\sigma^{--}(\boldsymbol{r}) \end{pmatrix}^{\mathrm{T}},$$

where $\hat{a}_\sigma^{+\pm}$ and $\hat{b}_\sigma^{+\pm}$ ($\hat{a}_\sigma^{-\pm}$ and $\hat{b}_\sigma^{-\pm}$) correspond to hole excitations near $\boldsymbol{K}_\pm$ ($-\boldsymbol{K}_\pm$). Eq. (2.15) can be written as

$$\begin{aligned} \mathcal{H}^{C_{4v}} = \sum_{\sigma=1}^{N_\sigma} \int \mathrm{d}\boldsymbol{r}\ \Psi_\sigma^\dagger(\boldsymbol{r}) \Bigg[& \\ & 2it \Bigg(\begin{pmatrix} \tau_1 & 0 \\ 0 & -\tau_1 \end{pmatrix} \oplus \begin{pmatrix} -\tau_2 & 0 \\ 0 & \tau_2 \end{pmatrix} \Bigg) \partial_{r_+} + 2it \Bigg(\begin{pmatrix} -\tau_2 & 0 \\ 0 & \tau_2 \end{pmatrix} \oplus \begin{pmatrix} \tau_1 & 0 \\ 0 & -\tau_1 \end{pmatrix} \Bigg) \partial_{r_-} \\ & + 2\sqrt{2}\xi\phi(\boldsymbol{r}) \Bigg(\begin{pmatrix} \tau_2 & 0 \\ 0 & \tau_2 \end{pmatrix} \oplus \begin{pmatrix} \tau_2 & 0 \\ 0 & \tau_2 \end{pmatrix} \Bigg) \\ & \Bigg] \Psi_\sigma(\boldsymbol{r}) + H_{\text{Ising}}\{\phi\}, \end{aligned}$$

where $\oplus$ denotes the matrix direct sum. Introducing another set of gamma matrices, that also realize a four-dimensional representation of the Clifford algebra

$$\tilde{\gamma}_0 = \begin{pmatrix} \tau_3 & 0 \\ 0 & -\tau_3 \end{pmatrix} \qquad \tilde{\gamma}_1 = \begin{pmatrix} \tau_1 & 0 \\ 0 & \tau_1 \end{pmatrix} \qquad \tilde{\gamma}_2 = \begin{pmatrix} \tau_2 & 0 \\ 0 & \tau_2 \end{pmatrix} \qquad \{\tilde{\gamma}_\alpha, \tilde{\gamma}_\beta\} = 2\delta_{\alpha\beta},$$

we can write the action in the compact form

$$\begin{aligned} S^{C_{4v}} = \int \mathrm{d}^D x \sum_{\sigma=1}^{N_\sigma} \Psi_\sigma^\dagger(x) \Big[& \mathbb{1}\partial_\tau + v\left(\tilde{\gamma}_0\tilde{\gamma}_1 \oplus \tilde{\gamma}_0\tilde{\gamma}_2\right)\partial_+ + v\left(\tilde{\gamma}_0\tilde{\gamma}_2 \oplus \tilde{\gamma}_0\tilde{\gamma}_1\right)\partial_- + g\phi(x)\left(\tilde{\gamma}_2 \oplus \tilde{\gamma}_2\right) \Big] \Psi_\sigma(x) \\ & + S_{\text{Ising}}(\{\phi\}). \end{aligned} \tag{2.17}$$

2.4.3 Finalized field theory

So, all in all, to leading order in the gradient expansion around the nodal points, we obtain the Euclidean action $S = \int \mathrm{d}^2x\mathrm{d}\tau(\mathcal{L}_\Psi + \mathcal{L}_\phi)$ with

$$\mathcal{L}_\Psi^{C_{2v}} = \Psi_\sigma^\dagger \left(\partial_\tau + \gamma_0\gamma_1 v_\parallel \partial_+ + \gamma_0\gamma_2 v_\perp \partial_- + g\phi\gamma_2\right)\Psi_\sigma. \tag{2.18}$$

and

$$\mathcal{L}_\Psi^{C_{4v}} = \Psi_\sigma^\dagger[\partial_\tau + \tilde{\gamma}_0(\tilde{\gamma}_1 v_\parallel \oplus \tilde{\gamma}_2 v_\perp)\partial_+ + \tilde{\gamma}_0(\tilde{\gamma}_2 v_\perp \oplus \tilde{\gamma}_1 v_\parallel)\partial_- + g\phi(\tilde{\gamma}_2 \oplus \tilde{\gamma}_2)]\Psi_\sigma \tag{2.19}$$

In the above Lagrangians, we have assumed the summation convention over repeated indices. Furthermore, we allow for anisotropic Fermi velocities $v_\parallel$ and $v_\perp$, corresponding to the directions parallel and perpendicular to the shift of the Dirac cones in the ordered phase, with $v_\parallel = v_\perp \sim t$ at the UV cutoff scale Λ. $\partial_\pm$ denotes the spatial derivative in the direction along $\boldsymbol{K}_\pm$. The fermions couple via $g \sim \xi$ to the Ising order-parameter field ϕ. The dynamics of the Ising field is governed by the usual ϕ^4 Lagrangian,

$$\mathcal{L}_\phi = \frac{1}{2}\phi(r - \partial_\tau^2 - c_+^2\partial_+^2 - c_-^2\partial_-^2)\phi + \lambda\phi^4,$$

with the tuning parameter r, the boson velocities $c_\pm$, and the bosonic self-interaction λ.

2.5 ϵ expansion

The presence of a unique upper critical spatial dimension of three allows an $\epsilon = 3 - d$ expansion, with $\epsilon = 1$ corresponding to the physical case. Because of the lack of Lorentz and continuous spatial rotational symmetries in the low-energy models, it is useful to employ a regularization in the frequency only, which allows us to rescale the different momentum components independently, and evaluate the loop integrals analytically (See *Appendix A.1* for details). Two central properties of nematic quantum phase transitions in Dirac systems are revealed by the one-loop RG analysis: First, both models admit a stable fixed point featuring anisotropic power laws of the fermion and order parameter correlation functions.

In the C_{2v} model, both components of the Fermi velocity remain finite at the stable fixed point with $0 < v_\parallel^* < v_\perp^*$. At the critical point, a unique timescale τ emerges for both fields Ψ and ϕ [60, 61], which scales with the two characteristic length scales ℓ_+ and ℓ_- as $\tau \sim \ell_+^{z_+} \sim \ell_-^{z_-}$, with associated dynamical critical exponents $z_\pm = [1 - \frac{1}{2}\eta_\phi + \frac{1}{2}\eta_\pm]^{-1}$ as $(z_+, z_-) = (1 + 0.3695\epsilon, 1 + 0.1086\epsilon) + \mathcal{O}(\epsilon^2)$, reflecting the absence of Lorentz and rotational symmetries at criticality.

By contrast, in the C_{4v} model, the fixed point is characterized by a maximal velocity anisotropy with $(v_\parallel^*, v_\perp^*) = (0, 1)$ in units of fixed boson velocities $c \equiv c_+ = c_- = 1$. This result is consistent with the large-N RG analysis in fixed $d = 2$ [52]. The fact that $v_\parallel^*$ vanishes leads to the interesting behavior that the fixed-point couplings g_*^2 and λ_* are bound to vanish in this case as well. This happens in a way that the ratio $(g^2/v_\parallel)_*$ remains finite, such that the boson anomalous dimensions become $\eta_\phi = \eta_+ = \eta_- = \epsilon$. Importantly, as the fixed-point couplings g_*^2 and λ^* vanish, we expect the one-loop result for the critical exponents to hold at *all* loop orders in the C_{4v} model. For the correlation-length exponent, we find $1/\nu = 2 - \epsilon$. The remaining exponents can then be computed by assuming the usual hyperscaling relations [62]. The susceptibility exponent, for instance, becomes $\gamma = 1$, independent of ϵ. This result is again consistent with the large-N calculation and has previously already been argued to hold exactly [52].

We note that the values of the exponents in the C_{4v} model are independent of the number of spinor components, in contrast to the situation in the C_{2v} model, as well as to the usual Gross-Neveu universality classes [43, 44, 45, 46, 57, 63]. The unique dynamical critical exponent in the C_{4v} model becomes $z = 1$. We emphasize, however, that the critical point still does *not* feature emergent Lorentz symmetry [64] due to the anisotropic fermion spectral function. The second important property revealed by the RG analysis is that the stable fixed points in both models are approached only extremely slowly as function of RG scale, Fig. 2.10. This is universally true for the C_{4v} model, in which case $v_\parallel$ corresponds to a marginally irrelevant parameter, hence scaling only logarithmically to zero while other irrelevant operators rapidly die out. This defines a quasiuniversal flow [65, 66] in which only the velocity anisotropy and not the initial ultraviolet values of other parameters determine the slow drift of the exponents. The RG suggests that this regime emerges at scales $1/b \lesssim 0.05$ (cf. *Appendix A.1*), such that it will dominate numerical as well as experimental realizations of this critical phenomena. For a reasonable set of ultraviolet starting values and $\epsilon = 1$,

we find that the effective correlation-length exponent $1/\nu_{\text{eff}}$ (anomalous dimension η_ϕ^{eff}) approaches one from above (below), with sizable deviations at intermediate RG scales. Moreover, we also observe that the initial flows at high energy in the two models resemble each other, despite the fact that they substantially deviate from each other at low energy. This suggests that the flow is generically slow in the C_{2v} model as well.

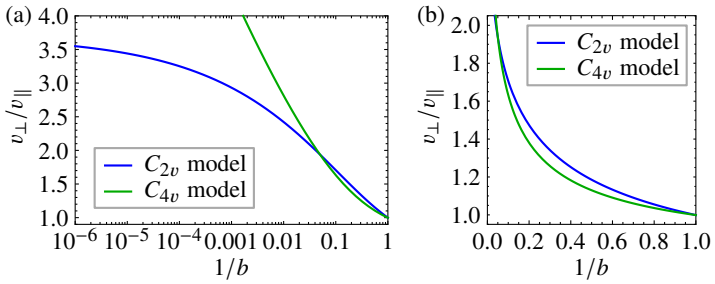


Fig. 2.10: Ratio of Fermi velocities $v_\perp/v_\parallel$ as function of RG scale $1/b$ for both models. We assume ultraviolet initial values of $v_\parallel(b=1) = v_\perp(b=1) = 0.25$, and set $g^2/(v_\parallel v_\perp)(b=1)$ to the value at the respective stable fixed point. (a) Semilogarithmic, (b) linear plots. Starting at a temperature scale representative of the ultraviolet initial parameters, one has to cool the system by 2 orders of magnitude to *start* observing the differences between both models.

2.6 QMC setup

For the numerical simulations, we used the ALF program package [11, 12] that provides a general implementation of the finite-temperature auxiliary field QMC algorithm [8, 9, 10]. To formulate the path integral, we use a Trotter decomposition with time step $\Delta_\tau t = 0.1$ and choose a basis where $\hat{s}^z_{\boldsymbol{R}}|s_{\boldsymbol{R}}\rangle = s_{\boldsymbol{R}}|s_{\boldsymbol{R}}\rangle$. The configuration space is that of a $(2+1)$-dimensional Ising model and we use a single-spin-flip update to sample it. As we will show in Section 2.6.1 both models are negative-sign-problem free for all values of N_σ [67]. For our simulations, we have used an inverse temperature $\beta = 4L$ for $L \times L$ lattices, and have checked that this choice of β reflects ground-state properties. For the results shown in the main text, we have fixed the parameters as $J = t = 1$ and $N_\sigma = 2$. In the C_{2v} model, we choose $\xi = 0.25$, since larger values of ξ lead to spurious size effects that could falsely be interpreted as first-order transitions, as discussed in Section 2.3, see also Ref. [57] for a detailed discussion. In the C_{4v} model, we set $\xi = 1$. As shown in *Appendix A.5*, other values of ξ and N_σ do not alter the continuous nature of the transition.

2.6.1 Absence of negative sign problem

Here we use the Majorana representation to demonstrate, using the results of Ref. [67], the absence of negative sign problem for all values of N_σ. Both models have SU(N_σ) symmetry. Since the Ising spins couple symmetrically to the fermion spins, SU(N_σ) symmetry is present for all Ising spin configurations. Thereby, the fermion determinant of the SU(N_σ) model corresponds to that of the U(1) model ($N_\sigma = 1$) elevated to the power N_σ. It hence suffices to demonstrate the absence of negative sign problem at $N_\sigma = 1$. In this section, we will hence omit the spin index. Additionally we include a chemical potential term $\mathcal{H}_\mu$, to show that there is also no sign problem under doping for even values of N_σ.

2.6.1.1 The C_{2v} model

Consider the canonical transformation,

$$\begin{pmatrix} \hat{a}_{\boldsymbol{k}} \\ \hat{b}_{\boldsymbol{k}} \end{pmatrix} \to \begin{pmatrix} 1 & 0 \\ 0 & e^{-i3\pi/4} \end{pmatrix} \begin{pmatrix} \hat{a}_{\boldsymbol{k}-\Delta\boldsymbol{K}} \\ \hat{b}_{\boldsymbol{k}-\Delta\boldsymbol{K}} \end{pmatrix}$$

with

$$\hat{b}_{\boldsymbol{R}} = \frac{1}{\sqrt{N}} \sum_{\boldsymbol{k}\in BZ} e^{i\boldsymbol{k}\cdot\boldsymbol{R}} \hat{b}_{\boldsymbol{k}}$$

and $\Delta \boldsymbol{K} = \frac{1}{4}(\boldsymbol{b}_- - \boldsymbol{b}_+)$. Here, $\boldsymbol{e}_i \cdot \boldsymbol{b}_j = 2\pi\delta_{i,j}$. This canonical transformation renders the Hamiltonian real: the π-flux, is realized by changing the sign of the intra unit-cell hopping with respect to the other hoppings. More precisely after the transformation, the fermionic part of the Hamiltonian takes the form:

$$\begin{aligned}
\mathcal{H}_0^{C_{2v}} &= -t \sum_{\boldsymbol{R}} \hat{a}^\dagger_{\boldsymbol{R}} \Big(-\hat{b}_{\boldsymbol{R}} + \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_-} + \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_- - \boldsymbol{e}_+} + \hat{b}_{\boldsymbol{R}-\boldsymbol{e}_+} \Big) + \text{H.c.} \\
\mathcal{H}_{\text{Yuk}}^{C_{2v}} &= -\xi \sum_{\boldsymbol{R}} s_{\boldsymbol{R}} \hat{a}^\dagger_{\boldsymbol{R}} \Big(-\hat{b}_{\boldsymbol{R}} - \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_-} - \hat{b}_{\boldsymbol{R}+\boldsymbol{e}_- - \boldsymbol{e}_+} + \hat{b}_{\boldsymbol{R}-\boldsymbol{e}_+} \Big) + \text{H.c.} \\
\mathcal{H}_\mu^{C_{2v}} &= \mu \sum_{\boldsymbol{R}} \left(\hat{a}^\dagger_{\boldsymbol{R}} \hat{a}_{\boldsymbol{R}} + \hat{b}^\dagger_{\boldsymbol{R}} \hat{b}_{\boldsymbol{R}} \right)
\end{aligned} \tag{2.20}$$

In the above, we have considered an arbitrary set of Ising spins $s_{\boldsymbol{R}} = \pm 1$.

Since equation (2.20) is real, the corresponding fermion determinant for $N_\sigma = 1$ is also real and therefore positive for even N_σ.

For the sign to remain positive with odd N_σ, we have to dismiss H_μ and introduce Majorana fermions:

$$\hat{a}^\dagger_{\boldsymbol{R}} = \tfrac{1}{2}\left(\hat{\gamma}_{\boldsymbol{R},1} - i\hat{\gamma}_{\boldsymbol{R},2}\right) \qquad\qquad \hat{b}^\dagger_{\boldsymbol{R}} = \tfrac{1}{2}\left(i\hat{\eta}_{\boldsymbol{R},1} + \hat{\eta}_{\boldsymbol{R},2}\right).$$

In the Majorana basis, the Fermionic part of the Hamiltonian reads:

$$\begin{aligned}
\mathcal{H}_0^{C_{2v}} &= \frac{it}{2} \sum_{\boldsymbol{R}} \hat{\boldsymbol{\gamma}}^{\mathrm{T}}_{\boldsymbol{R}} \Big(-\hat{\boldsymbol{\eta}}_{\boldsymbol{R}} + \hat{\boldsymbol{\eta}}_{\boldsymbol{R}+\boldsymbol{e}_-} + \hat{\boldsymbol{\eta}}_{\boldsymbol{R}+\boldsymbol{e}_- - \boldsymbol{e}_+} + \hat{\boldsymbol{\eta}}_{\boldsymbol{R}-\boldsymbol{e}_+} \Big) \\
\mathcal{H}_{\text{Yuk}}^{C_{2v}} &= \frac{i\xi}{2} \sum_{\boldsymbol{R}} s_{\boldsymbol{R}} \hat{\boldsymbol{\gamma}}^{\mathrm{T}}_{\boldsymbol{R}} \Big(-\hat{\boldsymbol{\eta}}_{\boldsymbol{R}} - \hat{\boldsymbol{\eta}}_{\boldsymbol{R}+\boldsymbol{e}_-} - \hat{\boldsymbol{\eta}}_{\boldsymbol{R}+\boldsymbol{e}_- - \boldsymbol{e}_+} + \hat{\boldsymbol{\eta}}_{\boldsymbol{R}-\boldsymbol{e}_+} \Big)
\end{aligned}$$

In the above, $\hat{\boldsymbol{\gamma}}^T_{\boldsymbol{R}} = (\hat{\gamma}_{\boldsymbol{R},1}, \hat{\gamma}_{\boldsymbol{R},2})$ and a similar form holds for $\hat{\boldsymbol{\eta}}_{\boldsymbol{R}}$. The fact that the Hamiltonian is diagonal in the Majorana index shows that it has a higher $\mathrm{O}(2N_\sigma)$ as opposed to the apparent $\mathrm{SU}(N_\sigma)$ one in the fermion representation. It also has as consequence that, for the $N_\sigma = 1$ case, the fermion determinant is nothing but the square of a Pfaffian that takes real values. Hence the negative sign problem is absent [68, 69].

We close this subsection by making contact with the work of Ref. [67], showing the absence of the sign problem with a alternative approach. Let $\boldsymbol{\mu}$ be a vector of Pauli matrices acting on the Majorana index. Adopting the notation of Ref. [67], we can define:

$$\begin{aligned}
\hat{T}_1^- \alpha \hat{\boldsymbol{\gamma}}_{\boldsymbol{R}} \left(\hat{T}_1^-\right)^{-1} &= \bar{\alpha} i \boldsymbol{\mu}_y \hat{\boldsymbol{\gamma}}_{\boldsymbol{R}} \\
\hat{T}_1^- \alpha \hat{\boldsymbol{\eta}}_{\boldsymbol{R}} \left(\hat{T}_1^-\right)^{-1} &= -\bar{\alpha} i \boldsymbol{\mu}_y \hat{\boldsymbol{\eta}}_{\boldsymbol{R}}
\end{aligned}$$

and

$$\begin{aligned}
\hat{T}_2^+ \alpha \hat{\boldsymbol{\gamma}}_{\boldsymbol{R}} \left(\hat{T}_2^+\right)^{-1} &= \bar{\alpha} \boldsymbol{\mu}_x \hat{\boldsymbol{\gamma}}_{\boldsymbol{R}} \\
\hat{T}_2^+ \alpha \hat{\boldsymbol{\eta}}_{\boldsymbol{R}} \left(\hat{T}_2^+\right)^{-1} &= -\bar{\alpha} \boldsymbol{\mu}_x \hat{\boldsymbol{\eta}}_{\boldsymbol{R}}.
\end{aligned}$$

Since $\left[\hat{T}_2^+, \mathcal{H}^{C_{2v}}\right] = \left[\hat{T}_1^-, \mathcal{H}^{C_{2v}}\right] = 0$ and $\hat{T}_1^-$ and $\hat{T}_2^+$ anti-commute, our Hamiltonian belongs to the so-called Majorana class, and is hence free of the negative sign problem.

2.6.1.2 The C_{4v} model

Consider the spinor $\hat{\boldsymbol{c}}^\dagger_{\boldsymbol{R}} = \left(\hat{a}^\dagger_{\boldsymbol{R}}, \hat{b}^\dagger_{\boldsymbol{R}}\right)$. With this notation, the fermionic part of the C_{4v} model takes the form:

$$\begin{aligned}
\mathcal{H}_0^{C_{4v}} &= -t \sum_{\boldsymbol{R} \in A, \delta = \pm} \left(\hat{\boldsymbol{c}}^\dagger_{\boldsymbol{R}}\, \boldsymbol{e}_+ \cdot \boldsymbol{\tau}\, \hat{\boldsymbol{c}}_{\boldsymbol{R}+\delta \boldsymbol{e}_x} + \hat{\boldsymbol{c}}^\dagger_{\boldsymbol{R}}\, \boldsymbol{e}_- \cdot \boldsymbol{\tau}\, \hat{\boldsymbol{c}}_{\boldsymbol{R}+\delta \boldsymbol{e}_y} + \text{H.c.} \right) \\
\mathcal{H}_{\text{Yuk}}^{C_{4v}} &= \xi \sum_{\boldsymbol{R}} s_{\boldsymbol{R}} \hat{\boldsymbol{c}}^\dagger_{\boldsymbol{R}} \boldsymbol{e}_y \cdot \boldsymbol{\tau} \hat{\boldsymbol{c}}_{\boldsymbol{R}} \\
\mathcal{H}_\mu^{C_{4v}} &= \mu \sum_{\boldsymbol{R}} \hat{\boldsymbol{c}}^\dagger_{\boldsymbol{R}} \hat{\boldsymbol{c}}_{\boldsymbol{R}}
\end{aligned}$$

In the above, $\boldsymbol{\tau}$ denotes a vector of Pauli matrices that act on the *orbital* space, $\boldsymbol{e}_{\pm} = \frac{1}{\sqrt{2}}\left(\boldsymbol{e}_x \pm \boldsymbol{e}_y\right)$. Further, $\boldsymbol{R} \in A$ denotes the sum of the A sub-lattice, $(-1)^{R_x+R_y} = 1$. Furthermore, we have to consider an arbitrary set of Ising spins $s_{\boldsymbol{R}} = \pm 1$. Consider the relation

$$U^{\dagger}(\boldsymbol{e},\theta)\boldsymbol{\tau}U(\boldsymbol{e},\theta) = R(\boldsymbol{e},\theta)\boldsymbol{\tau}.$$

$U(\boldsymbol{e},\theta) = e^{-i\theta\boldsymbol{e}\cdot\boldsymbol{\tau}/2}$ is an SU(2) rotation of angle θ around axis $\boldsymbol{e}$ ($|\boldsymbol{e}| = 1$) and $R(\boldsymbol{e},\theta)$ an SO(3) with the same angle and axis. We can hence carry out a canonical transformation,

$$\hat{d}_{\boldsymbol{R}} = U\hat{c}_{\boldsymbol{R}},$$

that rotates $\boldsymbol{e}_+ \to \boldsymbol{e}_z$, $\boldsymbol{e}_- \to \boldsymbol{e}_x$, and $\boldsymbol{e}_y \to -\frac{1}{\sqrt{2}}\left(\boldsymbol{e}_x - \boldsymbol{e}_z\right)$ by combing a $\pi/4$ rotation around the z-axis and subsequently a $\pi/2$ rotation around the x-axis. After this canonical transformation, the Hamiltonian is real, and takes the form:

$$\begin{aligned}
\mathcal{H}_0^{C_{4v}} &= -t\sum_{\boldsymbol{R}\in A,\delta=\pm}\left(\hat{\boldsymbol{d}}_{\boldsymbol{R}}^{\dagger}\,\tau_z\,\hat{\boldsymbol{d}}_{\boldsymbol{R}+\delta\boldsymbol{e}_x} + \hat{\boldsymbol{d}}_{\boldsymbol{R}}^{\dagger}\,\tau_x\,\hat{\boldsymbol{d}}_{\boldsymbol{R}+\delta\boldsymbol{e}_y} + \text{H.c.}\right)\\
\mathcal{H}_{\text{Yuk}}^{C_{4v}} &= -\frac{\xi}{\sqrt{2}}\sum_{\boldsymbol{R}} s_{\boldsymbol{R}}\hat{\boldsymbol{d}}_{\boldsymbol{R}}^{\dagger}\left(\tau_x - \tau_z\right)\hat{\boldsymbol{d}}_{\boldsymbol{R}}\\
\mathcal{H}_{\mu}^{C_{4v}} &= \mu\sum_{\boldsymbol{R}}\hat{\boldsymbol{d}}_{\boldsymbol{R}}^{\dagger}\hat{\boldsymbol{d}}_{\boldsymbol{R}}
\end{aligned}$$

We can now express the model in terms of Majorana fermions and choose

$$\hat{\boldsymbol{d}}_{\boldsymbol{R}}^{\dagger} = \frac{1}{2}\left(\hat{\boldsymbol{\gamma}}_{\boldsymbol{R},1} - i\hat{\boldsymbol{\gamma}}_{\boldsymbol{R},2}\right)$$

as representation for $(-1)^{R_x+R_y} = 1$ and

$$\hat{\boldsymbol{d}}_{\boldsymbol{R}}^{\dagger} = \frac{1}{2}\left(i\hat{\boldsymbol{\gamma}}_{\boldsymbol{R},1} + \hat{\boldsymbol{\gamma}}_{\boldsymbol{R},2}\right)$$

as representation for $(-1)^{R_x+R_y} = -1$. Let $\boldsymbol{\mu}$ be a vector of Pauli spin matrices that acts on the Majorana index. With this choice, the Hamiltonian then takes the form:

$$\begin{aligned}
\mathcal{H}_0^{C_{4v}} &= \frac{it}{2}\sum_{\boldsymbol{R}\in A,\delta=\pm}\left(\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}^{\mathrm{T}}\,\boldsymbol{\tau}_z\,\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}+\delta\boldsymbol{e}_x} + \hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}^{T}\,\boldsymbol{\tau}_x\,\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}+\delta\boldsymbol{e}_y}\right)\\
\mathcal{H}_{\text{Yuk}}^{C_{4v}} &= \frac{\xi}{4\sqrt{2}}\sum_{\boldsymbol{R}} s_{\boldsymbol{R}}\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}^{\mathrm{T}}\left(\boldsymbol{\tau}_x\boldsymbol{\mu}_y - \boldsymbol{\tau}_z\boldsymbol{\mu}_y\right)\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}\\
\mathcal{H}_{\mu}^{C_{4v}} &= \mu\sum_{\boldsymbol{R}}\left(2 - \hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}^{\mathrm{T}}\mu_y\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}\right)
\end{aligned}$$

Using the notation of Ref. [67] we define:

$$\hat{T}_1^{-}\alpha\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}\left(\hat{T}_1^{-}\right)^{-1} = \bar{\alpha}i\boldsymbol{\tau}_y\boldsymbol{\mu}_x\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}$$

and

$$\hat{T}_2^{-}\alpha\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}\left(\hat{T}_2^{-}\right)^{-1} = \bar{\alpha}i\boldsymbol{\tau}_y\boldsymbol{\mu}_z\hat{\boldsymbol{\gamma}}_{\boldsymbol{R}}$$

that satisfy

$$\left[\mathcal{H}^{C_{4v}},\hat{T}_1^{-}\right] = \left[\mathcal{H}^{C_{4v}},\hat{T}_2^{-}\right] = 0. \tag{2.21}$$

Both above symmetries square to (-1) and anti-commute with each other. This hence places us in the Kramers class, see Ref. [67, 70], and no negative sign problem occurs.

2.7 QMC Observables

In the following, we define the QMC observables used throughout this project to study the quantum phase transition. We have used quantities based on both bosonic and fermionic degrees of freedom.

2.7.1 Bosonic degrees of freedom

2.7.1.1 Order parameters

The structure factor $S(\boldsymbol{k})$ and susceptibility $\chi(\boldsymbol{k})$ are defined as

$$S(\boldsymbol{k}) = \frac{1}{N^2} \sum_{\langle \boldsymbol{R}, \boldsymbol{R}' \rangle} \left(\langle s^z_{\boldsymbol{R}} s^z_{\boldsymbol{R}'} \rangle - \langle s^z_{\boldsymbol{R}} \rangle \langle s^z_{\boldsymbol{R}'} \rangle \right) e^{i\boldsymbol{k}(\boldsymbol{R}-\boldsymbol{R}')}, \tag{2.22}$$

and

$$\chi(\boldsymbol{k}) = \int \mathrm{d}\tau \, \frac{1}{N^2} \sum_{\langle \boldsymbol{R}, \boldsymbol{R}' \rangle} \left(\langle s^z_{\boldsymbol{R}}(0) s^z_{\boldsymbol{R}'}(\tau) \rangle - \langle s^z_{\boldsymbol{R}}(0) \rangle \langle s^z_{\boldsymbol{R}'}(\tau) \rangle \right) e^{i\boldsymbol{k}(\boldsymbol{R}-\boldsymbol{R}')}. \tag{2.23}$$

Both $S(\boldsymbol{k}=0)$ and $\chi(\boldsymbol{k}=0)$ are suitable order parameters to probe the paramagnetic-ferromagnetic phase transition.

2.7.1.2 RG-invariant quantities

Next, we define a set of renormalization group (RG) invariant observables. These are quantities with vanishing scaling dimension, that in the thermodynamic limit either converge to 1 if the state is ordered (ferromagnetically, in the case of this project), or to 0 in the absence of order.

At a quantum critical point, RG-invariant quantities follow the form $f[L^z/\beta, (h-h_\mathrm{c})L^{1/\nu}, L^{-\Delta z}, L^{-\omega}]$. Here, we have taken into account the possibility of two characteristic length scales: $\Delta z = 1 - z_-/z_+$. Since our temperature is representative of the ground state, we can neglect the dependence on L^z/β. The term includes two corrections to scaling, the regular term —governed by an exponent ω— and another correction that is present if $z_- \neq z_+$. Up to these corrections, the data for different lattice sizes cross at the critical field h_c.

The first RG-invariant quantity is the well-known Binder ratio, B [34], defined as

$$B = \frac{1}{2} \left(3 - \frac{\langle (s^z)^4 \rangle}{\langle (s^z)^2 \rangle^2} \right). \tag{2.24}$$

Furthermore, we define two more RG-invariant quantities R_S and R_χ through the correlation ratio: Given a local order parameter O at momentum $\boldsymbol{p}$, one can define the correlation ratio R_O as

$$R_O \equiv 1 - \frac{O(\boldsymbol{p}+\delta\boldsymbol{p})}{O(\boldsymbol{p})} \quad \text{with} \quad O \in \{S, \chi\}, \tag{2.25}$$

where $O(\boldsymbol{p})$ is the two-point function of the order parameter in Fourier space, and $\delta\boldsymbol{p}$ is the minimum nonzero momentum on a finite lattice. For the C_{2v} model $\delta\boldsymbol{p} = (\pi/L, \pi/L)$ or $\delta\boldsymbol{p} = (\pi/L, -\pi/L)$, while for the C_{4v} model $\delta\boldsymbol{p} = (2\pi/L, 0)$ or $\delta\boldsymbol{p} = (0, 2\pi/L)$; as usual, one can average over the two minimum displacements to obtain an improved estimator. In principle R_O has the same asymptotical behavior for bigger values of $\delta\boldsymbol{p}$, as long as they scale with $1/L$, however, we have found that using the minimal versions works best for us.

2.7.1.3 Derivative of the free energy

To provide further information on the nature of the transition, we use the derivative of the free energy,

$$\frac{1}{N}\frac{\partial F}{\partial h} = \left\langle \frac{1}{N}\sum_{\boldsymbol{R}} s^x_{\boldsymbol{R}} \right\rangle \equiv X. \tag{2.26}$$

2.7.2 Fermionic degrees of freedom

The fermionic observables consist of the momentum-resolved single-particle gap $\Delta_{\text{sp}}(\boldsymbol{k})$ which we use to image the meandering of Dirac points. We furthermore use this quantity to determine the Fermi velocity anisotropy $v_\perp/v_\parallel$.

2.7.2.1 Fermionic single-particle gap

To properly define $\Delta_{\text{sp}}(\boldsymbol{k})$, we first introduce an energy basis:

$$\mathcal{H}\left|\Psi_n^N(\boldsymbol{k})\right\rangle = E_n^N(\boldsymbol{k})\left|\Psi_n^N(\boldsymbol{k})\right\rangle,$$

where $\left|\Psi_n^N(\boldsymbol{k})\right\rangle$ are also eigenstates of particle number $\hat{N}$ and momentum $\hat{\boldsymbol{k}}$ operators:

$$\hat{N}\left|\Psi_n^N(\boldsymbol{k})\right\rangle = N\left|\Psi_n^N(\boldsymbol{k})\right\rangle \qquad\qquad \hat{\boldsymbol{k}}\left|\Psi_n^N(\boldsymbol{k})\right\rangle = \boldsymbol{k}\left|\Psi_n^N(\boldsymbol{k})\right\rangle$$

In this basis, the gap is:

$$\Delta_{\text{sp}}(\boldsymbol{k}) = E_0^{N_0+1}(\boldsymbol{k}) - E_0^{N_0},$$

where N_0 is the particle number of the half-filled system.

Now consider the time-displaced Green function

$$G(\boldsymbol{k},\tau) = \left\langle \hat{c}_{\boldsymbol{k}}(\tau)\hat{c}_{\boldsymbol{k}}^\dagger \right\rangle \quad \text{with} \quad \hat{c}_{\boldsymbol{k}}(\tau) = e^{\tau\mathcal{H}}\hat{c}_{\boldsymbol{k}}e^{-\tau\mathcal{H}}.$$

Assuming a unique ground state, the $T = 0$ Green function reads:

$$\lim_{\beta\to\infty} G(\boldsymbol{k},\tau) = \left\langle \Psi_0^{N_0}\middle|\, \hat{c}_{\boldsymbol{k}}(\tau)\hat{c}_{\boldsymbol{k}}^\dagger \,\middle|\Psi_0^{N_0}\right\rangle = \sum_n e^{-\tau\left(E_n^{N_0+1}(\boldsymbol{k})-E_0^{N_0}\right)}\left|\left\langle \Psi_n^{N_0+1}(\boldsymbol{k})\middle|\, \hat{c}_{\boldsymbol{k}}^\dagger \,\middle|\Psi_0^{N_0}\right\rangle\right|^2.$$

Provided that the wave function renormalization, $\left|\left\langle \Psi_n^{N_0+1}(\boldsymbol{k})\middle|\, \hat{c}_{\boldsymbol{k}}^\dagger \,\middle|\Psi_0^{N_0}\right\rangle\right|^2$ is finite and that $|\Psi_0^{N_0+1}(\boldsymbol{k})\rangle$ is non-degenerate, then

$$\lim_{\tau\to\infty}\lim_{\beta\to\infty} G(\boldsymbol{k},\tau) = e^{-\tau\left(E_0^{N_0+1}(\boldsymbol{k})-E_0^{N_0}\right)}\left|\left\langle \Psi_0^{N_0+1}(\boldsymbol{k})\middle|\, \hat{c}_{\boldsymbol{k}}^\dagger \,\middle|\Psi_0^{N_0}\right\rangle\right|^2 \tag{2.27}$$

and we can extract $\Delta_{\text{sp}}(\boldsymbol{k}) = E_0^{N_0+1}(\boldsymbol{k}) - E_0^{N_0}$ by fitting the tail of $G(\boldsymbol{k},\tau)$ to an exponential form.

The source code used for this fit is displayed in *Appendix A.4* and *Appendix A.2.2.10* demonstrates how to apply the fitting function.

In Fig. 2.11 we show that this approach works, by comparing the dispersions deep in the disordered and ordered phases to mean field results. Note that in a fully ergodic QMC simulation we would sample both options for breaking the Ising $\mathbb{Z}_2$ symmetry. To produce the results of Figs. 2.11(a2,c2), we have omitted the global move that flips all the spins and comes with a unit acceptance.

We observe a slight systematic derivation between mean field results and QMC data in the disordered phase. This stems from fluctuations of the order parameter in the vicinity of the critical point.

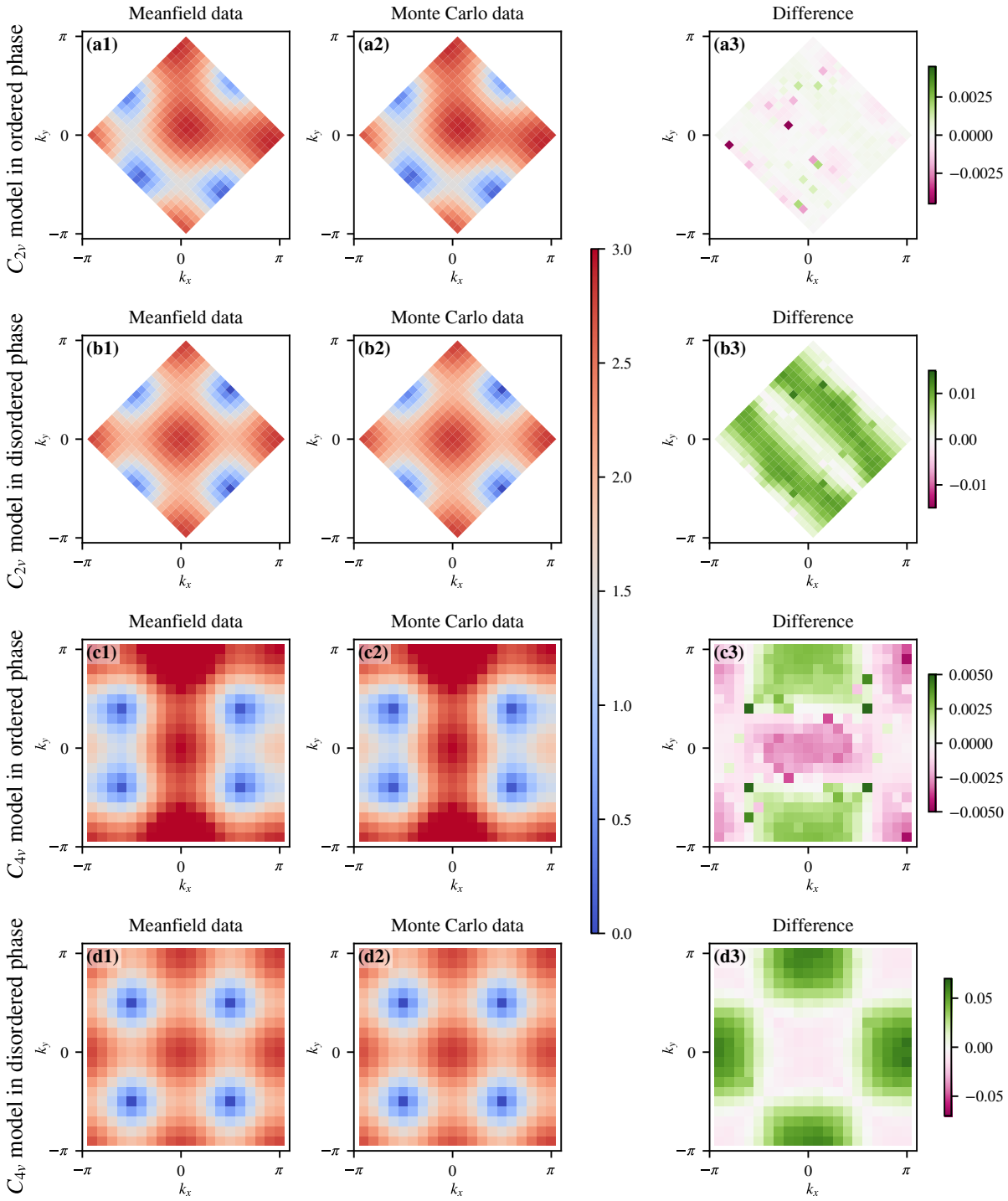


Fig. 2.11: Testing the dispersion of the fermionic single particle gap obtained from QMC data against mean field results. The first column shows the mean field results according to Eq. (2.13), while the second column shows numerical results obtained by employing Eq. (2.27) and the last column shows mean field minus QMC results. **(a)** C_{2v} model in ordered phase, at $h = 1.0$. The value of the mean field parameter ϕ is set to $\langle s^z \rangle$ from the QMC simulation. **(b)** C_{2v} model in disordered phase, at $h = 5.0$. The value of the mean field parameter is set to $\phi = 0$. **(c)** Same as (a), but for the C_{4v} model. **(d)** Same as (b), but for the C_{4v} model.

2.7.2.2 Fermi velocity anisotropy $v_\perp/v_\parallel$

With $\Delta_{\text{sp}}(\boldsymbol{k})$ we can extract the anisotropy at the nodal points $\boldsymbol{K}$, via,

$$\frac{v_\perp}{v_\parallel} = \lim_{\delta\to 0} \frac{\Delta_{\text{sp}}(\boldsymbol{K}+\delta\boldsymbol{e}_\perp) - \Delta_{\text{sp}}(\boldsymbol{K})}{\Delta_{\text{sp}}(\boldsymbol{K}+\delta\boldsymbol{e}_\parallel) - \Delta_{\text{sp}}(\boldsymbol{K})}. \tag{2.28}$$

Where $\boldsymbol{e}_\perp$ and $\boldsymbol{e}_\parallel$ are unit vectors perpendicular and parallel to the meandering direction of the Dirac cones. We have considered three different strategies for approaching this limit on the finite size lattices, that are all equivalent in the thermodynamic limit:

1. The direct approach. The most straightforward implementation of Eq. (2.28) on a finite lattice would be

$$\frac{v_\perp}{v_\parallel} = \frac{\Delta_{\text{sp}}(\boldsymbol{K}+\boldsymbol{\delta}_\perp) - \Delta_{\text{sp}}(\boldsymbol{K})}{\Delta_{\text{sp}}(\boldsymbol{K}+\boldsymbol{\delta}_\parallel) - \Delta_{\text{sp}}(\boldsymbol{K})}, \tag{2.29}$$

where $\boldsymbol{\delta}_\perp$, $\boldsymbol{\delta}_\parallel$ are the shortest distances from the nodal point on the finite-size $\boldsymbol{k}$-Lattice.

2. Manually setting the finite size gap $\Delta_{\text{sp}}(\boldsymbol{K}) = 0$. This approach makes sense, since we know that in the thermodynamic limit the gap vanishes. With this strategy, Eq. (2.28) takes the form

$$\frac{v_\perp}{v_\parallel} = \frac{\Delta_{\text{sp}}(\boldsymbol{K}+\boldsymbol{\delta}_\perp)}{\Delta_{\text{sp}}(\boldsymbol{K}+\boldsymbol{\delta}_\parallel)}. \tag{2.30}$$

3. Avoid the nodal points. Another approach for avoiding the finite size gap is to measure one step away from it:

$$\frac{v_\perp}{v_\parallel} = \frac{\Delta_{\text{sp}}(\boldsymbol{K}+2\boldsymbol{\delta}_\perp) - \Delta_{\text{sp}}(\boldsymbol{K}+\boldsymbol{\delta}_\perp)}{\Delta_{\text{sp}}(\boldsymbol{K}+2\boldsymbol{\delta}_\parallel) - \Delta_{\text{sp}}(\boldsymbol{K}+\boldsymbol{\delta}_\parallel)} \tag{2.31}$$

The results for these different approaches are shown in Fig. 2.12. The third strategy results in velocity anisotropies < 1, while Fig. 2.1(c) clearly shows that $v_\perp/v_\parallel > 1$ at the critical point. This implies that the approach strongly underestimates the anisotropy due to the fact that the considered lattices sizes are too small for not measuring directly at the nodal point.

The other two approaches, while not giving quantitatively the same results, are qualitatively equivalent. We have opted to use the second strategy, corresponding to Eq. (2.30).

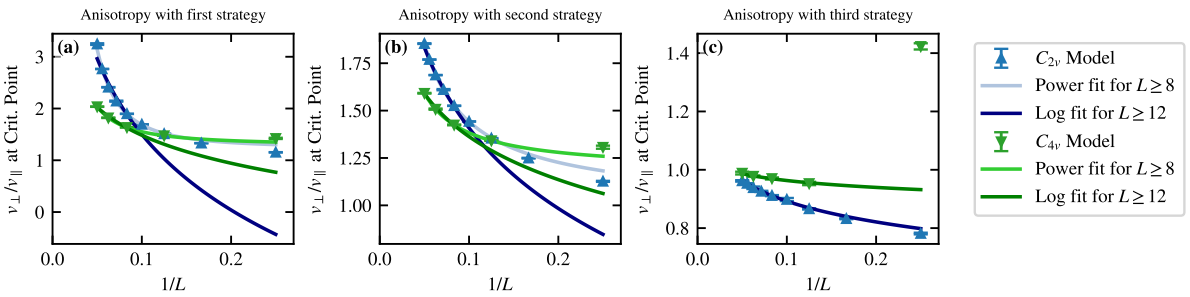


Fig. 2.12: Fermi velocity anisotropy at the critical point as function of $1/L$, determined with (a): Eq. (2.29), (b): Eq. (2.30), (c): Eq. (2.31). Power law and logarithmic fits are shown, except for (c), where only a logarithmic fit is performed.

2.8 QMC results

In this section, we first give an overview of the QMC results, showing that there is indeed a continuous phase transition, after which we determine the critical exponents and finally discuss odd-even effects found in the C_{4v} model.

2.8.1 Overview

Fig. 2.13 displays the previously defined bosonic observables close to the phase transition. The RG-invariant quantities defined in Eqs. (2.24) and (2.25) shown in Figs. 2.13(a-c) cross for different system sizes at $h_c \approx 3.25$ ($h_c \approx 3.65$) for the C_{2v} (C_{4v}) model, signifying a phase transition at that point. Both the order parameter $S(\boldsymbol{k} = 0)$ (Fig. 2.13(d)) and derivative of the free energy with respect to the tuning parameter, $\partial F/\partial h$ (Fig. 2.13(e)) do not show any discontinuity, supporting a continuous phase transition.

Due to finite size effects, the RG-invariant quantities do not cross at the same point, but display a drift with system size. Fig. 2.14(c,d) show the crossing points between L and $L + \Delta L$ lattices, with $\Delta L = 2$ (4) for the C_{2v} (C_{4v}) model. As apparent, we obtain consistent results for h_c when considering different RG-invariant quantities. We estimate the correlation-length exponents $1/\nu$ by data collapse for the two models in Fig. 2.14(a,b). Considering values of $L \geq L_{\text{min}} = 12$ we obtain $1/\nu = 1.376(6)$ [$1/\nu = 1.38(1)$] for the C_{2v} [C_{4v}] model. These values are in the ballpark of the ϵ-expansion results in the quasiuniversal regime (cf. *Appendix A.1*). Section 2.8.2 will give data for various values of L_{min}, that are standing in agreement with the above values. Although seemingly converged, the fact that the velocity anisotropy is expected to flow extremely slowly suggest that the exponents are subject to considerable size effects, see Section 2.5. The impact of critical fluctuations on the fermion spectrum is displayed in Figs. 2.1(b,c). In the disordered phase, Fig. 2.1(b), the dispersion relation exhibits rotational symmetry around the Dirac points. On the other hand, at criticality, Fig. 2.1(c), the dispersion relation suggests a velocity anisotropy, $v_\parallel < v_\perp$ at the Dirac point. Figure 2.14(e) demonstrates that this anisotropy grows as a function of system size, in qualitative agreement with the RG predictions. Although our system sizes are too small to detect convergence or divergence of the velocity ratio, we find it reassuring that its dependence on system size qualitatively resembles the scale dependence predicted from the integrated RG flow; cf. Fig. 2.14(e) with Fig. 2.10.

2.8.2 Critical exponents

2.8.2.1 Correlation length exponent ν from RG invariant quantities.

A renormalization-group invariant quantity, R, has by definition a vanishing scaling dimension. Consider a system at temperature β, of size $L_+ \times L_-$ with a single relevant coupling h. Under a renormalization group transformation that rescales $L_+ \to L_+/b$ with $b > 1$, we expect [71]:

$$R((h - h_c), \beta, L_+, L_-) = R((h - h_c)', \beta/b^z, L_+/b, L_-/b^{1+\Delta z}). \tag{2.32}$$

In the above $\Delta z \neq 0$ encodes the difference in scaling between the L_+ and L_- directions. Linearization of the RG transformation, $(h - h_c)' = b^{1/\nu}(h - h_c)$ and setting the scale $b = L$ as well as $L_- = L_+ = L$, in accordance to our simulations, yields:

$$R\left((h - h_c), \beta, L\right) = f\left(L^{1/\nu}(h - h_c), L^z/\beta, L^{-\Delta z}, L^{-\omega}\right). \tag{2.33}$$

In the above we have accounted for possible corrections to scaling $L^{-\omega}$. In the presence of a single length scale $\Delta z = 0$, such that the generic finite size scaling form is recovered.

Since in our simulations the temperature is representative of the ground state, we can neglect the dependence on L^z/β. Up to corrections to scaling, ω, and the possibility of $\Delta z \neq 0$, which would result in another correction to scaling term, the data for different lattice sizes cross at the critical field h_c and should collapse when plotted as function of $(h - h_c)L^{1/\nu}$. The results for such data collapses are shown in Table 2.1 and Table 2.2 (cf. *Appendix A.2.2.8* for a demonstration on how such a data collapse can be carried out). Furthermore, Fig. 2.15 shows $1/\nu$ for the C_{2v} and C_{4v} models from pairwise data collapse of RG-invariant quantities, using system sizes L and $L + 2$ ($L + 4$). Both suggest a relatively well converged result for $L \geq 12$. Although seemingly converged, our system sizes are too small to detect the logarithmic drift in exponents suggested by the RG analysis.

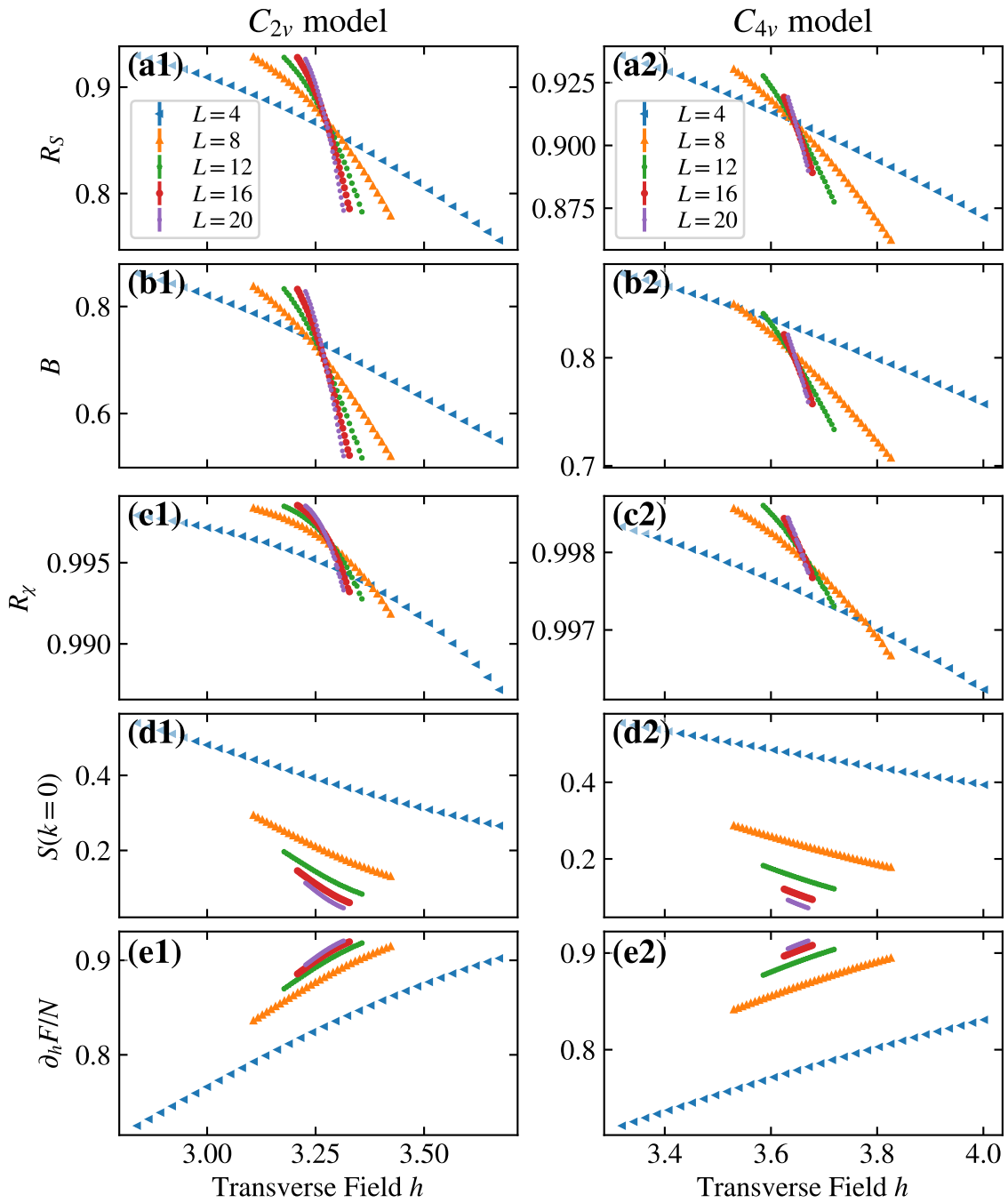


Fig. 2.13: Bosonic observables as function of transverse Ising field h close to the critical point $h = h_c$. (a-c) The RG-invariant quantities: Structure factor correlation ration R_S, Binder ratio B and Susceptibility correlation ratio R_χ. The point where lines for different system sizes cross indicates a phase transition. (d) Order parameter $S(\boldsymbol{k} = 0)$ [Eq. (2.22)]. (e) Derivative of free energy, exhibiting no discontinuities.

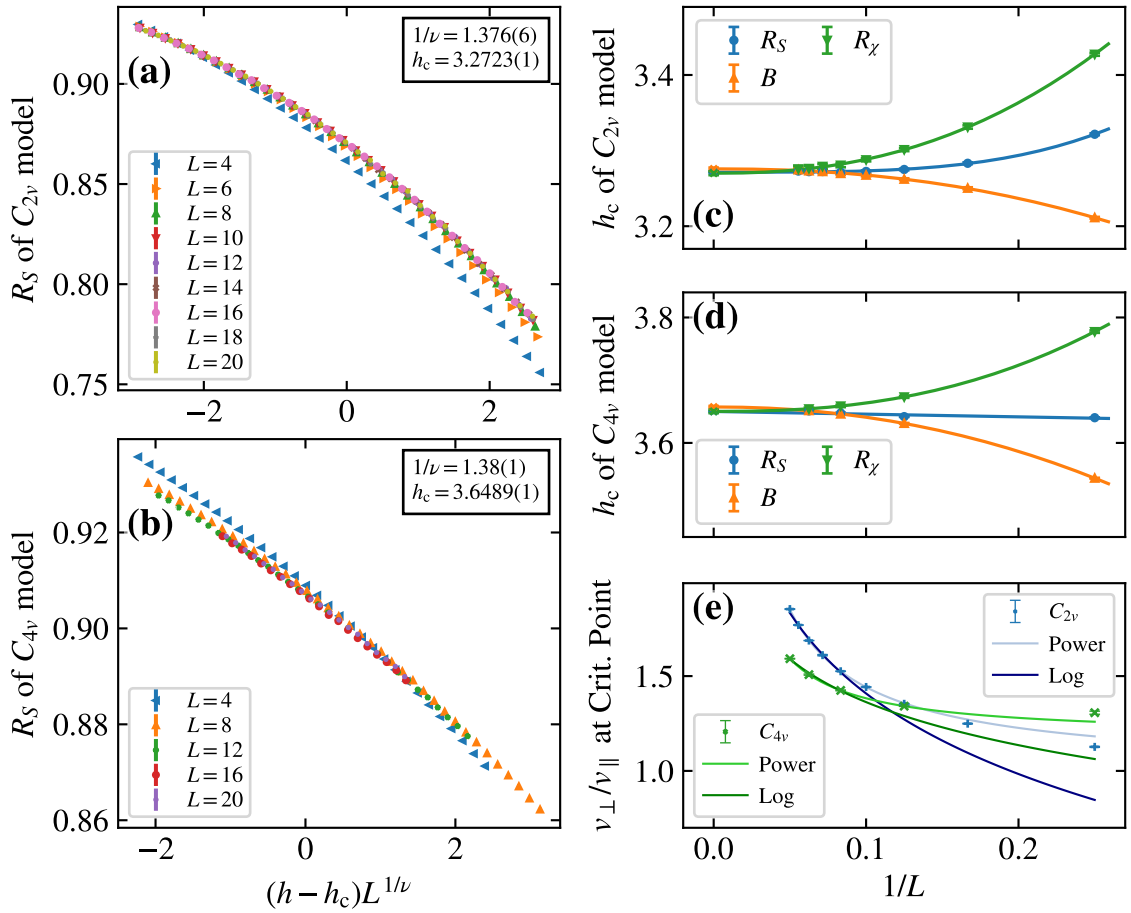


Fig. 2.14: (a) R_S as function of $(h - h_c)L^{1/\nu}$ for the C_{2v} model, revealing a data collapse for $L \gtrsim 12$, assuming $1/\nu = 1.376$. (b) Same as (a), but for C_{4v} model, assuming $1/\nu = 1.38$. (c) Crossing points of different RG-invariant quantities as function of $1/L$ with $\Delta L = 2$ in C_{2v} model, indicating a unique critical point $h_c = 3.27$ for $L \to \infty$. (d) Same as (c), but for C_{4v} model and $\Delta L = 4$, extrapolating to $h_c = 3.65$. (e) Ratio of Fermi velocities $v_\perp/v_\parallel$ as function of $1/L$ at h_c, revealing that the velocity anisotropy increases with increasing system size. The solid lines show power law fits for $L \geq 8$ and logarithmic fits for $L \geq 12$.

Table 2.1: Data collapse results for RG-invariant quantities of C_{2v} model.

Observables	Used system sizes	h_c	$1/\nu$	χ^2
R_S	8, 10, 12, 14, 16, 18, 20	3.272715 ± 0.000074	1.358934 ± 0.001824	2.4
R_S	10, 12, 14, 16, 18, 20	3.272222 ± 0.000065	1.373499 ± 0.002887	1.8
R_S	12, 14, 16, 18, 20	3.272304 ± 0.000099	1.376110 ± 0.006104	1.9
R_S	14, 16, 18, 20	3.272617 ± 0.000138	1.375085 ± 0.007038	1.8
R_S	16, 18, 20	3.272521 ± 0.000274	1.368184 ± 0.014756	1.9
R_S	18, 20	3.273322 ± 0.000449	1.373454 ± 0.022000	1.9
B	8, 10, 12, 14, 16, 18, 20	3.270955 ± 0.000089	1.317011 ± 0.002273	13.4
B	10, 12, 14, 16, 18, 20	3.271532 ± 0.000141	1.344485 ± 0.004371	4.6
B	12, 14, 16, 18, 20	3.272535 ± 0.000175	1.352409 ± 0.006242	3.0
B	14, 16, 18, 20	3.273181 ± 0.000152	1.361056 ± 0.007298	2.6
B	16, 18, 20	3.273783 ± 0.000210	1.370788 ± 0.013022	2.2
B	18, 20	3.274325 ± 0.000464	1.340845 ± 0.028414	1.8
R_χ	8, 10, 12, 14, 16, 18, 20	3.281138 ± 0.000030	1.421387 ± 0.000002	22.0
R_χ	10, 12, 14, 16, 18, 20	3.279752 ± 0.000073	1.381046 ± 0.000002	7.3
R_χ	12, 14, 16, 18, 20	3.275155 ± 0.000500	1.369774 ± 0.001609	5.8
R_χ	14, 16, 18, 20	3.277021 ± 0.000233	1.338662 ± 0.010808	2.2
R_χ	16, 18, 20	3.276434 ± 0.000176	1.342788 ± 0.004183	2.2
R_χ	18, 20	3.275856 ± 0.000693	1.369095 ± 0.034679	2.4

Table 2.2: Data collapse results for RG-invariant quantities of C_{4v} model.

Observables	Used system sizes	h_c	$1/\nu$	χ^2
R_S	8, 12, 16, 20	3.64606 ± 0.00007	1.328 ± 0.006	18.7
R_S	12, 16, 20	3.64886 ± 0.00011	1.381 ± 0.011	3.2
R_S	16, 20	3.65108 ± 0.00022	1.402 ± 0.023	1.7
B	8, 12, 16, 20	3.64108 ± 0.00009	1.254 ± 0.007	73.2
B	12, 16, 20	3.64818 ± 0.00012	1.340 ± 0.014	5.1
B	16, 20	3.65138 ± 0.00025	1.362 ± 0.026	1.8
R_χ	8, 12, 16, 20	3.66319 ± 0.00027	1.309 ± 0.018	16.5
R_χ	12, 16, 20	3.65708 ± 0.00031	1.368 ± 0.028	3.1
R_χ	16, 20	3.65537 ± 0.00070	1.428 ± 0.092	2.4

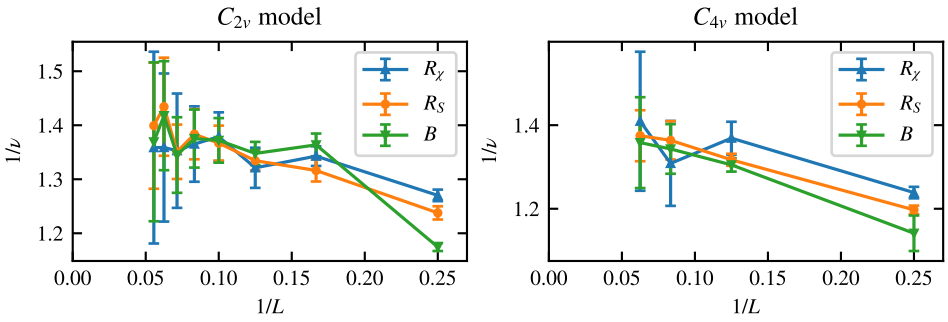


Fig. 2.15: Critical exponent $1/\nu$ of C_{2v} (C_{4v}) model from pairwise data collapse of RG-invariant quantities, using linear system sizes L and $L+2$ ($L+4$).

2.8.2.2 Scaling dimensions and scaling anisotropy

Next, we examine the scaling dimension of the bosonic field from the Ising spin correlations:

$$S(\boldsymbol{x}) = \langle \hat{s}^z_{\mathbf{R}}(\tau)\hat{s}^z_{\mathbf{0}}(0)\rangle \tag{2.34}$$

where $\boldsymbol{x} = (\boldsymbol{R}, \tau)$ is a space-time coordinate. The models considered in this research are not Lorentz invariant such that the scaling dimension acquires a direction dependence. Following Eq. (2.33) we expect:

$$S\left(r\hat{\boldsymbol{d}}_*, h\right) \propto \frac{1}{|r\hat{\boldsymbol{d}}_*|^{2\Delta_{s,*}}} f\left(L^z/\beta, (h-h_c)L^{1/\nu}, L^{-\Delta z}, L^{-\omega}\right) \tag{2.35}$$

where $\hat{\boldsymbol{d}}_*$ defines the direction.

To determine the scaling dimensions, we consider $S(L\hat{\boldsymbol{d}}_*, h)$, for different system sizes L and use an RG-invariant quantity R to replace in leading order $f(L^z/\beta, (h-h_c)L^{1/\nu}, L^{-\Delta z}, L^{-\omega}) = \tilde{f}(R)$. Using this form, we perform data collapses using system sizes L and $L+2$ (L and $L+4$), where the only free parameter is $\Delta_{s,*}$. The considered directions are defined in Table 2.3, the C_{4v} symmetry of the second model enforces $\Delta_{s,x} = \Delta_{s,y}$ and $\Delta_{s,+} = \Delta_{s,-}$. As the results in Fig. 2.16 and Fig. 2.17 show, we cannot resolve a scaling anisotropy between the chosen directions. We conjecture that anisotropies in the exponents will emerge in the infrared limit. Given the very slow flow we believe that our numerical simulations are not in a position to probe these energy scales.

Table 2.3: Considered directions for the scaling dimension.

$*$	$\hat{\boldsymbol{d}}_*$
x	$(\hat{\boldsymbol{e}}_x, 0)$
y	$(\hat{\boldsymbol{e}}_y, 0)$
$+$	$\frac{1}{2}(\hat{\boldsymbol{e}}_x + \hat{\boldsymbol{e}}_y, 0)$
$-$	$\frac{1}{2}(\hat{\boldsymbol{e}}_x - \hat{\boldsymbol{e}}_y, 0)$
τ	$(0, 0.3)$

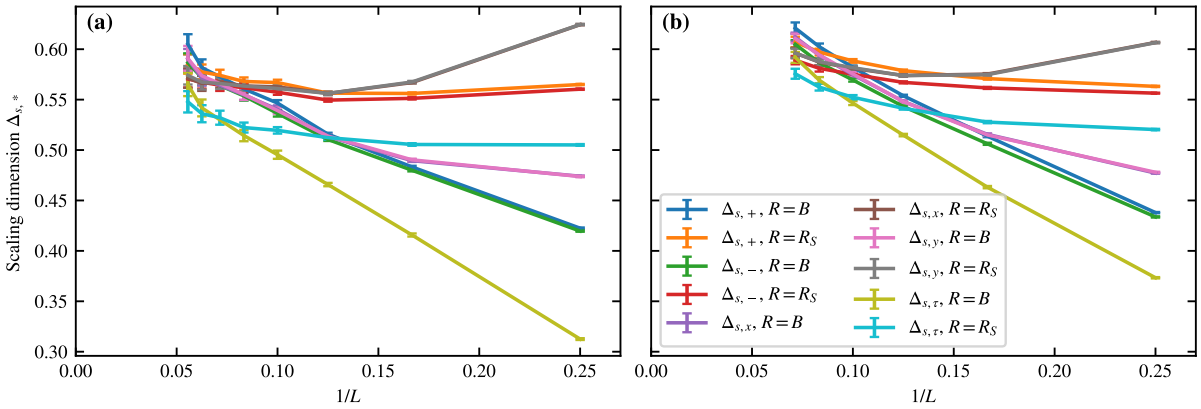


Fig. 2.16: Scaling dimension of Ising field of C_{2v} model. For **(a)** $\xi = 0.25$, **(b)** $\xi = 0.4$. Note: $\Delta_{s,y}$, $R = B$ is indistinguishable from $\Delta_{s,x}$, $R = B$ and $\Delta_{s,y}$, $R = R_S$ is identical to $\Delta_{s,x}$, $R = R_S$.

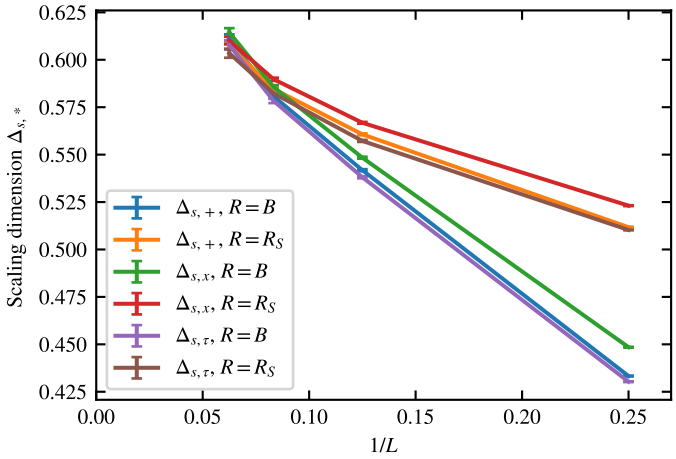


Fig. 2.17: Scaling dimension of Ising field of C_{4v} model.

2.8.2.3 Dynamical exponent z

To determine the dynamical exponent of the C_{4v} model, we assume isotropic scaling in space, as suggested by the RG analysis. Then the RG-invariant quantities follow the form

$$R = f(L^z/\beta, (h - h_c)L^{1/\nu}, L^{-\omega}) \tag{2.36}$$

at the critical point.

At the crossing points $h_*(L)$, with $R(h_*(L), L) = R(h_*(L), L + \Delta_L)$ and $\Delta_L = 4$, we measure $R(\beta)$. Omitting corrections to scaling leads to $R(L, \beta) = f(L^z/\beta)$. From this we derive

$$z = \frac{\log\left(\frac{\partial_\beta R(L)}{\partial_\beta R(L+\Delta_L)}\right)}{\log\left(\frac{L+\Delta_L}{L}\right))}. \tag{2.37}$$

The results are shown in Fig. 2.18, and are consistent with $z = 1$ as suggested in the RG analysis.

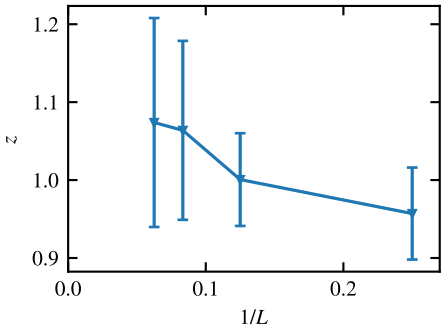


Fig. 2.18: Dynamical exponent z of C_{4v} model.

2.8.3 Odd-even effects

The C_{4v} model has strong odd-even effects. For linear system sizes $L \in 4\mathbb{N}$ (≡even) and periodic boundary conditions, the Dirac points are included in the discrete set of $\boldsymbol{k}$ vectors. This is not the case for odd lattices, $L \in 4\mathbb{N}+2$. Interestingly, the value of the Binder and correlation ratios depend on this choice of the boundary, see Fig. 2.19(b),(c),(d). We believe that this stems form the fact that both quantities do not have a well defined thermodynamic limit at $h = h_c$. i.e. $\lim_{L\to\infty} R_O(h = h_c)$ is mathematically not defined. However, the free energy (Fig. 2.19(a)), the critical field (Fig. 2.20(a)) and the exponents (Figs. 2.20(b-d)), should ultimately converge to the same value. On odd lattices, finite size effects seem to be larger.

The critical exponents $2\beta/\nu$ and η in Figs. 2.20(c,d), stem from the scaling assumptions

$$S(\boldsymbol{k}=0, h=h_c, L) \propto L^{2\beta/\nu} \qquad \chi(\boldsymbol{k}=0, h=h_c, L) \propto L^{2-\eta_\phi}, \tag{2.38}$$

where we omitted, as before, the dependence on the inverse temperature β and on corrections to scaling. Replacing h_c by the crossing point $h_*(L)$ of an RG-invariant quantity R, meaning $R(h_*(L), L) = R(h_*(L), L+4)$ with $R \in \{R_S, R_\chi, B\}$, we obtain:

$$2\beta/\nu = \log\left(\frac{S(\boldsymbol{k}=0, L+4, h=h_*(L))}{S(\boldsymbol{k}=0, L, h=h_*(L))}\right) / \log\left(\frac{L+4}{L}\right), \tag{2.39}$$

and

$$\eta_\phi = 2 - \log\left(\frac{\chi(\boldsymbol{k}=0, L+4, h=h_*(L))}{\chi(\boldsymbol{k}=0, L, h=h_*(L))}\right) / \log\left(\frac{L+4}{L}\right). \tag{2.40}$$

As apparent in Fig. 2.20, h_c has the smallest corrections to scaling when determined from R_S. However, the smallest corrections to scaling are when determining the critical exponents $2\beta/\nu$, η_ϕ and z, are obtained by using h_c as determined from R_χ. Finally, the velocity anisotropy at the critical point grows in both cases, but is much smaller for odd system sizes, cf. Fig. 2.21.

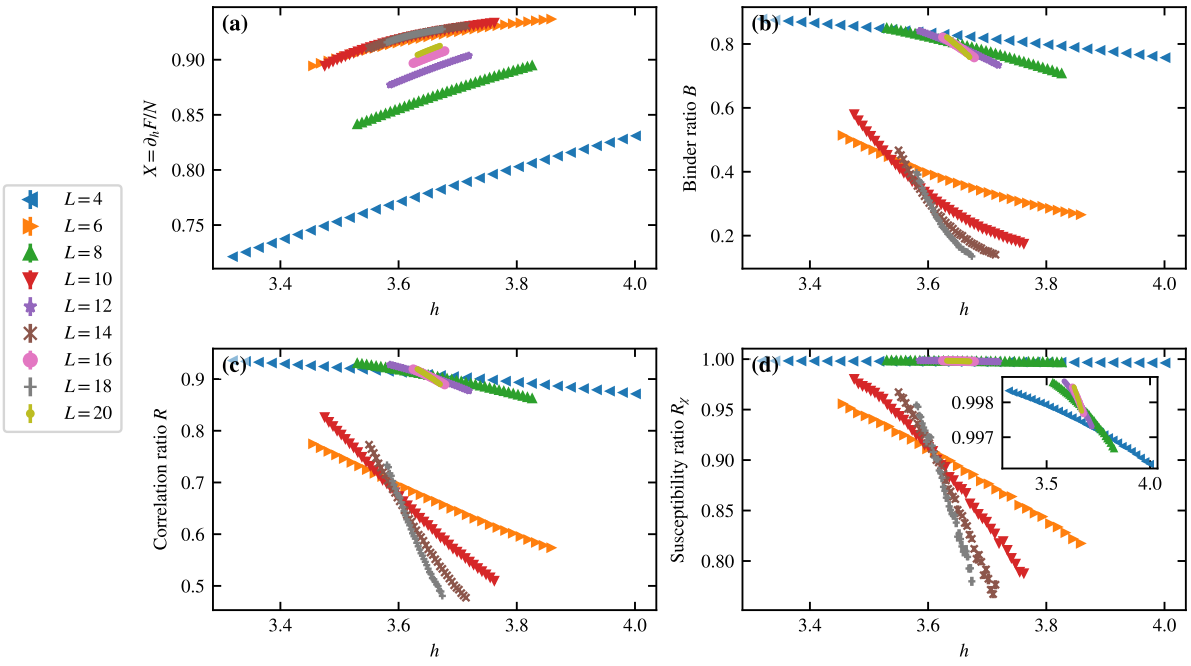


Fig. 2.19: Derivative of free energy and three RG-invariant quantities, showing a continuous transition around $h \approx 3.65$. Notable is an odd-even effect between linear system sizes $L \in 4\mathbb{N}$ (=even) and $L \in 4\mathbb{N}+2$ (=odd).

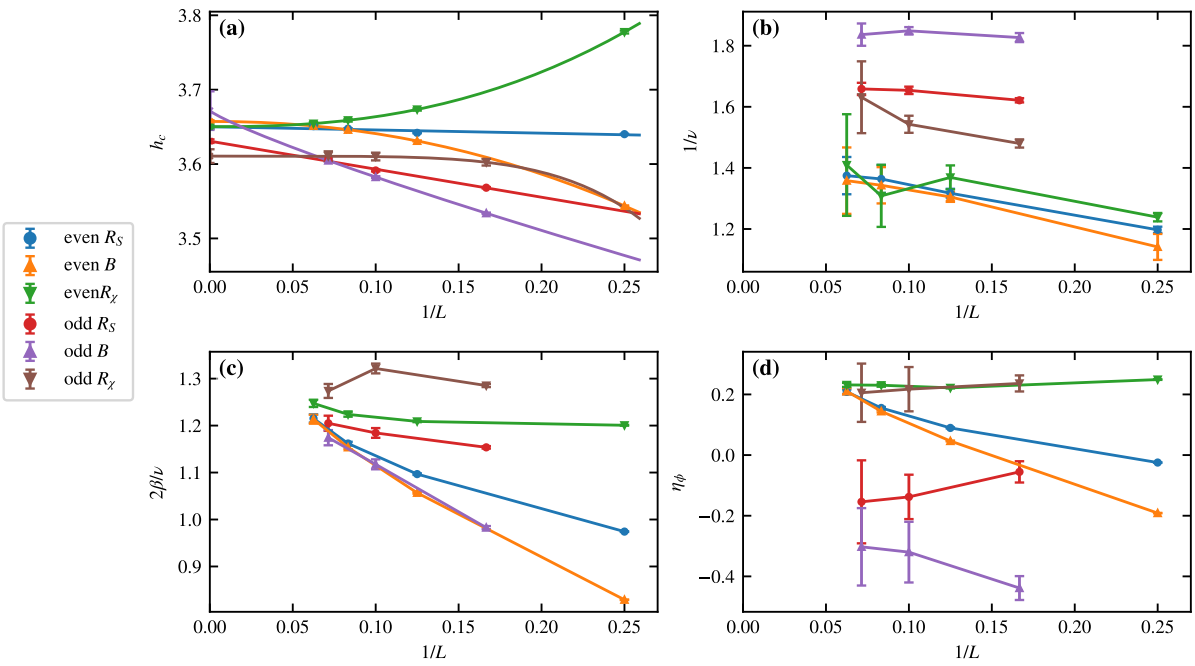


Fig. 2.20: Demonstration of odd-even effects for the C_{4v} model. **(a)**: Critical field h_c, extracted from the three RG-invariant quantities as determined by the crossing points between linear system sizes L and $L+4$. Odd and even system sizes show different behavior. Even system size shows better convergence. **(b)**: Critical exponent $1/\nu$ as determined by data collapse of the three RG-invariant quantities, correlation ratio R, Binder ratio B and susceptibility ratio R_χ, for linear system sizes L and $L+4$. Odd and even system sizes show different behavior. Even system size shows better convergence. **(c)**: Critical exponent $2\beta/\nu$ as determined with Eq. (2.39). **(d)**: Critical exponent η_ϕ as determined with Eq. (2.40).

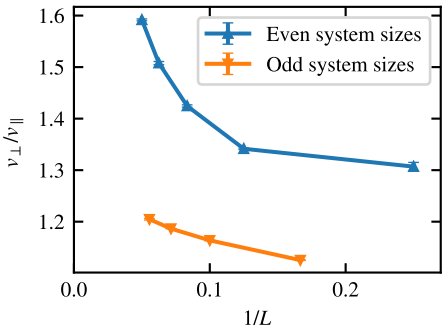


Fig. 2.21: Odd-even effects for the C_{4v} model on the anisotropy velocity of Dirac cones at the critical point.

2.9 Summary

Both the ϵ-expansion analysis and the QMC simulations show that our two symmetry distinct models of Dirac fermions support continuous nematic transitions. In both cases, the key feature of the quantum critical point is a velocity anisotropy that is best seen in the QMC data of Fig. 2.1(c). For the C_{4v} model, the ϵ-expansion shows that it diverges logarithmically with system size, in agreement with previous large-N results [52]. This law is supported by finite-size analysis based on QMC data up to linear system size $L = 20$, which is close to the upper bound allowed by current computational approaches. Since the effective exponents flow with the velocity anisotropy, we foresee that lattice sizes beyond the reach of our numerical approach and experiments at ultralow temperatures will be required to obtain converged values. The QMC data captures a quasiuniversal regime [65, 66], in which irrelevant operators aside from the velocity anisotropy die out. In fact, the RG prediction for exponents in this intermediate-energy regime is roughly consistent with the finite-size QMC measurements, Fig. A2(c). Furthermore, for a reasonable set of starting values, the integrated RG flows of the two models are initially very similar and deviate from each other only at very low energy scales. A similar behavior of the two models is also observed in the QMC data.

An advantage of our models is that the Dirac points are pinned by symmetry, such that QMC approaches that take momentum-space patches around these points into account [72] represent an attractive direction for future work. Our models equally allow for large-N generalizations, such that QMC and analytical large-N calculations can be compared as a function of increasing N. Finally, we can make contact to nematic transitions in $(2 + 1)$-dimensional Fermi liquids [48, 49], since our models do not suffer from the negative-sign problem under doping.

CHAPTER

THREE

PHASE DIAGRAM OF THE SU(N) ANTIFERROMAGNET OF SPIN S ON A SQUARE LATTICE

The results presented in this chapter are the outcome from a collaboration with Francesco Parisen Toldin and Fakher F. Assaad. These findings have been published in [17], with significant portions reproduced here verbatim. My contribution to the project comprise the quantum Monte Carlo (QMC) calculations, including the implementation of the model in code. The QMC model has been designed by F. F. A., while the group-theoretical proof for the projections has been executed by F. P. T.. The interpretation of data and written text is a combined work of all authors.

3.1 Introduction

Spin systems are ubiquitous in nature and form one of the most fundamental concept in condensed matter and statistical physics. Their complex collective behavior has spurred numerous experimental and theoretical studies, aimed at understanding their nature and properties. At the same time, modeling of spin systems represents a primary theoretical laboratory to investigate fundamental physics. Starting with the classical Ising model [33], spin systems have played an crucial role in our understanding of phase transitions [37], phases of matter, frustration and disorder [73], emergent gauge theories [74, 75] and exotic critical behavior [76]. The impact of spin models extends beyond the realm of condensed matter physics, and has found application in other areas, such as information processing [77] and quantum computing [78], where the fundamental unit of information, a qubit, is a single spin-$1/2$ system.

In condensed matter, spin systems are realized in Mott insulators, which arise when charge fluctuations in a given unit cell are suppressed. For instance, in undoped cuprates the copper atom is in a Cu^{2+} state and corresponds to a net spin $S = 1/2$ degree of freedom. Super-exchange leads to an $S = 1/2$, SU(2) Heisenberg spin model that has been studied numerically [79, 80] and experimentally [81] at length. Higher spin SU(2) systems arise when $2S$ electrons are localized on a single orbital and a strong Hund's rule favors a maximal spin state with a totally symmetric wave function. For example, in the Haldane chain realized by the $CsNiCl_3$ compound Ni^{2+} ions carry spin 1 [82]. SU(N)-invariant models, for $N > 2$, naturally arise as special cases of the Kugel-Khomski model [83, 84], where spin and orbital degrees of freedom turn out to play a very symmetric role. In particular, the observed spin-orbital liquid behavior in $Ba_3CuSb_2O_9$ [85] has been interpreted in terms of an SU(4) quantum antiferromagnet in the defining representation [86]. Beyond the solid state physics, SU(N) spin models can be realized in the realm of cold atomic gases [87, 88].

Topology plays a decisive role in the understanding of SU(2) invariant spin systems. In fact, using a spin coherent-state path integral approach to antiferromagnetic (AFM) Heisenberg chains, one identifies a Berry phase. It corresponds to the skyrmion count of the three-components normalized order parameter in 1+1 dimensions and at angle $\theta = 2\pi S$ [89]. This provides a topological understanding of the observed differences between half-integer and integer spin chains. In two spatial dimensions, topology enters through singular skyrmion number changing events in space time: monopoles [90]. For a square lattice with C_4 symmetry, only quadrupole (double) monopole events are allowed for half-integer (odd) spin by symmetry. There is no constraint on the monopole number for even values of the spin. For the plain vanilla SU(2) Heisenberg model at arbitrary spin S, the spin-wave approximation captures well the ground state and topological excitations lie high in the spectrum. In this context, the theory of deconfined quantum criticality essentially poses the question of the nature of the quantum phase transition that emanates when one decreases the energy of monopoles and ultimately condenses them [76]. For half-integer spin systems, where only quadrupole monopole insertions are allowed, one can conjecture that the Hilbert space splits into four orthogonal subspaces characterized by the number of monopoles modulo four. This provides an understanding of how

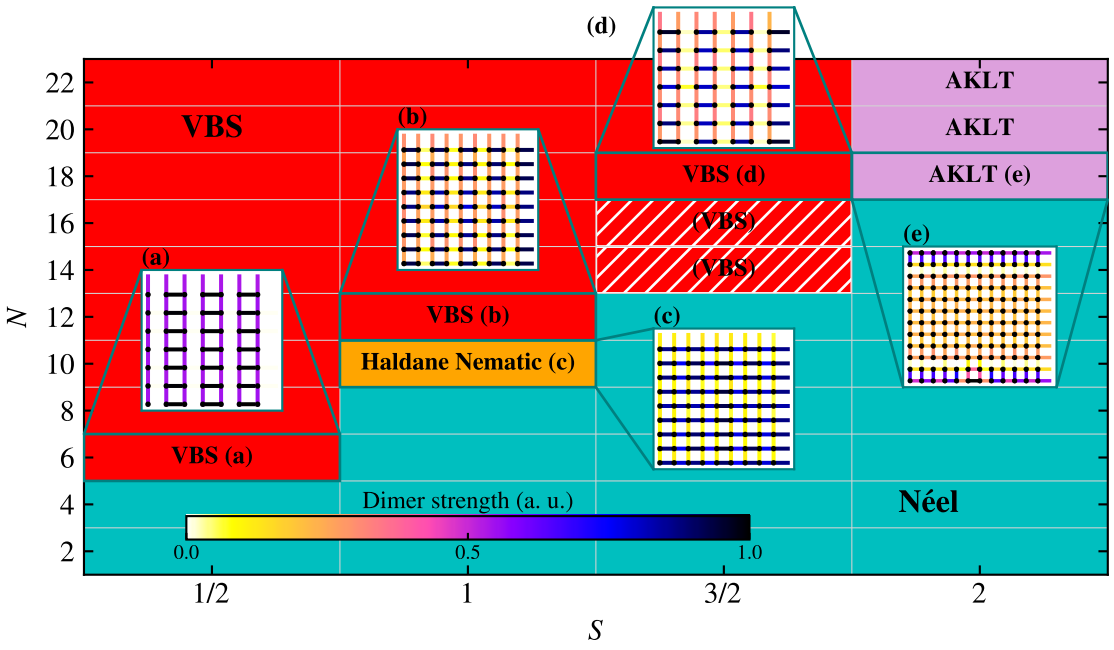


Fig. 3.1: Ground-state phase diagram of the SU(N)-antiferromagnet model (3.1) on the square lattice, as obtained from QMC simulations. S identifies the chosen representation of the $\mathfrak{su}(N)$ algebra of SU(N), illustrated by the Young tableau in Fig. 3.2. Striped regions indicate the part of the phase diagram where current QMC data do not allow an unambiguous identification of the phase; in such cases we indicate between parenthesis the most likely identified order. The insets show QMC data in the highlighted dimerized phases, obtained through a pinning-field approach (see Sec. 3.4.1).

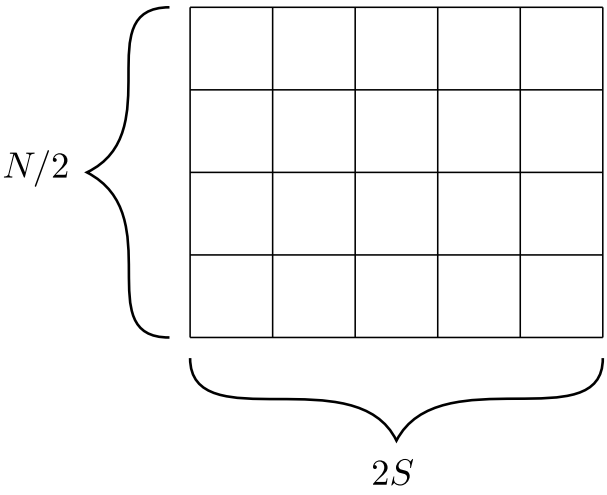


Fig. 3.2: Young tableau corresponding to the irreducible representation of $\mathfrak{su}(N)$ considered here; N is even and S can take half-integer and integer values.

the fourfold degenerate valence bond solid (VBS) state emerges for condensing topological excitations of the quantum antiferromagnet [91]. Similarly, for spin-1 (spin-2) systems, condensing monopoles should generate a twofold (zerofold) degenerate disordered state.

A crucial question is how to control the monopole energy. The seminal work of Read and Sachdev [18, 20, 21] shows that the discussion above can be carried over to SU(N) spin systems, $N \geq 2$. Furthermore, enhancing N has the potential of lowering the monopole energy. In this work, we show that it is possible to formulate negative sign-free auxiliary-field (AF) quantum Monte Carlo (QMC) simulations [8, 9, 10, 12, 92] of the SU(N) AFM spin-S Heisenberg model, for representations given by a Young tableau with $N/2$ rows and $2S$ columns. This generalizes the work of Ref. [93] to generic values of S. Specifically, we consider the model:

$$\hat{H} = J \sum_{\langle i,j\rangle,a} \hat{S}_i^{(a)} \hat{S}_j^{(a)}, \tag{3.1}$$

where the sum extends over the pair of nearest-neighbor sites $\langle i, j\rangle$, and a runs over $N^2 - 1$ generators of the said representation of the $\mathfrak{su}(N)$ algebra of SU(N). The main result of this work is the rich phase diagram illustrated in Fig. 3.1. Remarkably, and for each considered value of $S = 1/2, 1, 3/2, 2$ just above the threshold value of N above which Néel order disappears, we observe four-, two- and zerofold degenerate disordered states at half-integer, odd and even values of S.

This chapter is organized as follows. In Sec. 3.2 we discuss how we construct the Hamiltonian (3.1), with the spin operators in the desired representation of Fig. 3.2. In Sec. 3.3 we illustrate its actual implementation within a fermionic representation, which can be sampled by means of QMC simulations in the AF approach. In Sec. 3.4 we present and discuss our QMC results for the phase diagram of the model. In Sec. 3.5 we summarize our findings. In *Appendix B.1* we discuss a formula giving the eigenvalue of the quadratic Casimir operator of a representation in terms of its Young tableau. In *Appendix B.2* we prove an upper bound on the eigenvalue of the Casimir operator of the irreducible representations emerging from a tensor product of representations discussed in Sec. 3.2. In *Appendix B.3* we discuss the systematic error in the QMC formulation, arising from the Trotter discretization. In *Appendix B.4* we prove a lower and upper bound for a bond observable used to diagnose the phases.

3.2 General formulation of the Hamiltonian

In the Hamiltonian (3.1), the operators $S_i^{(a)}$ form an irreducible representation of the $\mathfrak{su}(N)$ algebra. This is uniquely specified by its maximum Dynkin weight Λ_α or, alternatively, by a Young tableau, from which the components Λ_{α_k} can be read off as [94]

$$\Lambda_{\alpha_k} = l_k - l_{k+1}, \qquad k = 1, \dots, N-1, \tag{3.2}$$

where l_k is the length of the k-th row of the Young tableau, and one can assume $l_N = 0$ for representations of $\mathfrak{su}(N)$.

Here we consider, on each lattice site, the representation corresponding to a Young tableau illustrated in Fig. 3.2, whose corresponding maximum Dynkin weight is

$$\Lambda_{\alpha_k} = 2S\delta_{k,N/2}. \tag{3.3}$$

The dimension of an irreducible representation can be computed with the hook-length formula, or with Weyl's formula [94]:

$$\dim = \prod_{i<j}^{N} \frac{l_i - l_j + j - i}{j - i}. \tag{3.4}$$

For the present case, we have

$$\dim = \prod_{j=0}^{N/2-1} \frac{(2S + \frac{N}{2} + j)!\, j!}{(2S + j)!\, (\frac{N}{2} + j)!}. \tag{3.5}$$

To realize this representation, we first introduce on each lattice site $2S$ independent irreducible representations. Their tensor product decomposes into different irreducible representations, including, in particular, the one of Fig. 3.2. In a

second step we project the Hilbert space onto that of the desired representation by maximizing the quadratic Casimir operator.

Let $\{T_a\}$, $a = 1, \dots, N^2 - 1$ be a basis of the $\mathfrak{su}(N)$ algebra. We start by introducing on each lattice site i the antisymmetric self-adjoint representation $T_a \to \Gamma(T_a) = \hat{T}_{a,i}$. Its maximum weight in the Dynkin representation is

$$\Lambda_{\alpha_k} = \delta_{k,N/2}, \tag{3.6}$$

which matches Eq. (3.3) for $S = 1/2$. Equivalently, in agreement with Eq. (3.2), this representation corresponds to a Young tableau with one column and $N/2$ boxes.

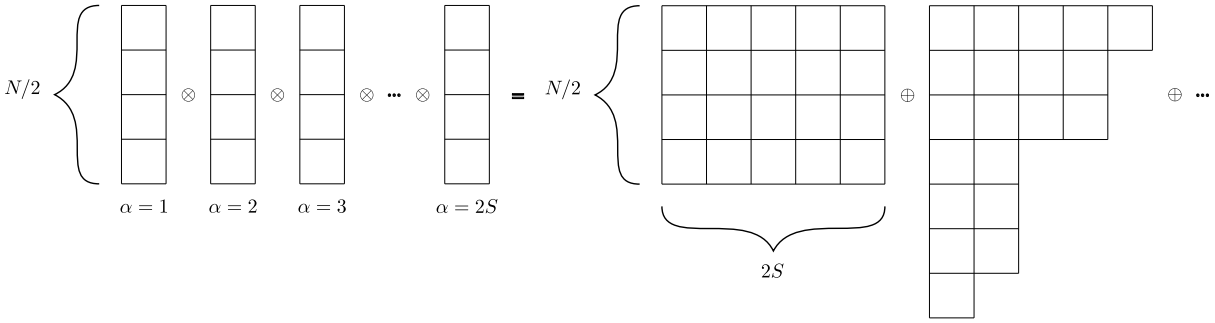


Fig. 3.3: Decomposition of the tensor product of $2S$ antisymmetric self-adjoint representations, whose maximum Dynkin weight is given in Eq. (3.6), into irreducible ones.

Next, we consider, for each lattice site i, $2S$ independent representations $\hat{T}_{a,i,\alpha}$, $\alpha = 1 \dots 2S$. We refer to α as the flavor index. The composite generators, i.e., the generators for the tensor product of the $2S$ representations, define the spin operators appearing in Eq. (3.1) and are given by

$$\hat{S}_i^{(a)} = \sum_{\alpha=1}^{2S} \hat{T}_{a,i,\alpha}, \tag{3.7}$$

Using Eq. (3.7), the interaction term in Eq. (3.1) is written as[1]

$$\begin{aligned} \hat{H}_J &= J \sum_{\langle i\ j\rangle} \sum_{a=1}^{N^2-1} \hat{S}_i^{(a)} \hat{S}_j^{(a)} \\ &= J \sum_{\langle i\ j\rangle} \sum_{a=1}^{N^2-1} \sum_{\alpha,\beta=1}^{2S} \hat{T}_{a,i,\alpha} \hat{T}_{a,j,\beta}. \end{aligned} \tag{3.8}$$

The operators $\hat{S}_i^{(a)}$ in Eq. (3.7) form a reducible representation of $\mathfrak{su}(N)$, which decomposes into several irreducible representations, illustrated in Fig. 3.3. As proven in *App. B.2*, among the resulting representations, the one of Fig. 3.2 exhibits the maximum eigenvalue of the quadratic Casimir operator. To explicitly compute it, we choose a basis $\{T_a\}$ of $\mathfrak{su}(N)$ such that

$$\mathrm{Tr}\{T_a T_b\} = \frac{1}{2}\delta_{ab}. \tag{3.9}$$

With this choice, the structure constants of the algebra are completely antisymmetric and the chosen basis is, up to a trivial normalization, self-dual with respect to the bilinear form (3.9). Thus, given an irreducible representation $\Gamma : \mathfrak{su}(N) \to GL(d, \mathbb{C})$, we define the quadratic Casimir operator as

$$\hat{C}_2 = \sum_a \Gamma(T_a)\Gamma(T_a) \equiv C\mathbb{1}_d, \tag{3.10}$$

where we have used the fact that $C_2 \propto \mathbb{1}_d$ (Schur's Lemma) to introduce the eigenvalue of the Casimir operator C[2].

[1] In Eq. (3.1) and Eq. (3.8) we have implicitly assumed a choice of the basis of $\mathfrak{su}(N)$, such that the interaction term is SU(N)-invariant. This condition is satisfied by the basis choice given below in Eq. (3.9).

[2] We notice that the operator defined in Eq. (3.10) commutes with the algebra only for completely antisymmetric structure constants. For a general choice of the base, one needs to introduce a metric tensor g_{ab} determined by the structure constants and the Casimir operator is defined as $\sum_{ab} g^{ab}\Gamma(T_a)\Gamma(T_b)$ [94]; for completely antisymmetric structure constants $g_{ab} \propto \delta_{ab}$. Also, for the same reason, a normalization is implicit in the definition of Eq. (3.10), discussed in *App. E.1*.

Using Eq. (3.7) in Eq. (3.10), the quadratic Casimir operator on the lattice site i is

$$\hat{C}_{2,\Gamma_i} = \sum_{a=1}^{N^2-1} \sum_{\alpha,\beta=1}^{2S} \hat{T}_{a,i,\alpha} \hat{T}_{a,i,\beta} \tag{3.11}$$

In order to project the Hilbert space to the subspace of the desired representation, we introduce on each site a term in the Hamiltonian which favors the states with the highest Casimir value

$$\begin{aligned} \hat{H}_{\text{Casimir}} &= -J_H \sum_i \hat{C}_{2,\Gamma_i} \\ &= -J_H \sum_i \sum_{a=1}^{N^2-1} \sum_{\alpha,\beta=1}^{2S} \hat{T}_{a,i,\alpha} \hat{T}_{a,i,\beta}, \end{aligned} \tag{3.12}$$

with $J_H > 0$. The term of Eq. (3.12) effectively introduces a ferromagnetic interaction between different flavors, with coupling strength J_H.

The Hamiltonian that we will solve numerically, reads:

$$\hat{H} = \hat{H}_J + \hat{H}_{\text{Casimir}}. \tag{3.13}$$

Importantly, $\left[\hat{H}_J, \hat{H}_{\text{Casimir}}\right] = 0$, such that the projection onto the desired irreducible representation turns out to be very efficient. And since the projection is a local onsite term, we expect it to scale independent from system size.

3.3 QMC formulation

3.3.1 Fermionic representation

As discussed in Sec. 3.2, the Hamiltonian is constructed using as basic building blocks antisymmetric self-adjoint representations, defined by the maximum weight of Eq. (3.6) or, equivalently, by a Young tableau with one column and $N/2$ boxes. The corresponding operators $\hat{T}_{a,i,\alpha}$ entering in Eqs. (3.8) and (3.12) can be realized by introducing, for every lattice site i and for every flavor index α, N nonrelativistic fermions, with creation and annihilation operators $\hat{c}^\dagger_{i,\alpha,\sigma}$, $\hat{c}_{i,\alpha,\sigma}$, $\sigma = 1 \dots N$, and fixing the total charge (i.e., the number of fermions) to half-filling, i.e., to $N/2$. For every i and α, a basis of this Hilbert space is generated by the states

$$\left(\hat{c}^\dagger_{i,\alpha,1}\right)^{n_{i,\alpha,1}} \cdots \left(\hat{c}^\dagger_{i,\alpha,N}\right)^{n_{i,\alpha,N}} |0\rangle, \qquad n_{i,\alpha,\sigma} = 0, 1, \tag{3.14}$$

with the constraint

$$\sum_{\sigma=1}^{N} n_{i,\alpha,\sigma} = \frac{N}{2}, \qquad \forall i, \alpha. \tag{3.15}$$

In this space, the (representation of the) $\mathfrak{su}(N)$ generators are

$$\hat{T}_{a,i,\alpha} = \sum_{\sigma,\sigma'} \hat{c}^\dagger_{i,\alpha,\sigma} (T_a)_{\sigma\sigma'} \hat{c}_{i,\alpha,\sigma'}. \tag{3.16}$$

It is easy to check that the maximum weight in the Dynkin representation agrees with Eq. (3.6), thus providing us the needed building block to simulate the Hamiltonian (3.1).

We study the model by means of finite-temperature AF QMC [8, 9, 10] and projective AF QMC [10, 92, 95]. In this framework, we sample respectively the grand canonical and canonical ensembles at half-filling, and charge fluctuations are generally present. Therefore, we need to additionally impose the constraint of Eq. (3.15). Notice that, unlike available techniques for canonical QMC simulations [96, 97], where the global charge of the system is fixed, here we need to impose half-filling on each lattice site. To this end, we add a repulsive Hubbard U-term on each site i and flavor α:

$$\hat{H}_U = U \sum_i \sum_{\alpha=1}^{2S} \left(\hat{n}_{i,\alpha} - \frac{N}{2}\right)^2, \quad \hat{n}_{i,\alpha} \equiv \sum_{\sigma=1}^{N} \hat{c}^\dagger_{i,\alpha,\sigma} \hat{c}_{i,\alpha,\sigma}. \tag{3.17}$$

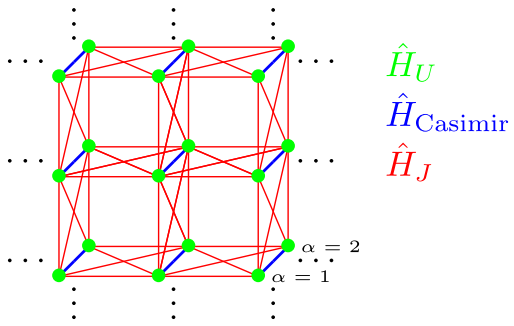


Fig. 3.4: Sketch of the structure of the QMC Hamiltonian for $2S = 2$. $\hat{H}_U$: Hubbard term for freezing out charge degrees of freedom. $\hat{H}_{\text{Casimir}}$: Term for maximizing the eigenvalue of the Casimir operator. $\hat{H}_J$: Antiferromagnetic interaction between elemental spins.

In summary, the Hamiltonian simulated with the AF QMC method is the sum of the interaction term given in Eq. (3.8), the Casimir term [Eq. (3.12)], and the Hubbard term [Eq. (3.17)], with the operators $\{\hat{T}_{a,i,\alpha}\}$ given in Eq. (3.16). Eqs. (3.8) and (3.12) can be further simplified using the following summation identity [98]

$$\sum_{a=1}^{N^2-1} (T_a)_{\sigma\sigma'} (T_a)_{\epsilon\epsilon'} = \frac{1}{2}\left(\delta_{\sigma\epsilon'}\delta_{\sigma'\epsilon} - \frac{1}{N}\delta_{\sigma\sigma'}\delta_{\epsilon\epsilon'}\right), \tag{3.18}$$

which holds for a choice of generators that satisfies Eq. (3.9). Using Eq. (3.18) and collecting the terms in Eqs. (3.8), (3.12) and (3.17), the QMC Hamiltonian is

$$\begin{aligned}\hat{H}_{\text{QMC}} =& \hat{H}_J + \hat{H}_{\text{Casimir}} + \hat{H}_U \\ =& -\frac{J}{4}\sum_{\langle i,j\rangle,\alpha,\beta}\left\{\hat{D}_{(i,\alpha),(j,\beta)}, \hat{D}^\dagger_{(i,\alpha),(j,\beta)}\right\} \\ &+ \frac{J_H}{2}\sum_i\sum_{\alpha>\beta}\left\{\hat{D}_{(i,\alpha),(i,\beta)}, \hat{D}^\dagger_{(i,\alpha),(i,\beta)}\right\} \\ &+ U\sum_{i,\alpha}\left(\hat{n}_{i,\alpha} - \frac{N}{2}\right)^2,\end{aligned} \tag{3.19}$$

where

$$\hat{D}_{(i,\alpha),(j,\beta)} \equiv \sum_\sigma \hat{c}^\dagger_{i,\alpha,\sigma}\hat{c}_{j,\beta,\sigma}, \tag{3.20}$$

$\left\{\hat{A}, \hat{B}\right\} \equiv \hat{A}\hat{B} + \hat{B}\hat{A}$, and $\hat{n}_{i,\alpha}$ as defined in Eq. (3.17). The Hamiltonian now takes the form of the Heisenberg model considered in Ref. [93] and the proof for the absence of sign problem is similar. In Fig. 3.4 we sketch the resulting interactions for the case $S = 1$.

Before proceeding, we would like to comment on the computational cost of the AF QMC algorithm [10] for this model. The total number of orbitals is given by $L^2 2S$ such that matrix operations required to compute, e.g., the single-particle spectral function, scales as $(L^2 2S)^3\beta$, where β is the inverse temperature. It turns out, that, in contrast to the generic Hubbard model with $L^2 2S$ sites, this is not the leading computational cost. The number of Hubbard-Stratonovich fields per imaginary time slice scales as $L^2 S^2$. Using fast updates, refreshing one field involves $(L^2 2S)^2$ floating point operations, such that the total cost of the updating scales as $L^6 S^4 \beta$. Hence large values of S are computationally expensive. In *Appendix B.3*, we show that the computational cost does not explicitly scale with N. We note that this estimate of the computational cost does not take into account auto-correlation times.

3.3.2 Test of projections

As discussed above, the Hamiltonian (3.19) is equivalent to Eq. (3.1) in the limit $J_H \to \infty$, and $U \to \infty$, under which the Hilbert space is projected to the representation of Fig. 3.2. To optimally test the projections, we use the finite-temperature AF QMC method, which evaluates $\langle \hat{O} \rangle = \mathrm{Tr}\left[e^{-\beta \hat{H}} \hat{O}\right] / \mathrm{Tr}\left[e^{-\beta \hat{H}}\right]$, where the trace runs over the grand canonical ensemble.

The interaction term of Eq. (3.8) and the Casimir term of Eq. (3.12) manifestly conserve the charge on each lattice site i. Hence, in the Gibbs density matrix $\exp(-\beta \hat{H})$, the Hubbard term factorizes out, resulting in an effective exponential suppression of the charge fluctuations,

$$\left\langle \left(\hat{n}_{i,\alpha} - N/2\right)^2 \right\rangle \propto e^{-\beta U}, \tag{3.21}$$

independent from system size. The suppression of charge fluctuations is therefore particularly efficient. This is illustrated in Fig. 3.5, where we show $\langle \left(\hat{n}_{i,1} - N/2\right)^2 \rangle$ in a semilogarithmic scale for $N = 2$ and $S = 1/2$ and as a function of βU. Besides the case of a Hamiltonian containing the Hubbard interaction [Eq. (3.17)] only, for which any observable depends only on βU, we consider the presence of the AFM interaction Eq. (3.8) for a lattice of linear size $L = 4$. In the latter case, there is an additional dependence on the inverse temperature β, which we illustrate by considering four values. In line with Eq. (3.21), we observe an exponential suppression of the charge fluctuations as a function of βU. Interestingly, in the interacting case, $\langle \left(\hat{n}_{i,1} - N/2\right)^2 \rangle$ decreases with the temperature for any given value of βU, even for $U = 0$. This implies that the AFM coupling itself suppresses the charge fluctuations.

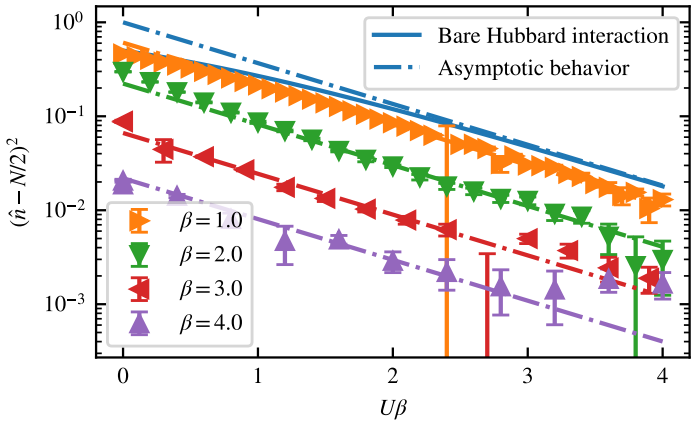


Fig. 3.5: Suppression of the charge fluctuations $\langle \left(\hat{n}_{i,1} - N/2\right)^2 \rangle$ as a function of βU, for $N = 2$ and $S = 1/2$. We consider a Hamiltonian containing the Hubbard term only and the case of a model with an antiferromagnetic interaction [Eq. (3.8)], with coupling constant $J = 1$ on a system size $L = 4$, and for different inverse temperatures β. The charge fluctuations fall off asymptotically as $\exp(-\beta U)$ [Eq. (3.21)].

In *Appendix B.1*, we discuss a formula that gives the value of the Casimir eigenvalue in terms of the Young tableau of the representation [99]. Employing this result, in *App. E.2* we determine, for the representation of Fig. 3.2:

$$C(N, S) = \frac{NS(2S + N)}{4}. \tag{3.22}$$

Furthermore, in *Appendix B.2*, we prove that Eq. (3.22) is the maximum Casimir eigenvalue among the irreducible representations arising from the tensor product of $2S$ self-adjoint antisymmetric representations given in Eq. (3.16), and that there is a finite gap $O(1)$ in the eigenvalues of the quadratic Casimir operators between the maximally symmetric representation of Fig. 3.2 and the other irreducible representations arising from the tensor product. Therefore, the term of Eq. (3.12) effectively selects a single representation, and the projection is efficient.

To control the projection, we compute the expectation value of the quadratic Casimir operator from the QMC simulations and compare it with the expected result of Eq. (3.22). An example of such a projection is shown in Fig. 3.6. In Fig. 3.6(a), we plot the difference between the computed and expected Casimir eigenvalue C, as a function of βJ_H, for different inverse temperatures and in a semilogarithmic scale. The deviation from the expected result is exponentially suppressed in βJ_H, underscoring the effectiveness of the projection. In Fig. 3.6(b), we show, as a function of N, the sampled value of C along with the expected result, and in the inset we plot their difference, which vanishes within error bars.

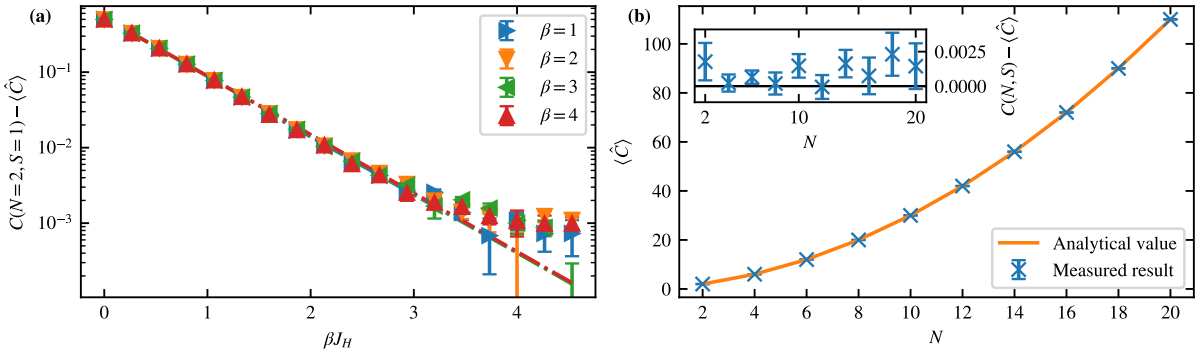


Fig. 3.6: Demonstrating the effectiveness of projection onto the fully symmetric representation for $S = 1$ by comparing the Casimir eigenvalue $C(N, S)$ [Eq. (3.22)] to the sampled one $\langle\hat{C}\rangle$. **(a)** Difference between $C(N = 2, S = 1)$ of the representation $S = 1$, $N = 2$, and $\langle\hat{C}\rangle$, as a function of the effective interaction strength βJ_H [Eq. (3.12)] with $J = 0$, and for four inverse temperatures. Data shown are obtained for a lattice of size $L = 4$, with a Hubbard interaction $\beta U = 6$ [Eq. (3.17)], vanishing nearest-neighbor antiferromagnetic interaction $J = 0$, and a Trotter discretization $\Delta\tau = 0.1$. **(b)** Casimir eigenvalue as a function of N. We compare the predicted value of Eq. (3.22) with the sampled Casimir eigenvalue from QMC simulations of a lattice with size $L = 4$, with a Hubbard interaction $U = 2$ [Eq. (3.17)], antiferromagnetic coupling $J = 1$ [Eq. (3.1)], projection strength $J_H = 1$ [Eq. (3.12)], and a Trotter discretization $\Delta\tau = 0.1$. In the inset we plot the difference between the sampled and expected value.

3.4 Results

3.4.1 Order parameters and phases

We have simulated the Hamiltonian Eq. (3.19) using the ALF package [11, 12], which provides a comprehensive library to program QMC simulations of interacting models of fermions, using the AF algorithm [8, 9, 10]. In particular, we used the projective formulation of the algorithm, which projects a trial wave function $|\Psi_T\rangle$ onto the ground state of the system. Observables are evaluated through

$$\langle\hat{O}\rangle = \frac{\left\langle\Psi_T\middle|e^{-\Theta\hat{H}}\hat{O}e^{-\Theta\hat{H}}\middle|\Psi_T\right\rangle}{\left\langle\Psi_T\middle|e^{-2\Theta\hat{H}}\middle|\Psi_T\right\rangle}, \tag{3.23}$$

with Θ the projection parameter. The algorithm employs a Hubbard-Stratonovich decomposition of the interaction terms. This results in a free fermionic system, where any observable can be computed via the Wick's theorem from the Green's functions. The QMC method consists in a stochastic sampling of the Hubbard-Stratonovich fields. We refer to Ref. [10] for a discussion of the AF QMC method.

As trial wave function $|\Psi_T\rangle$, we used the half-filled ground state of

$$\hat{H}_T = \sum_{\langle i,j\rangle}\sum_{\alpha=1}^{2S}\left(\hat{D}_{(i,\alpha),(j,\alpha)} + \text{H.c.}\right). \tag{3.24}$$

We scaled the projection parameter Θ with linear system size L, usually comparing the results obtained with $\Theta = L/4$ and $\Theta = L/2$, ensuring that they reflect ground state properties. Furthermore, we chose the parameters for suppression of charge fluctuations and projection onto the maximally symmetric representation around $U = 4/\Theta$, $J_H = 4/\theta$, while always checking that charge fluctuations are sufficiently suppressed and $\langle\hat{C}\rangle = C(N, S)$ [cf. Eq. (3.22)].

To detect the realization of different ground states, we have sampled the spin two-point function $S(\boldsymbol{k})$ and the correlations of the dimer operator $D_{ij}(\boldsymbol{k})$ in momentum space, defined as

$$S(\boldsymbol{k}) \equiv \frac{1}{(N^2-1)N_r^2}\sum_{\boldsymbol{r},a} e^{i\boldsymbol{k}\boldsymbol{r}}\left\langle\hat{S}_0^{(a)}\hat{S}_{\boldsymbol{r}}^{(a)}\right\rangle, \tag{3.25}$$

$$D_{ij}(\boldsymbol{k}) \equiv \frac{1}{(N^2-1)N_r^2} \sum_{\boldsymbol{r},a,b} e^{i\boldsymbol{k}\boldsymbol{r}} \cdot \left[\left\langle \left(\hat{S}_0^{(a)} \hat{S}_{0+\boldsymbol{e}_i}^{(a)}\right) \left(\hat{S}_{\boldsymbol{r}}^{(b)} \hat{S}_{\boldsymbol{r}+\boldsymbol{e}_j}^{(b)}\right) \right\rangle - \left\langle \hat{S}_0^{(a)} \hat{S}_{0+\boldsymbol{e}_i}^{(a)} \right\rangle \left\langle \hat{S}_{\boldsymbol{r}}^{(b)} \hat{S}_{\boldsymbol{r}+\boldsymbol{e}_j}^{(b)} \right\rangle \right], \tag{3.26}$$

where $N_r = L^2$ is the number of sites in a lattice of linear size L and $\boldsymbol{e}_i$ is the elementary lattice unit vector on the $i-$th direction. The normalization in Eqs. (3.25) and (3.26) ensure a finite thermodynamic and large-N limit. Using these observables, we can distinguish the Néel state and different dimerized ground states, to be discussed below.

The AFM Néel state exhibits long-range spin-spin correlations at momentum $\boldsymbol{k} = (\pi, \pi)$. Thus, it can be detected by the staggered magnetization m

$$m^2 = S(\boldsymbol{k} = (\pi, \pi)). \tag{3.27}$$

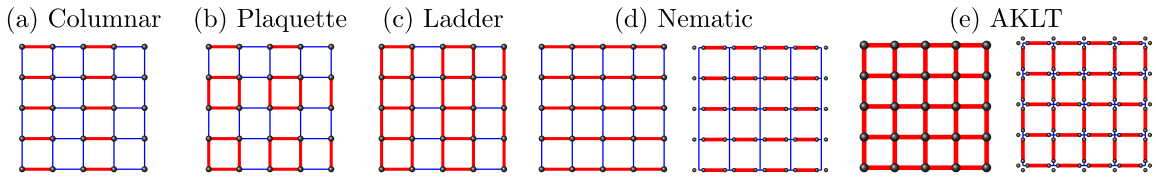


Fig. 3.7: Sketch of possible dimerized ground states. (a)-(c) The VBS states. Each of those states is fourfold degenerate, the corresponding states can be obtained by rotations and translations. (d) A "Haldane nematic" state, an equivalent state is obtained by rotations of 90°. (e) A unique ground state. States (d) and (e) are at best understood within an AKLT construction, in which, very much as done in our calculation, the spin S on each site is constructed by a totally symmetric superposition of $2S$ states that are denoted by bullets around each site on the right-hand side of (d) and (e). These bullets correspond to an irreducible representation of $\mathfrak{su}(N)$ with one column ($S = 1/2$) and $N/2$ rows. In the nematic state, each spin-$1/2$ forms a singlet with the nearest neighbor along the axis of the broken symmetry. The AKLT state is relevant for the $S = 2$ state, where each spin-$1/2$ on a given site can be combined into a singlet with a nearest-neighbor spin-$1/2$ without breaking a lattice symmetry.

The valence bond state (VBS) breaks the lattice rotation and translation symmetries, realizing a fourfold degenerate pattern of strong and weak dimers. This is realized by different sets of bond configurations, illustrated in Figs. 3.7(a)-(c). Beyond the commonly identified columnar order, sketched in Fig. 3.7(a), there are two additional VBS states [Figs. 3.7(b) and (c)]. Notably, all three patterns break the lattice translation symmetry, but only columnar and ladder order break the four-fold rotation symmetry. VBS order can be detected by a suitable order parameter ϕ defined in terms of the dimer correlations

$$\phi^2 \equiv D_{xx}(\boldsymbol{k}) + D_{yy}(\boldsymbol{k}), \qquad \boldsymbol{k} = (\pi, 0). \tag{3.28}$$

We average ϕ over the two equivalent momenta $(\pi, 0)$, and $(0, \pi)$ as to obtain an improved estimator.

For integer values of S, we investigate the possible realization of the "Haldane nematic" Affleck-Kennedy-Lieb-Tasaki (AKLT) phase [18, 19, 20, 21]. This state is twofold degenerate and breaks the rotational symmetry but, unlike the VBS state, does not break translational symmetry. We illustrate it in Fig. 3.7(d). For such a phase we have $\phi = 0$ and a suitable order parameter can be defined as [100]

$$\psi^2 \equiv D_{xx}(\boldsymbol{k}) + D_{yy}(\boldsymbol{k}) - D_{xy}(\boldsymbol{k}) - D_{yx}(\boldsymbol{k}), \qquad \boldsymbol{k} = 0. \tag{3.29}$$

ψ is designed to pick up rotation symmetry breaking in the dimers and therefore does also not vanish for columnar and ladder order. Therefore, ψ distinguishes plaquette with C_4 symmetry from VBS order with broken C_4 symmetry.

Finally, a two-dimensional version of the AKLT phase, with singlets on all bonds as sketched in Fig. 3.7(e), is also possible for $S = 2$. This non-degenerate state does not break any symmetry, therefore all previously defined order parameters vanish. In Table 3.1 we summarize the different orders.

Table 3.1: List of considered ground states with their ordering momenta in reciprocal space and matrix of order parameters defined in Eqs. (3.27), (3.28), (3.29).

Phase	Ordering momenta	C_4 Lattice symmetry preserved	m	ϕ	ψ
Néel	(π, π)	yes	$\neq 0$	0	0
Columnar	$(\pi, 0)$ or $(0, \pi)$	no	0	$\neq 0$	$\neq 0$
Plaquette	$(\pi, 0)$ and $(0, \pi)$	yes	0	$\neq 0$	0
Ladder	$(\pi, 0)$ or $(0, \pi)$	no	0	$\neq 0$	$\neq 0$
Nematic	$(0, 0)$	no	0	0	$\neq 0$
2d AKLT	$(0, 0)$	yes	0	0	0

Same as in Chapter 2, we use the correlation ratio R_O to pinpoint the order:

$$R_O \equiv 1 - \frac{O(\boldsymbol{p} + \delta\boldsymbol{p})}{O(\boldsymbol{p})}, \tag{3.30}$$

where $O(\boldsymbol{p})$ is the two-point function of the order parameter in Fourier space, and $\delta\boldsymbol{p}$ is the minimum nonzero momentum on a finite lattice. On the square lattice, $\delta\boldsymbol{p} = (2\pi/L, 0)$ or $\delta\boldsymbol{p} = (0, 2\pi/L)$; as usual, one can average over the two minimum displacements to obtain an improved estimator. The correlation ratio is closely related to the second-moment finite-size correlation length ξ, which on a square lattice can be defined as [101, 102]

$$\xi = \frac{1}{2\sin(\pi/L)} \sqrt{\frac{O(\boldsymbol{p})}{O(\boldsymbol{p} + \delta\boldsymbol{p})} - 1}. \tag{3.31}$$

In a disordered phase, ξ as defined in Eq. (3.31) converges to the second-moment correlation length for $L \to \infty$, such that $\xi/L \to 0$ and $R_0 \to 0$. In an ordered phase, due to the lack of spontaneous symmetry breaking in any finite size, ξ/L diverges for $L \to \infty$ and, conversely, $R_0 \to 1$. In the vicinity of a critical point, R_O and ξ/L are renormalization-group invariant quantities. Their crossing can be used to locate the onset of the phase transition, rendering them powerful quantities to diagnose the ground-state order and to study phase transitions.

An ergodic QMC simulation averages over all symmetry-breaking states. As a result, we are not able to observe the ordered state directly, but have to refer to correlation functions that do not average out to zero when averaging over all degenerate ground states. Unfortunately, such an approach does not distinguish between the different VBS states illustrated in Figs. 3.7(a)-(c). To obtain additional insights we use the method of a pinning field [103, 104]. In this approach, we explicitly break the symmetry by making one AFM interaction at the origin $J_{\text{pin}} \sum_a \hat{S}_0^{(a)} \hat{S}_{0+\boldsymbol{e}_\text{x}}^{(a)}$ stronger than the other interactions $J \sum_a \hat{S}_{\boldsymbol{r}}^{(a)} \hat{S}_{\boldsymbol{r}+\boldsymbol{e}_i}^{(a)}$. The resulting Hamiltonian reads:

$$\hat{H} = J \sum_{(\boldsymbol{r},i)\neq(0,\text{x})} \sum_a \hat{S}_{\boldsymbol{r}}^{(a)} \hat{S}_{\boldsymbol{r}+\boldsymbol{e}_i}^{(a)} + J_{\text{pin}} \sum_a \hat{S}_0^{(a)} \hat{S}_{0+\boldsymbol{e}_\text{x}}^{(a)}, \tag{3.32}$$

with $J_{\text{pin}} > J$. Therefore, we explicitly choose one of multiple degenerate ground states by pinning the bond $(0, 0 + \boldsymbol{e}_\text{x})$ and the bond observable

$$B_i(\boldsymbol{r}) \equiv \frac{1}{C(N,S)} \left\langle \sum_a \hat{S}_{\boldsymbol{r}}^{(a)} \hat{S}_{\boldsymbol{r}+\boldsymbol{e}_i}^{(a)} \right\rangle \qquad i = \text{x}, \text{y} \tag{3.33}$$

does not vanish as in the unpinned case. We notice that the AFM interactions in the Hamiltonian favors a minimization of $B_i(\boldsymbol{r})$. As proven in *Appendix. B.4*, $B_i(\boldsymbol{r}) \geq -1$ and approaches -1 when the two spins form a singlet.

We have chosen J_{pin} such that the pinned bond satisfies

$$\left\langle \sum_a \hat{S}_0^{(a)} \hat{S}_{0+\boldsymbol{e}_\text{x}}^{(a)} \right\rangle \approx \frac{1}{2} \left(\frac{1}{2N_r} \sum_{(\boldsymbol{r},i),a} \left\langle \hat{S}_{\boldsymbol{r}}^{(a)} \hat{S}_{\boldsymbol{r}+\boldsymbol{e}_i}^{(a)} \right\rangle - C(N,S) \right). \tag{3.34}$$

The right-hand side corresponds to the point halfway between the background and the minimal value of $\left\langle \sum_a \hat{S}_0^{(a)} \hat{S}_{0+\boldsymbol{e}_\text{x}}^{(a)} \right\rangle$. This results in J_{pin} between $1.2J$ and $1.5J$.

At large distances from the pinned bond, one will either be able to explicitly observe the "selected" order through $B_i(\boldsymbol{r})$ (if ϕ or ψ are nonzero), or the order will vanish.

3.4.2 $S = 1/2$

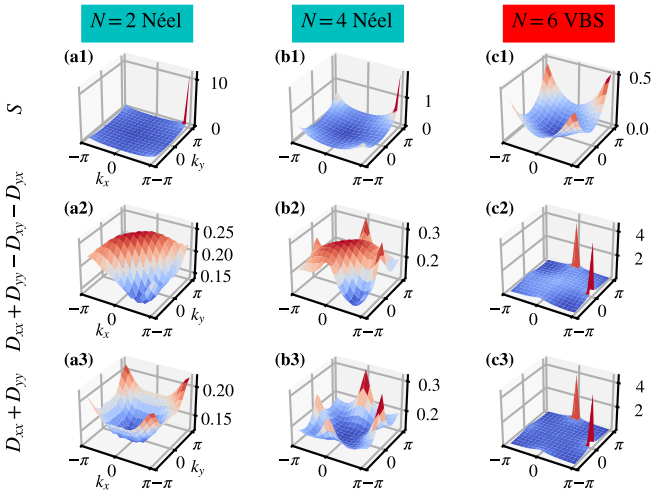


Fig. 3.8: Correlation functions $S(\boldsymbol{k})$ [Eq. (3.25)] and $D_{ij}(\boldsymbol{k})$ [Eq. (3.25)] for the representation of Fig. 3.2 with $S = 1/2$ and different values of N.

We first study the representations of Fig. 3.2 with $S = 1/2$. In Fig. 3.8, we show the structure factor $S(\boldsymbol{k})$ [Eq. (3.25)], and the two combinations of dimer correlations $D_{ij}(\boldsymbol{k})$ appearing in the definitions of the order parameter ϕ [Eq. (3.28)] and ψ [Eq. (3.29)]. By considering three values of N, we analyze the evolution of the order parameters across the transition between the Néel and VBS order. For $N = 2$, we realize a standard $S = 1/2$ Heisenberg model on the square lattice, which displays a Néel order in the ground state. As expected, $S(\boldsymbol{k})$ shows a strong peak at (π, π); in comparison, the other order parameters are suppressed. Upon increasing N to $N = 4$, we observe the emergence of peaks at $(\pi, 0)$ and $(0, \pi)$ for the other order parameters. As shown below using the correlation ratios, though close to the phase transition to the VBS state, the ground state is still Néel ordered. For $N = 6$ we observe a strong peak at $(\pi, 0)$ and $(0, \pi)$ for the $D_{xx}(\boldsymbol{k}) + D_{yy}(\boldsymbol{k})$ order parameters, indicating the realization of the VBS phase.

To obtain a reliable determination of the ground state, we study the correlation ratios of the three order parameters discussed in Sec. 3.4.1. In Fig. 3.9, we show the three correlation ratios R_m, R_ϕ, and R_ψ as a function of N, for lattice sizes $L = 4, 8, 12, 16$. The crossing plot of R_m indicates the disappearance of Néel order at $N \approx 4$. At the same time, the curves of R_ϕ and R_ψ exhibit a crossing for values of $N > 4$; in particular, for $N = 4$ both R_ϕ and R_ψ decrease with the lattice size, indicating that both VBS and nematic order are short-ranged for $N = 4$. Therefore, as anticipated above, for $N = 4$ the ground state is still a Néel order. This confirms the result of Ref. [105, 106, 107]. At $N = 6$, R_m decreases with L, while R_ϕ and R_ψ increases in L, for $L \geq 6$; the $L = 4$ data set is dominated by finite-size effects. Accordingly, for $N = 6$ the ground state realizes VBS order.

These observations are confirmed by the real-space plot of the bond intensity shown in Fig. 3.10, as obtained with the pinning-field method. For $N = 4$, in the vicinity of the pinned bond we observe a pattern reminiscent of the VBS order of Fig. 3.7. However, at larger distances the modulation of bond intensity quickly decays, confirming that VBS order is actually short ranged. For $N = 6$, we observe a very clear pattern of strong and weak bonds that realize the VBS order.

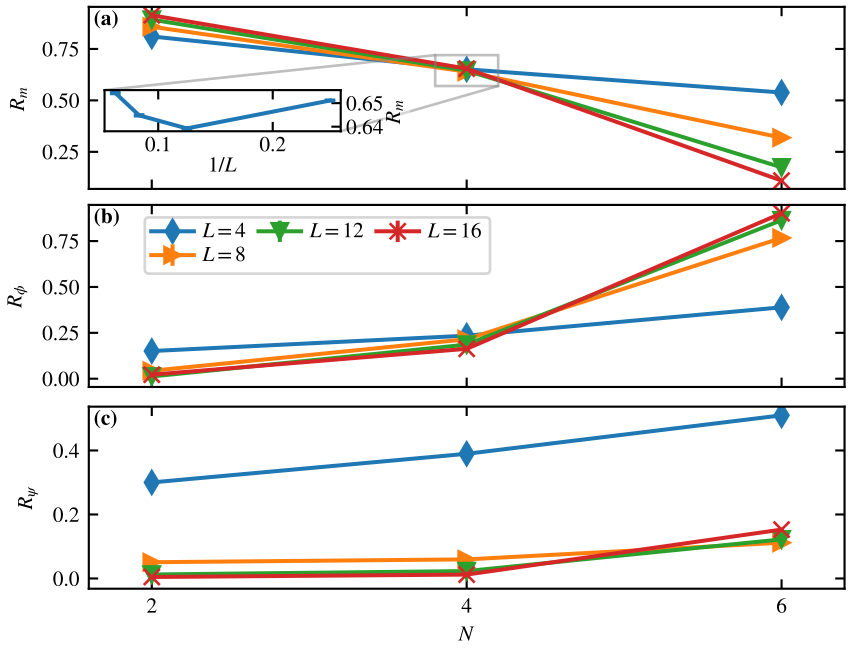


Fig. 3.9: Correlation ratios of the staggered magnetization m [Eq. (3.27)], ϕ [Eq. (3.28)] and ψ [Eq. (3.29)] order parameters for $S = 1/2$, as a function of N, and for lattice sizes $L = 4 - 16$.

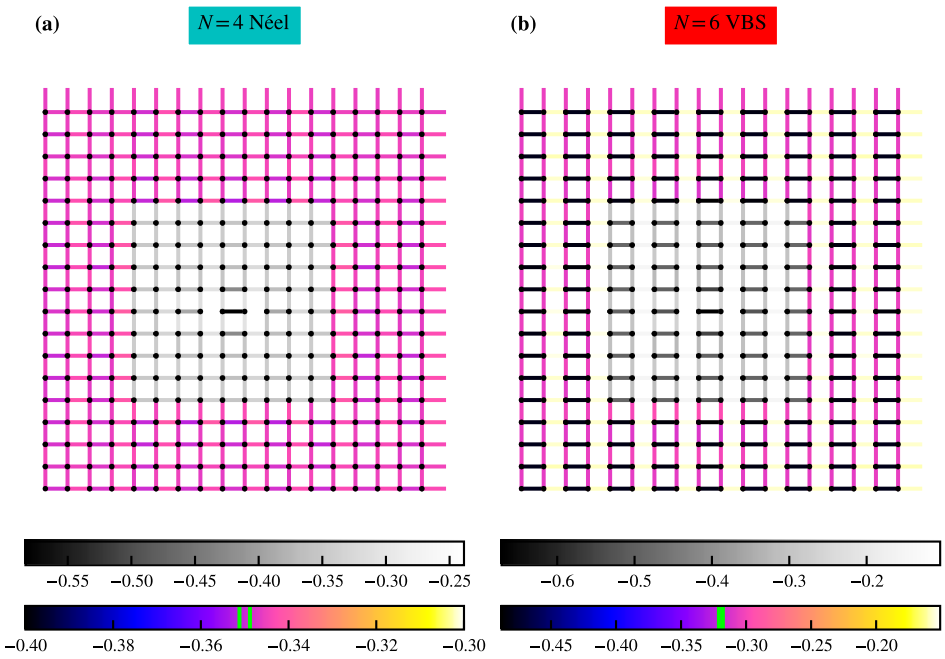


Fig. 3.10: Real-space value of bonds $B_i(\boldsymbol{r})$ [Eq. (3.33)], as measured after pinning the central bond, for $S = 1/2$ and lattice size $L = 18$. Due to the observed different variations in the bond strength, in order to better highlight the patterns of bond correlations we have used different color scales for the region close and far from the pinned bond. The biggest error of the outer bonds is indicated by two red lines on the color scale. We have symmetrized the results with regards to inversions $y \to -y$, $x \to -x$ around the pinned bond.

3.4.3 $S = 1$

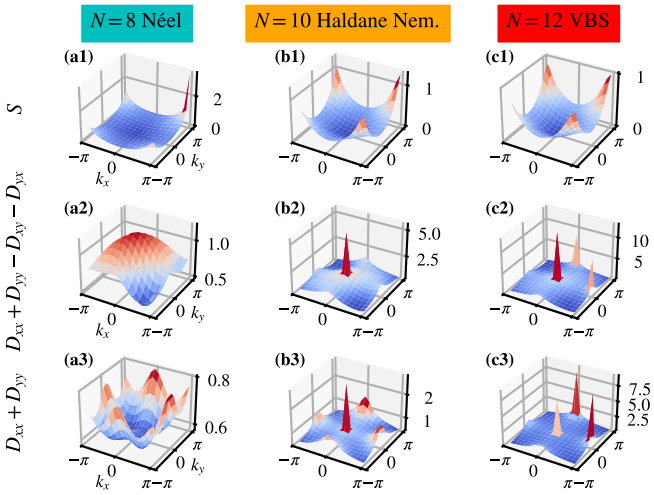


Fig. 3.11: Same as Fig. 3.8 for $S = 1$.

In studying the representations with $S = 1$, we proceed analogously to the $S = 1/2$ case discussed in Sec. 3.4.2. In Fig. 3.11, we show the order parameters in momentum space for the case $S = 1$, and three representative values of N. For $N = 8$, a clear peak of the spin structure factor at (π, π), along with a comparatively smoother momentum dependence of the other order parameters, indicate the presence of Néel order. At $N = 10$, we observe instead the emergence of a clear signal of the nematic order parameter at zero momentum. The VBS order parameter exhibits a similar peak at zero momentum, whose signal predominantly arises from the nematic order parameter, while at momentum $(\pi, 0)$ a subdominant peak is observed. The Néel order parameter instead does not show a predominant signal at (π, π), but rather equally large values at the corners of the Brillouin zone. These behaviors suggest the onset of the two-fold degenerate nematic order. For $N = 12$ a clear signal at $(\pi, 0)$ momentum appears in the VBS order parameter. Along with the sharp zero-momentum value of the nematic order parameter, these findings suggest the realization of VBS order for $N = 12$.

As we did for the $S = 1/2$ case, the above qualitative observations on the momentum dependence of the various order parameters can be put on firm ground by examining the correlation ratios shown in Fig. 3.12. The magnetic correlation ratio R_m displays a crossing at about $N \approx 8$, such that for $N > 8$, R_m decreases with the lattice size. On the other hand, at $N = 8$ both R_ϕ and R_ψ decrease on increasing L, implying that both VBS and nematic order are short ranged. Therefore, one can conclude that for $N = 8$ the ground state is antiferromagnetically ordered. The behavior of R_ϕ shows rather important finite-size corrections. In fact, while curves for $L \leq 10$ cross for $8 < N < 10$, the crossing point quickly increases with L, such that for $L \geq 12$ a crossing is found for $10 < N < 12$. In particular, $N = 10$ has a nonmonotonic behavior, increasing in L for $L \leq 10$, and decreasing for $L \geq 12$. This observation supports the presence of a significant, but still short-ranged, VBS order, which is responsible for important finite-size corrections. The nematic correlation ratio R_ψ shows a crossing between $N = 8$ and $N = 10$. Also, here we observe a clear drift in the crossing of R_ψ, although the situation for $N = 10$ is rather clear and indicates long-range order in ψ. In view of these observations, and referring to Table 3.1, we conclude that a nematic ground state is realized for $N = 10$, while for $N \geq 12$ the ground-state is VBS ordered.

These conclusions are nicely confirmed by the real-space plots of the bond strength obtained with the pinning-field method and shown in Fig. 3.13. For $N = 10$ we clearly observe the formation of a twofold degenerate stripe-like structure, signalling the presence of the nematic order. For $N = 12$ instead a VBS order is found. Interestingly, while for $S = 1/2$ the VBS order found at $N = 6$ (Fig. 3.10) resembles the ladder order illustrated in Fig. 3.7, for $S = 1$ and $N = 12$ the VBS pattern shown in Fig. 3.13 rather suggests the plaquette order of Fig. 3.7.

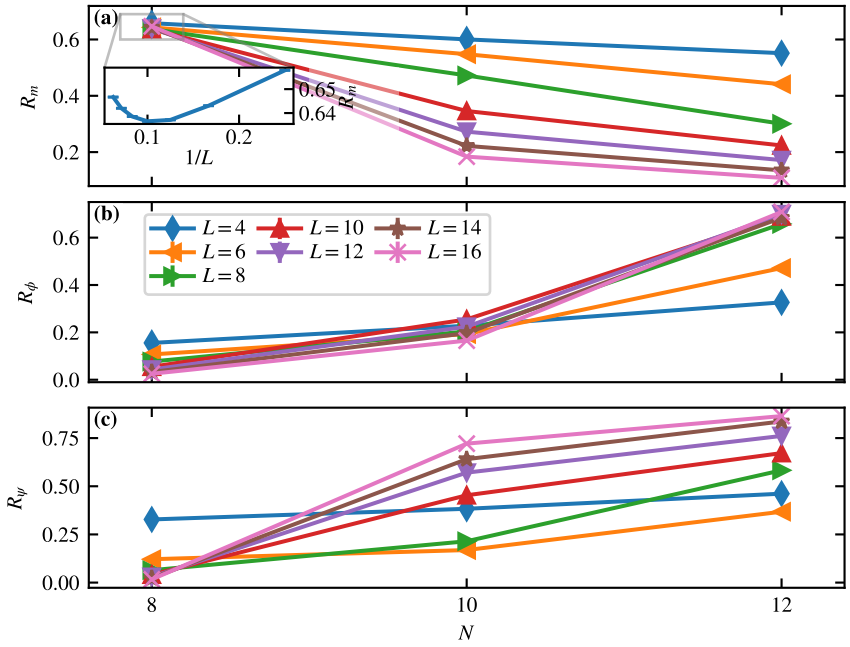


Fig. 3.12: Same as Fig. 3.9 for $S = 1$.

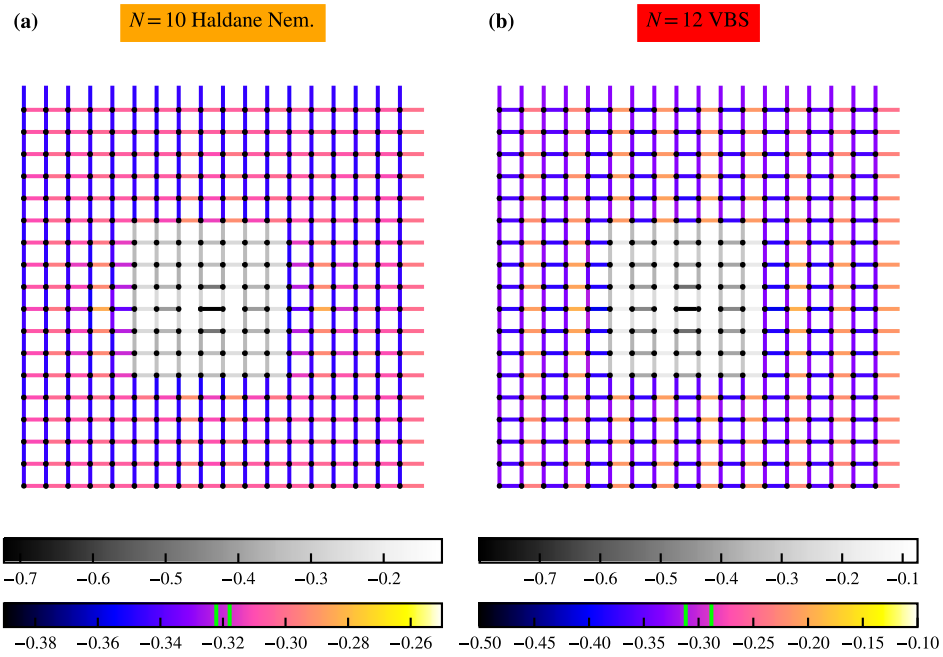


Fig. 3.13: Same as Fig. 3.10 for $S = 1$.

3.4.4 $S = 3/2$

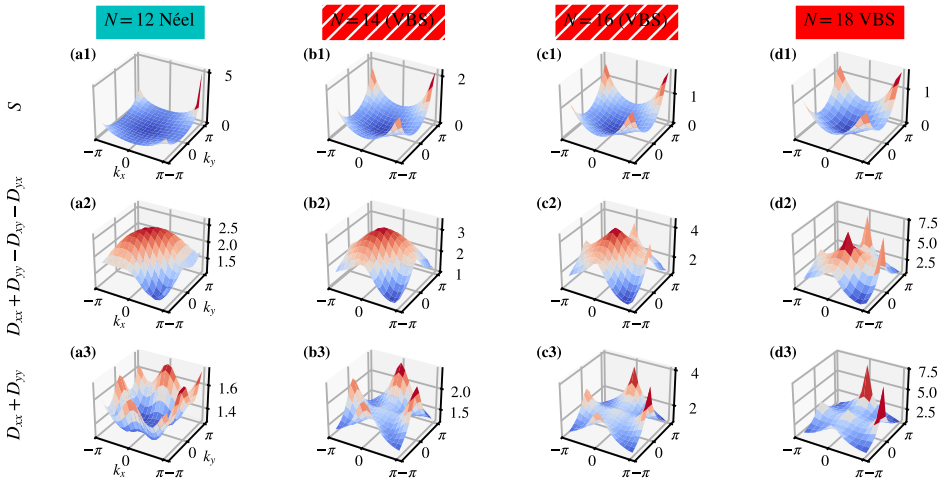


Fig. 3.14: Same as Fig. 3.8 for $S = 3/2$.

In Fig. 3.14, we show the order parameters in momentum space, and $N = 12 - 18$. Analogous to the cases analyzed in the previous sections, for $N = 12$ we find a signal of Néel order. In the region $14 \leq N \leq 16$, QMC data do not allow us to unambiguously single out the ground state. Upon increasing N, the peak at (π, π) in $S(\boldsymbol{k})$ slowly decreases in magnitude. At the same time, we observe the appearance of a maximum in the nematic order parameter at zero momentum, and in the VBS order parameter for $(\pi, 0)$ and $(0, \pi)$ momenta. Eventually, for $N = 18$ the momentum structure of the order parameters more clearly favors the realization of VBS order. The observed behavior suggests a comparatively broad critical region around $14 \leq N \leq 18$.

In an attempt to better understand the ground-state diagram for $S = 3/2$, as for the other values of S we have analyzed the correlation ratios, shown in Fig. 3.15. Due to the increased computational costs, we restricted the simulations for the larger lattice sizes $L \geq 12$ to the more involved cases $12 \leq N \leq 16$. For smaller lattice sizes $L \leq 10$, the curves for R_m appear to cross at a value of N very close, but smaller than $N < 12$. We observe, however, some drift towards larger values of N in the crossings. Furthermore, as shown in the inset of Fig. 3.15, R_m at $N = 12$ exhibits an upwards trend for $L > 10$. Together with the observed slow decrease of R_ϕ and R_ψ in L for $N = 12$, this implies Néel order for $N = 12$.

For $N \geq 14$, the $L-$dependence of R_m clearly rules out Néel order. On the other hand, R_ϕ slowly decreases with L for $N = 14$, and for $N = 16$ it grows slightly up to $L = 8$. In both cases, QMC data for $L \geq 8$ are indistinguishable within error bars. A similar flattening of QMC data is found in R_ψ for $N = 14, 16$. This behavior does not allow us to draw firm conclusions on the nature of the ground state for $N = 14, 16$. Since a Néel state can be ruled out, a reasonable hypothesis is the realization of a VBS state, however, with a weak order parameter.

For $N > 16$, both R_ϕ and R_ψ show a crossing close to $N = 18$. Furthermore, we observe a monotonic growth of R_ϕ in L for $N = 18$. This leads us to conclude a VBS order for $N = 18$.

Finally, we have studied the pattern of bond strength in real space with the pinning-field method. The results for $N = 14, 16, 18$ are reported in Fig. 3.16. For $N = 14, 16$, despite some signs of dimerization, we do not observe a clear VBS pattern. In line with the previous analysis, for $N = 18$ we find a bond dimerization which confirms a VBS ground state.

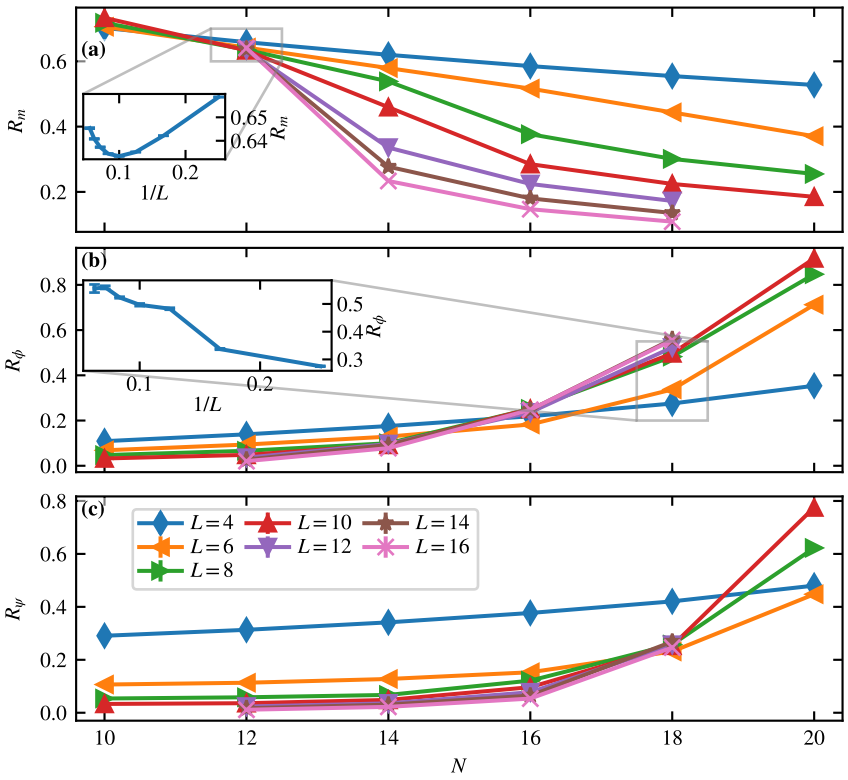


Fig. 3.15: Same as Fig. 3.9 for $S = 3/2$.

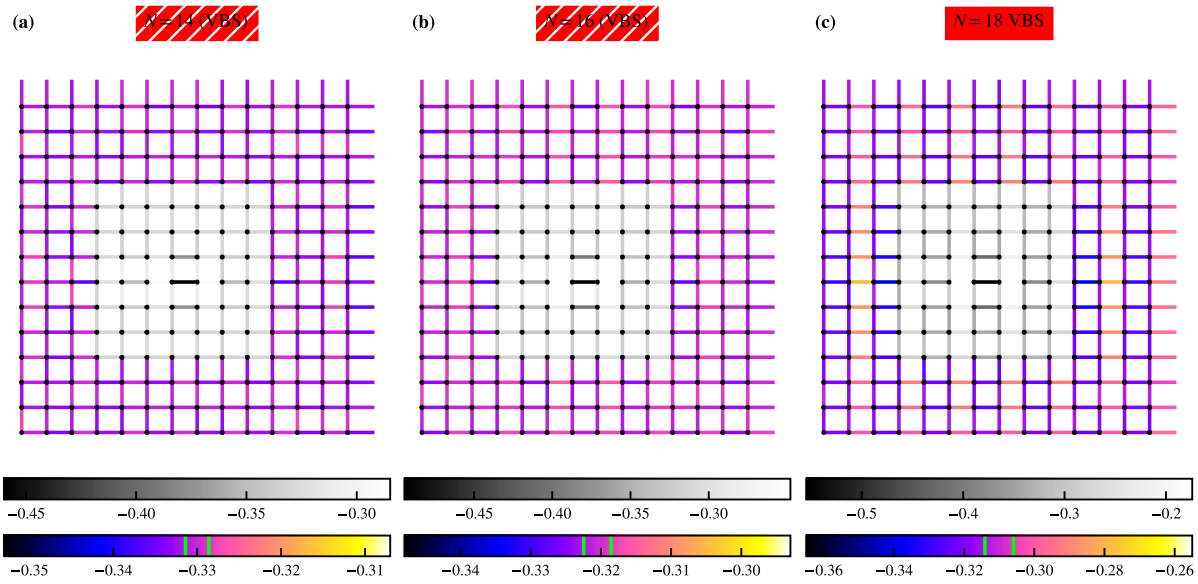


Fig. 3.16: Same as Fig. 3.10 for $S = 3/2$ and lattice size $L = 14$ $(N = 14, 16)$, $L = 12$ $(N = 18)$.

3.4.5 $S = 2$

As for previous values of S, we begin our investigation for $S = 2$ with momentum space plots of correlation functions shown in Fig. 3.17. At $N = 16$, the spin structure factor shows a sharp peak at (π, π), indicating long-range AFM order. For bigger values of N the peak weakens and broadens, suggesting short-range AFM order. The correlations for nematic and VBS order show only very broad maxima for the range $N \in [16, 22]$, implying the absence of both of these orders. The correlation ratios plotted in Fig. 3.18 support these qualitative observations. The inset in Fig. 3.18(a) shows that while R_m decreases at $N = 16$ from $L = 4$ to $L = 12$, the trend is reversed on bigger lattices and R_m increases from $L = 12$ to $L = 16$. This indicates that $N = 16$ has a Néel ground state which is close to a competing order. Both R_ϕ and R_ψ decrease with increasing system size in the investigated range $N \in [16, 24]$. As per Table 3.1, this leaves a two-dimensional AKLT order as a ground-state candidate for $N \in [18, 24]$. With the AKLT construction corresponding to the right-hand side of Fig. 3.7(e), we understand that each boundary site hosts an $\mathfrak{su}(n)$ representation corresponding to one column ($S = 1/2$) and $N/2$ rows. Hence the boundary defines a one-dimensional chain in the aforementioned representation. It is known that for $N \geq 4$ this chain dimerizes [105, 106, 108, 109].

To further investigate this possibility, we simulate the model on a lattice with periodic boundary conditions in the x direction and open boundary conditions along y, corresponding to a cylinder geometry. Fig. 3.19 shows the results for $S = 2, N = 18$ in a pinning-field approach with pinned bonds at the edge and in the bulk, respectively. The induced dimerization pattern propagates on the edge much further than in the bulk, supporting the presence of an AKLT phase with boundary corresponding to an $S = 1/2$ SU(N) chain in the totally antisymmetric self-adjoint representation.

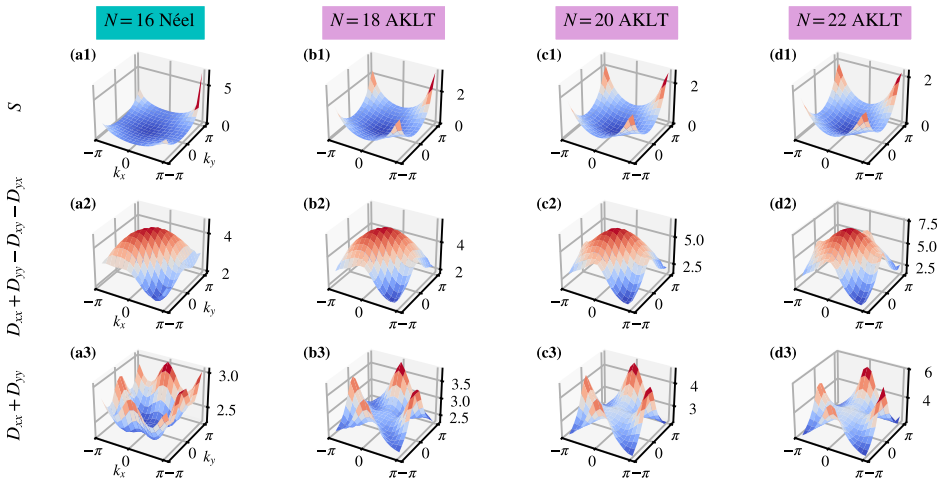


Fig. 3.17: Same as Fig. 3.8 for $S = 2$

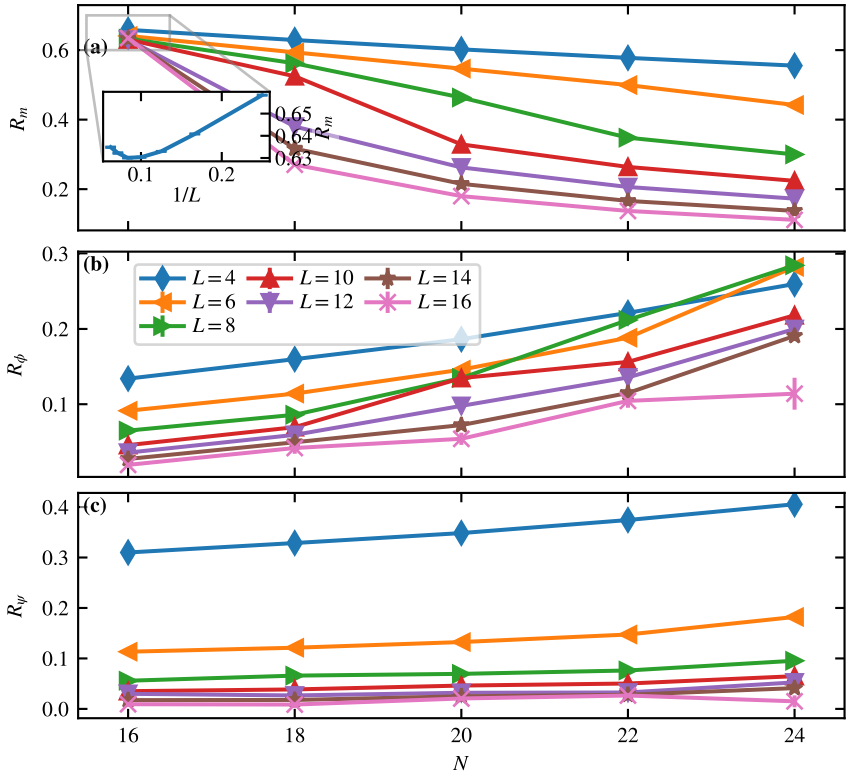


Fig. 3.18: Same as Fig. 3.9 for $S = 2$

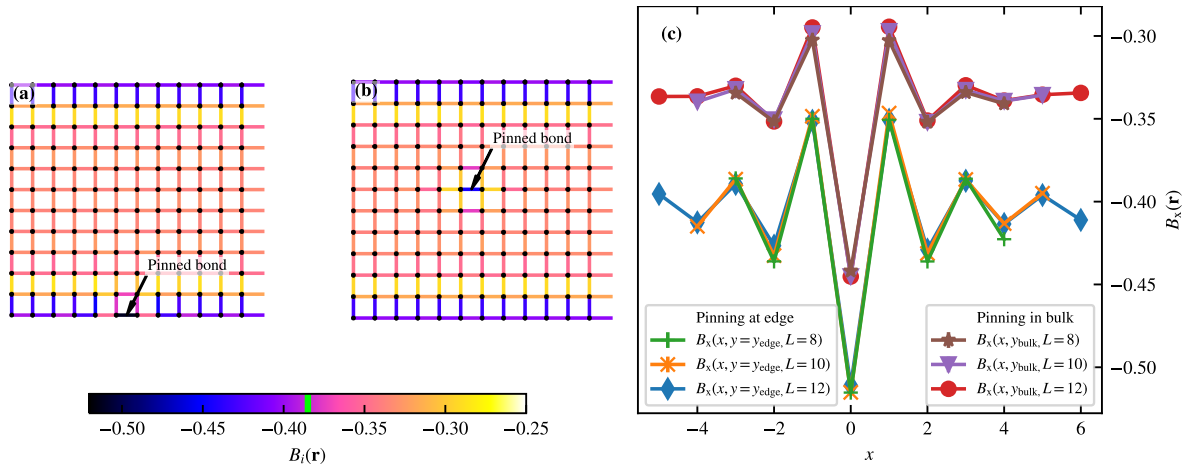


Fig. 3.19: Real-space value of bonds $B_i(\boldsymbol{r})$ for $S = 2$, $N = 18$, lattice size $L \in \{8, 10, 12\}$ and open boundary conditions in y direction. Comparison between pinning a bond at the edge **(a)** and a bond in the bulk **(b)**. **(c)** $B_x(\boldsymbol{r})$ on horizontal lines through the pinned bonds.

3.5 Summary

We have studied the ground-state phase diagram of an SU(N) AFM model on the square lattice, with irreducible representations of $\mathfrak{su}(N)$ illustrated in Fig. 3.2 and characterized by a Young tableau consisting of $2S$ columns and $N/2$ rows. For even values of N we have presented negative sign free QMC data that results in the rich phase diagram of Fig. 3.1. In line with field-theoretical studies [18, 20, 21], for any value of the generalized spin S, we found Néel order at small values of N, and a dimerized VBS state for large N. The disordered states proximate to the melting of the Néel state can be naturally understood in terms of condensation of monopoles. These states turn out to be located along the $N = 8S + 2$ in the S versus N phase diagram. At $S = 1$ we observe nematic AKLT [19] states where C_4 symmetry is spontaneously reduced to C_2, with the emergence of spin-1 chains along one lattice direction. At $S = 2$, the AKLT construction provides an understanding of the non-degenerate state. In fact, this construction can be generalized to any spin $S = Z/2$ system on a lattice with coordination number Z; an example for the honeycomb lattice is given in Refs. [110, 111]. In our specific case, the edge state corresponds to an SU(N) spin system in an irreducible representation specified by a Young tableau with one column ($S = 1/2$) and $N/2$ rows. At $N \geq 4$, such a state is known to dimerize [106]. Our simulations with open boundary conditions support this picture. For half-integer values of S, a fourfold degenerate VBS state emerges. The detailed nature of the dimerization was studied using a pinning-field approach [103].

While the resulting phase diagram reproduced in Fig. 3.1 will serve as a benchmark for future studies, our findings points to future avenues of research. It would be very interesting to study in detail the quantum phase transitions between the various states. Although the present setup allows us to consider integer values of N only, it may be possible to investigate the phase transitions with a suitably defined designer Hamiltonian containing, e.g., some interactions that favor a specific phase, so as to be able to interpolate between them. The topological arguments that lead to the observed phase diagram carry over to other representation of $\mathfrak{su}(N)$, such that further calculations with alternative methods such as stochastic series expansion [112] are certainly desirable.

CHAPTER

FOUR

PYALF DOCUMENTATION

The two previous chapters covered my research projects, which used the quantum Monte Carlo (QMC) package ALF [11, 12]. It is a powerful tool for simulating a broad set of fermionic systems, but since it's written in Fortran, it is not very dynamic and can be a bit daunting for new users.

In the process of performing my research projects, I developed an ALF workflow using Python scripts, which has resulted in the pyALF package. 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://git.physik.uni-wuerzburg.de/ALF.

Section 4.1 describes the prerequisites of pyALF and how to set things up to be able to use it in a productive manner.

Section 4.2 displays the features of pyALF and how to use them on small examples.

For a reference on pyALF's features, see Section 4.3.

4.1 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.

4.1.1 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[1]. Other supported and tested compiler frameworks are from the Intel® oneAPI Toolkits[2] and the NVIDIA HPC SDK[3]. The latter is denoted as `PGI` in ALF.

4.1.2 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[4].

```
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] https://gcc.gnu.org/
[2] https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html
[3] https://developer.nvidia.com/nvidia-hpc-sdk-downloads
[4] https://pip.pypa.io/en/stable/

4.1.2.1 Development installation

If you want to develop pyALF, you can clone the repository and install it in development mode[5], 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[6] or a conda environment[7]. The following example shows how to install pyALF in development mode using venv.

```
git clone https://git.physik.uni-wuerzburg.de/ALF/pyALF.git
cd pyALF
python -m venv .venv
source .venv/bin/activate

pip install --editable .
```

4.1.3 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[8] 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://git.physik.uni-wuerzburg.de/ALF/ALF.git
```

This will create a folder called `ALF` in the current working directory of the terminal and download the repository there[9].

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`.

4.1.4 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.

[5] https://setuptools.pypa.io/en/latest/userguide/development_mode.html
[6] https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/
[7] https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html
[8] https://git.physik.uni-wuerzburg.de/ALF/ALF
[9] 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`.

4.1.5 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[10]) in one file. pyALF includes example notebooks, online available from here[11], or by cloning the pyALF repository[12].

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[13]):

```
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[14], a versatile and extendable source-code editor.

4.1.6 Ready-to-use container image

For a ready-to-use environment, one can use the Docker image alfcollaboration/jupyter-pyalf-full[15], which has the above mentioned dependencies, ALF and pyALF installed. With a suitable container runtime e.g. Docker[16] or Podman[17], it can be used to run ALF and pyALF without any further setup. It is derived from the Jupyter Docker Stacks, therefore this documentation[18] 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
```

- The -p[19] 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[20] 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[21] flag instructs Docker to automatically remove the container after it exits, avoiding cluttering up the system with unused containers.
- The -i[22] and -t[23] 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'
```

[10] https://www.markdownguide.org/
[11] https://git.physik.uni-wuerzburg.de/ALF/pyALF/-/tree/master/Notebooks
[12] https://git.physik.uni-wuerzburg.de/ALF/pyALF
[13] https://jupyter.org/install
[14] https://code.visualstudio.com/
[15] https://hub.docker.com/r/alfcollaboration/jupyter-pyalf-full
[16] https://www.docker.com/
[17] https://podman.io/
[18] https://jupyter-docker-stacks.readthedocs.io
[19] https://docs.docker.com/reference/cli/docker/container/run/#publish
[20] https://docs.docker.com/reference/cli/docker/container/run/#volume
[21] https://docs.docker.com/reference/cli/docker/container/run/#rm
[22] https://docs.docker.com/reference/cli/docker/container/run/#interactive
[23] https://docs.docker.com/reference/cli/docker/container/run/#tty

4.1.7 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.

4.1.7.1 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://git.physik.uni-wuerzburg.de:443/ALF/ALF.git
git clone git@git.physik.uni-wuerzburg.de:ALF/ALF.git
```

But on some systems with very restrictive firewalls, this approach might not work. This is where the ssh option -R[24] 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 `git.physik.uni-wuerzburg.de`, the following commands can be used, connecting to port 443 or 22, for the HTTPS or SSH protocol, respectively.

```
ssh -R <PortNum>:git.physik.uni-wuerzburg.de:443 <username>@<servername>
ssh -R <PortNum>:git.physik.uni-wuerzburg.de: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 [113]. And `<username>@<servername>` is the usual SSH address. Alternatively to the command line option `-R`, the SSH config file option RemoteForward[25] 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/ALF.git
git clone ssh://git@localhost:<PortNum>/ALF/ALF.git
```

The HTTPS version needs the option `-c http.sslVerify=false` because the SSL certificate for `git.physik.uni-wuerzburg.de` does not apply to `localhost`.

One can omit the host value in the `-R` option (in the example above `git.physik.uni-wuerzburg.de: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[26] 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
```

[24] https://man.openbsd.org/ssh#R
[25] https://man.openbsd.org/ssh_config#RemoteForward
[26] https://github.com/Anorov/PySocks/blob/master/socks.py

4.1.7.2 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[27] 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 trough the address `http://localhost:<local_port_number>:/lab?token=<token>`.

4.1.7.3 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

---

4.2 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 4.1. 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 4.3.

> **Tip**
>
> The Python builtin `help()`[28] is very useful for getting information on an object. Try e.g. `help(Simulation)` after importing `Simulation` from `py_alf`.

[27] https://man.openbsd.org/ssh#L
[28] https://docs.python.org/3/library/functions.html#help

4.2.1 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[29], 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.
```

[29] https://git.physik.uni-wuerzburg.de/ALF

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

```
sim.run()
```

```
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/usage/ALF_data/Hubbard_
 ↪Square ###
/home/jonas/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/usage/ALF_data/Hubbard_Square
No orbital locations saved.
```

```
obs
```

```
                                                  continuous  ham_chem  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...           0       0.0

                                                  ham_t  ham_t2  ham_tperp  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...   1.0     1.0        1.0

                                                  ham_u  ham_u2  mz  l1  l2  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...   4.0     4.0   1   6   6

                                                  ...  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  ...
```

(continues on next page)

(continued from previous page)

```
↪      SpinXY_tauK_err  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  [[0.22628722504548163, 0.
↪376854598540396, 0.01...

↪          SpinXY_tauR  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  [[0.0575988003757765, -0.
↪10378742200169441, 0....

↪      SpinXY_tauR_err  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  [[0.012488346667988096, 0.
↪036871289378231954, ...

↪   SpinXY_tau_lattice  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  {'L1': [6.0, 0.0], 'L2': [0.
↪0, 6.0], 'a1': [1....

↪           SpinZ_tauK  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  [[0.9040036362033993, 0.
↪5628191896020671, 0.61...

↪       SpinZ_tauK_err  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  [[0.14947867503118553, 0.
↪06006336638595224, 0....

↪           SpinZ_tauR  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  [[0.10134420531394882, -0.
↪11445552391592617, 0...

↪       SpinZ_tauR_err  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  [[0.06273438448505807, 0.
↪0563907804306374, 0.0...

↪    SpinZ_tau_lattice  \
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  {'L1': [6.0, 0.0], 'L2': [0.
↪0, 6.0], 'a1': [1....

↪              lattice
/home/jonas/dissertation/jb/chap4_pyalf/usage/A...  {'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
↪']]
```

```
Ener_scal0           -29.821914
Ener_scal0_err          0.13032
Ener_scal_sign              1.0
Ener_scal_sign_err          0.0
```

(continues on next page)

(continued from previous page)

```
Name: /home/jonas/dissertation/jb/chap4_pyalf/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[30].

- 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/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/usage/ALF_data/Hubbard_
 ↪Square ###
/home/jonas/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/usage/ALF_data/Hubbard_Square,␣
 ↪dtype: object
```

[30] https://git.physik.uni-wuerzburg.de/ALF/ALF/-/jobs/artifacts/master/raw/Documentation/doc.pdf?job=create_doc

```
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 0.13032001865023543
to 0.13680286324412563
```

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

4.2.2 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 4.2.1.

We start with some imports:

```
from pprint import pprint  # Pretty print
from py_alf import ALF_source, Simulation  # Interface with ALF
```

4.2.2.1 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://git.physik.uni-wuerzburg.de/ALF/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'))
```

```
OrderedDict([('VAR_lattice',
              {'L1': {'comment': 'Length in direction a_1',
                      'defined_in_base': False,
                      'value': 6},
               'L2': {'comment': 'Length in direction a_2',
                      'defined_in_base': False,
                      'value': 6},
               'Lattice_type': {'comment': '',
                                'defined_in_base': False,
                                'value': 'Square'},
               'Model': {'comment': 'Value not relevant',
                         'defined_in_base': False,
                         'value': 'Hubbard'}}),
             ('VAR_Model_Generic',
              {'Beta': {'comment': 'Inverse temperature',
                        'defined_in_base': False,
                        'value': 5.0},
               'Bulk': {'comment': 'Twist as a vector potential (.T.), or at '
                                   'the boundary (.F.)',
                        'defined_in_base': False,
                        'value': True},
               'Checkerboard': {'comment': 'Whether checkerboard decomposition '
                                           'is used',
                                'defined_in_base': False,
                                'value': True},
               'Dtau': {'comment': 'Thereby Ltrot=Beta/dtau',
                        'defined_in_base': False,
                        'value': 0.1},
               'N_FL': {'comment': 'Number of flavors',
                        'defined_in_base': True,
                        'value': 1},
               'N_Phi': {'comment': 'Total number of flux quanta traversing '
                                    'the lattice',
                         'defined_in_base': False,
                         'value': 0},
               'N_SUN': {'comment': 'Number of colors',
                         'defined_in_base': True,
                         'value': 2},
               'Phi_X': {'comment': 'Twist along the L_1 direction, in units '
                                    'of the flux quanta',
                         'defined_in_base': False,
                         'value': 0.0},
               'Phi_Y': {'comment': 'Twist along the L_2 direction, in units '
                                    'of the flux quanta',
                         'defined_in_base': False,
                         'value': 0.0},
               'Projector': {'comment': 'Whether the projective algorithm is '
                                        'used',
                             'defined_in_base': True,
                             'value': False},
               'Symm': {'comment': 'Whether symmetrization takes place',
                        'defined_in_base': True,
                        'value': True},
               'Theta': {'comment': 'Projection parameter',
                         'defined_in_base': False,
                         'value': 10.0}}),
             ('VAR_Hubbard',
```

(continues on next page)

(continued from previous page)

```
 {'Continuous': {'comment': 'Uses (T: continuous; F: discrete) HS '
                            'transformation',
                 'defined_in_base': False,
                 'value': False},
  'Ham_U': {'comment': 'Hubbard interaction',
            'defined_in_base': False,
            'value': 4.0},
  'Ham_U2': {'comment': 'For bilayer systems',
             'defined_in_base': False,
             'value': 4.0},
  'Ham_chem': {'comment': 'Chemical potential',
               'defined_in_base': False,
               'value': 0.0},
  'Mz': {'comment': 'When true, sets the M_z-Hubbard model: Nf=2, '
                    'demands that N_sun is even, HS field couples '
                    'to the z-component of magnetization; '
                    'otherwise, HS field couples to the density',
         'defined_in_base': False,
         'value': True},
  'ham_T': {'comment': 'Hopping parameter',
            'defined_in_base': False,
            'value': 1.0},
  'ham_T2': {'comment': 'For bilayer systems',
             'defined_in_base': False,
             'value': 1.0},
  'ham_Tperp': {'comment': 'For bilayer systems',
                'defined_in_base': False,
                'value': 1.0}}),
 ('VAR_QMC',
  {'Amplitude': {'comment': 'Width of the box distribution for '
                            'update of type t=3,4 fields.  '
                            'Defaults  to 1.0.',
                 'value': 1.0},
  'CPU_MAX': {'comment': 'Code stops after CPU_MAX hours, if 0 or '
                         'not specified, the code stops after '
                         'Nbin bins',
              'value': 0.0},
  'Delta_t_Langevin_HMC': {'comment': 'Time step for Langevin or '
                                      'HMC',
                           'value': 0.1},
  'Global_moves': {'comment': 'Allows for global moves in space '
                              'and time.',
                   'value': False},
  'Global_tau_moves': {'comment': 'Allows for global moves on a '
                                  'single time slice.',
                       'value': False},
  'HMC': {'comment': 'HMC update', 'value': False},
  'LOBS_EN': {'comment': 'End measurements at time slice LOBS_EN',
              'value': 0},
  'LOBS_ST': {'comment': 'Start measurements at time slice '
                         'LOBS_ST',
              'value': 0},
  'Langevin': {'comment': 'Langevin update', 'value': False},
  'Leapfrog_steps': {'comment': 'Number of leapfrog iterations',
                     'value': 0},
  'Ltau': {'comment': '1 to calculate time-displaced Green '
                      'functions; 0 otherwise.',
           'value': 1},
  'Max_Force': {'comment': 'Max Force for Langevin', 'value': 5.0},
  'N_HMC_sweeps': {'comment': 'Number of HMC sweeps', 'value': 1},
  'N_global': {'comment': 'Number of global moves per sweep.',
```

(continues on next page)

(continued from previous page)

```
                'value': 1},
  'N_global_tau': {'comment': 'Number of global moves that will '
                              'be carried out on a single time '
                              'slice.',
                   'value': 1},
  'Nbin': {'comment': 'Number of bins.', 'value': 5},
  'Nsweep': {'comment': 'Number of sweeps per bin.', 'value': 20},
  'Nt_sequential_end': {'comment': '', 'value': -1},
  'Nt_sequential_start': {'comment': '', 'value': 0},
  'Nwrap': {'comment': 'Stabilization. Green functions will be '
                       'computed from scratch after each time '
                       'interval Nwrap*Dtau.',
            'value': 10},
  'Propose_S0': {'comment': 'Proposes single spin flip moves with '
                            'probability exp(-S0).',
                 'value': False},
  'sequential': {'comment': 'Conventional updating scheme',
                 'value': True}}),
('VAR_errors',
 {'N_Back': {'comment': 'If set to 1, substract background in '
                        'correlation functions. Is ignored in '
                        'Python analysis.',
             'value': 1},
  'N_Cov': {'comment': 'If set to 1, covariance computed for '
                       'time-displaced correlation functions. Is '
                       'ignored in Python analysis.',
            'value': 0},
  'N_auto': {'comment': 'If > 0, calculate autocorrelation. Is '
                        'ignored in Python analysis.',
             'value': 0},
  'N_rebin': {'comment': 'Rebinning: Number of bins to combine '
                         'into one.',
              'value': 1},
  'N_skip': {'comment': 'Number of bins to be skipped.',
             'value': 1}}),
('VAR_TEMP',
 {'N_Tempering_frequency': {'comment': 'The frequency, in units '
                                       'of sweeps, at which the '
                                       'exchange moves are '
                                       'carried out.',
                            'value': 10},
  'N_exchange_steps': {'comment': 'Number of exchange moves.',
                       'value': 6},
  'Tempering_calc_det': {'comment': 'Specifies whether the '
                                    'fermion weight has to be '
                                    'taken into account while '
                                    'tempering. Can be set to .F. '
                                    'if the parameters that get '
                                    'varied only enter the Ising '
                                    'action S_0',
                         'value': True},
  'mpi_per_parameter_set': {'comment': 'Number of mpi-processes '
                                       'per parameter set.',
                            'value': 2}}),
('VAR_Max_Stoch',
 {'Checkpoint': {'comment': '', 'value': False},
  'NBins': {'comment': 'Number of bins for Monte Carlo.',
            'value': 250},
  'NSweeps': {'comment': 'Number of sweeps per bin.', 'value': 70},
  'N_alpha': {'comment': 'Number of temperatures.', 'value': 14},
  'Ndis': {'comment': 'Number of boxes for histogram.',
```

(continues on next page)

(continued from previous page)

```
                'value': 2000},
 'Ngamma': {'comment': 'Number of Dirac delta-functions for '
                       'parametrization.',
            'value': 400},
 'Nwarm': {'comment': 'The Nwarm first bins will be ommitted.',
           'value': 20},
 'Om_en': {'comment': 'Frequency range upper bound.',
           'value': 10.0},
 'Om_st': {'comment': 'Frequency range lower bound.',
           'value': -10.0},
 'R': {'comment': '', 'value': 1.2},
 'Tolerance': {'comment': '', 'value': 0.1},
 'alpha_st': {'comment': '', 'value': 1.0}})])
```

4.2.2.2 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.
```

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

```
sim.run()
```

```
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/ALF_data/
 ↪Hubbard" 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
```

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/dissertation/jb/chap4_pyalf/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
 t2            :    1.0000000000000000
 Ham_U2        :    4.0000000000000000
 Ham_tperp     :    1.0000000000000000
 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 24234d19 of branch master.
  Precision Green  Mean, Max :    3.4949708528056399E-011   3.3070368266052697E-
↪007
  Precision Phase, Max       :    0.0000000000000000
  Precision tau    Mean, Max :    6.3437557434930669E-012   3.6803000015572795E-
↪008
  Acceptance                 :   0.42958333333333332
  Effective Acceptance       :   0.42958333333333332
  CPU Time                   :    4.0360565509999997
```

4.2.2.3 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/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/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
 t2            :    1.0000000000000000
```

(continues on next page)

(continued from previous page)

```
 Ham_U2         :    4.0000000000000000
 Ham_tperp      :    1.0000000000000000
 Ham_chem       :    0.0000000000000000
 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 24234d19 of branch master.
  Precision Green  Mean, Max :    3.6448966471271451E-011   2.5289980969123160E-
↪007
  Precision Phase, Max       :    0.0000000000000000
  Precision tau    Mean, Max :    8.7134381626013185E-012   2.0840410398792475E-
↪007
  Acceptance                 :   0.42600781250000003
  Effective Acceptance       :   0.42600781250000003
  CPU Time                   :    3.8189617760000001
```

4.2.2.4 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[31] 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 0x7f1550971c10>,
 <py_alf.simulation.Simulation at 0x7f155c1bdc40>,
 <py_alf.simulation.Simulation at 0x7f157c102630>]
```

[31] https://www.open-mpi.org/doc

Note

The above employs Python's list comprehensions[32], a convenient and readable way to create Python lists. Here is a simple example, employing list comprehension (and f-strings[33]):

```
>>> [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.
```

Loop over list of jobs:

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

```
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/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
```

(continues on next page)

[32] https://docs.python.org/3/tutorial/datastructures.html#tut-listcomps
[33] https://docs.python.org/3/reference/lexical_analysis.html#f-strings

(continued from previous page)

```
 This is free software, and you are welcome to redistribute it under certain␣
 ↪conditions.
 No initial configuration
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/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()
```

```
===== /home/jonas/dissertation/jb/chap4_pyalf/usage/ALF_data/Hubbard_L1=4_L2=4_
 ↪U=1.0/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         :    1.0000000000000000
 t2            :    1.0000000000000000
 Ham_U2        :    4.0000000000000000
 Ham_tperp     :    1.0000000000000000
 Ham_chem      :    0.0000000000000000
 No initial configuration, Seed_in      814342
 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
 Number of mpi-processes :            4
```

(continues on next page)

(continued from previous page)

```
 This executable represents commit 24234d19 of branch master.
  Precision Green  Mean, Max :    4.7770138116091871E-014   3.1588759386025345E-
↪012
  Precision Phase, Max       :    0.0000000000000000
  Precision tau    Mean, Max :    1.3814341899797833E-014   5.2165077812915683E-
↪012
  Acceptance                 :   0.44432578125000000
  Effective Acceptance       :   0.44432578125000000
  CPU Time                   :    4.6781053379999999

===== /home/jonas/dissertation/jb/chap4_pyalf/usage/ALF_data/Hubbard_L1=4_L2=4_
↪U=2.0/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         :    2.0000000000000000
 t2            :    1.0000000000000000
 Ham_U2        :    4.0000000000000000
 Ham_tperp     :    1.0000000000000000
 Ham_chem      :    0.0000000000000000
 No initial configuration, Seed_in      814342
 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
 Number of mpi-processes :            4
 This executable represents commit 24234d19 of branch master.
  Precision Green  Mean, Max :    2.6978900287668899E-013   1.9449365382118167E-
↪010
  Precision Phase, Max       :    0.0000000000000000
  Precision tau    Mean, Max :    7.6016248578584409E-014   2.8435920285119209E-
↪010
  Acceptance                 :   0.43524999999999997
  Effective Acceptance       :   0.43524999999999997
  CPU Time                   :    4.4471584740000001

===== /home/jonas/dissertation/jb/chap4_pyalf/usage/ALF_data/Hubbard_L1=4_L2=4_
↪U=3.0/info =====
 ====================================
 Model is      : Hubbard
 Lattice is    : Square
 # unit cells  :           16
```

(continues on next page)

(continued from previous page)

```
# 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         :    3.0000000000000000
t2            :    1.0000000000000000
Ham_U2        :    4.0000000000000000
Ham_tperp     :    1.0000000000000000
Ham_chem      :    0.0000000000000000
No initial configuration, Seed_in      814342
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
Number of mpi-processes :            4
This executable represents commit 24234d19 of branch master.
 Precision Green  Mean, Max :    2.6692496305360124E-012   6.9421481896370096E-
↪009
 Precision Phase, Max       :    0.0000000000000000
 Precision tau    Mean, Max :    6.9494808527483376E-013   8.4657683641076176E-
↪009
 Acceptance                 :   0.42918789062500001
 Effective Acceptance       :   0.42918789062500001
 CPU Time                   :    4.6657046165000002
```

4.2.2.5 Parallel Tempering

ALF offers the possibility to employ Parallel Tempering [114], also known as Exchange Monte Carlo [115], 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
```

(continues on next page)

(continued from previous page)

```
        } 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_
 ↪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/dissertation/jb/chap4_pyalf/usage/ALF_data/
 ↪temper_Hubbard_L1=4_L2=4_U=2.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/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/dissertation/jb/chap4_pyalf/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
```

```
sim.print_info_file()
```

The output from this command has been omitted for brevity.

4.2.2.6 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
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.
```

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

```
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK0.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK0.3" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK0.6" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK0.9" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK1.2" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK1.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK1.8" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK2.1" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK2.4" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK2.7" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/chap4_pyalf/usage/
 ↪KondoBilayerSquareL16/JK3.0" for Monte Carlo run.
Create new directory.
```

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

```
ls KondoBilayerSquareL16/*
```

```
KondoBilayerSquareL16/JK0.0:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK0.3:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK0.6:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK0.9:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK1.2:
```

(continues on next page)

(continued from previous page)

```
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK1.5:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK1.8:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK2.1:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK2.4:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK2.7:
ALF.out*  parameters  seeds

KondoBilayerSquareL16/JK3.0:
ALF.out*  parameters  seeds
```

4.2.3 Postprocessing

The following sections demonstrate the postprocessing features in pyALF, each section can be executed individually, if QMC raw data from Section 4.2.2 is present.

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

4.2.3.1 Basic analysis

As already shown in Section 4.2.1, 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[34], which is not necessary in this part, but for the functions used in Section 4.2.3.3, therefore it makes sense to establish it as a default.

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

# Imports
import numpy as np                        # Numerical libary
import matplotlib.pyplot as plt           # Plotting library
from py_alf.analysis import analysis      # Analysis function
from py_alf.utils import find_sim_dirs    # Function for finding QMC bins
from py_alf.ana import load_res           # Function for loading analysis results
```

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.

[34] https://github.com/matplotlib/ipympl

```
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',
 './ALF_data_back/Hubbard_Square']
```

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[35]. On the resulting bins, Jackknife resampling [116] is applied to estimate expectation values and their standard error. For brevity, the resulting printout is truncated.

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

```
### Analyzing ./ALF_data/Hubbard ###
/home/jonas/dissertation/jb/chap4_pyalf/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
### Analyzing ./ALF_data/Hubbard_L1=4_L2=4_U=1.0 ###
/home/jonas/dissertation/jb/chap4_pyalf/usage
Scalar observables:
...
...
...
```

[35] We will elaborate further on rebinning in Section 4.2.3.3.

4.2.3.1.1 Get analysis results

The analysis results are saved in each simulation directory, both in plain text in the folder `res` and as a `pickled`[36] 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`[37], 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.
./ALF_data_back/Hubbard_Square
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',
       './ALF_data_back/Hubbard_Square'],
      dtype='object')
```

Column indices can be accessed through:

```
res.columns
```

```
Index(['continuous', 'ham_chem', 'ham_t', 'ham_t2', 'ham_tperp', 'ham_u',
       'ham_u2', 'mz', 'l1', 'l2',
       ...
       'R_Ferro', 'R_Ferro_err', 'R_AFM', 'R_AFM_err', 'SpinZ_pipi',
       'SpinZ_pipi_err', 'SpinXY_pipi', 'SpinXY_pipi_err', 'SpinXYZ_pipi',
       'SpinXYZ_pipi_err'],
      dtype='object', length=129)
```

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

[36] https://docs.python.org/3/library/pickle.html#module-pickle
[37] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html#pandas.DataFrame

```
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 ALF_data/Hubbard/res
```

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

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.keys():
    if i.startswith('Ener_scal'):
        print(i, item[i])
```

```
Ener_scal_sign 1.0
Ener_scal_sign_err 0.0
Ener_scal0 -29.821914139681642
Ener_scal0_err 0.13032001865023543
```

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
-2.9821914139681641e+01 1.3032001865023543e-01
```

Example

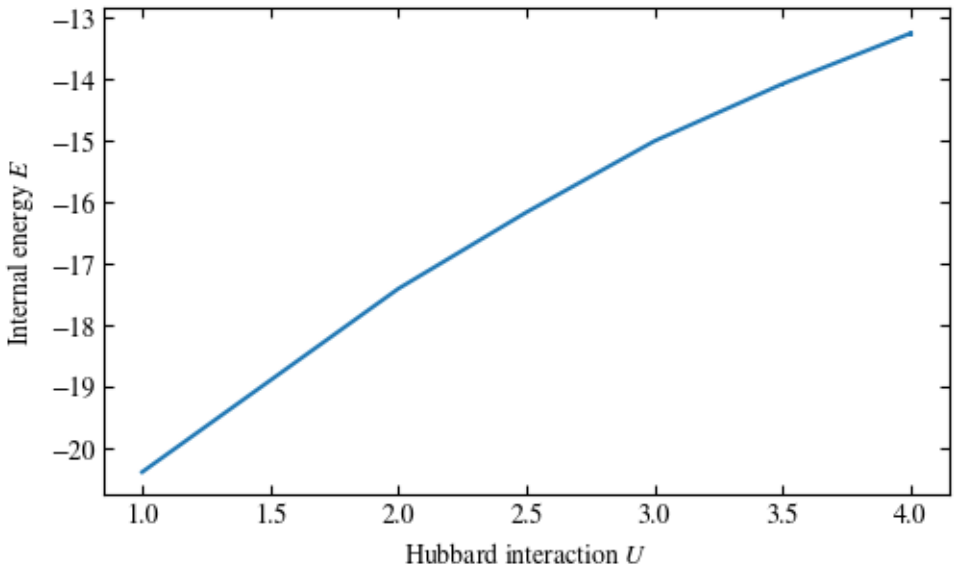
Here is a simple example that demonstrates the convenience of working with pandas DataFrames. We select out of all simulations the one with $L_1 = 4$ and plot their internal energy against The value of the Hubbard U.

```
# 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')
```

(continues on next page)

(continued from previous page)

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



Equal-time correlation functions

ALF and pyALF offer support for correlation functions of the form

$$\begin{aligned} C(\boldsymbol{r}, n_1, n_2) &= \frac{1}{N_r} \sum_{\boldsymbol{r}_0} \langle O(\boldsymbol{r}_0, n_1) O(\boldsymbol{r}_0 + \boldsymbol{r}, n_2) \rangle - \langle O(n_1) \rangle \langle O(n_2) \rangle \\ C(\boldsymbol{k}, n_1, n_2) &= \frac{1}{N_r} \sum_{\boldsymbol{r}} e^{i\boldsymbol{k}\boldsymbol{r}} C(\boldsymbol{r}, n_1, n_2) \end{aligned} \tag{4.1}$$

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.keys():
    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,)
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. (4.1) and their errors. They have the shape $(N_{\text{orb}}, N_{\text{orb}}, N_r)$, 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'])
```

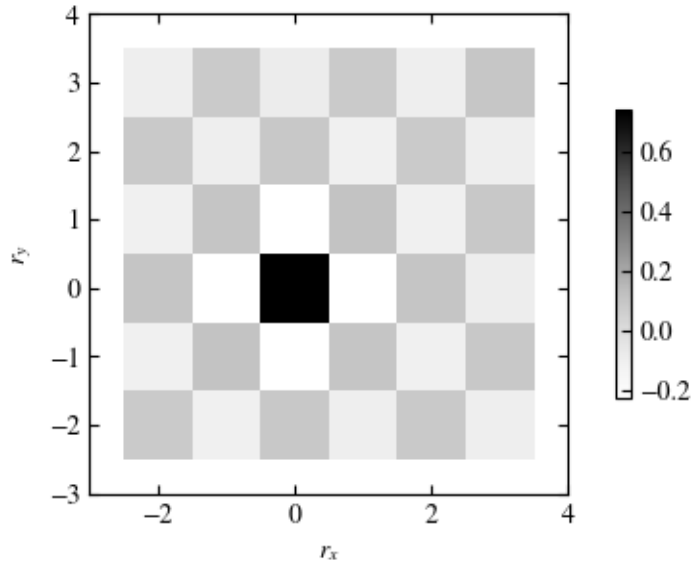
Here is, for example, the equal-time Greens function at $\boldsymbol{k} = (\pi, \pi)$ with its error:

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

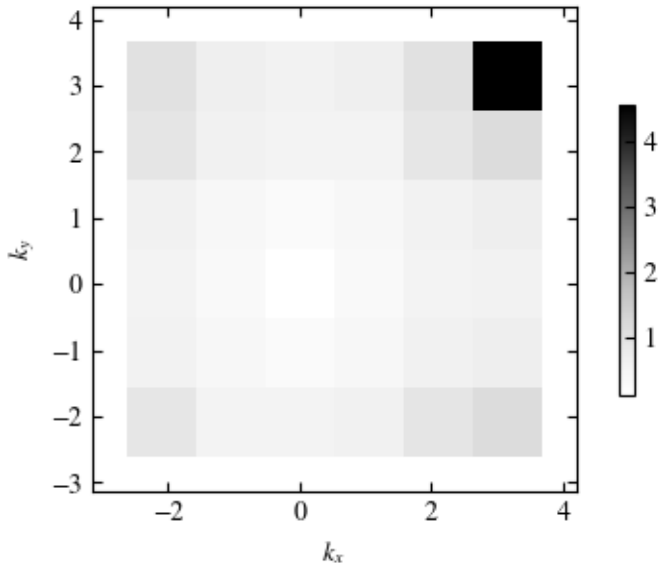
```
0.06526692607779117 0.0013046070203645834
```

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  6.0737564927e-03         1.
 ↪4417820888e-01  6.0737564927e-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.

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.keys():
    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 ()
```

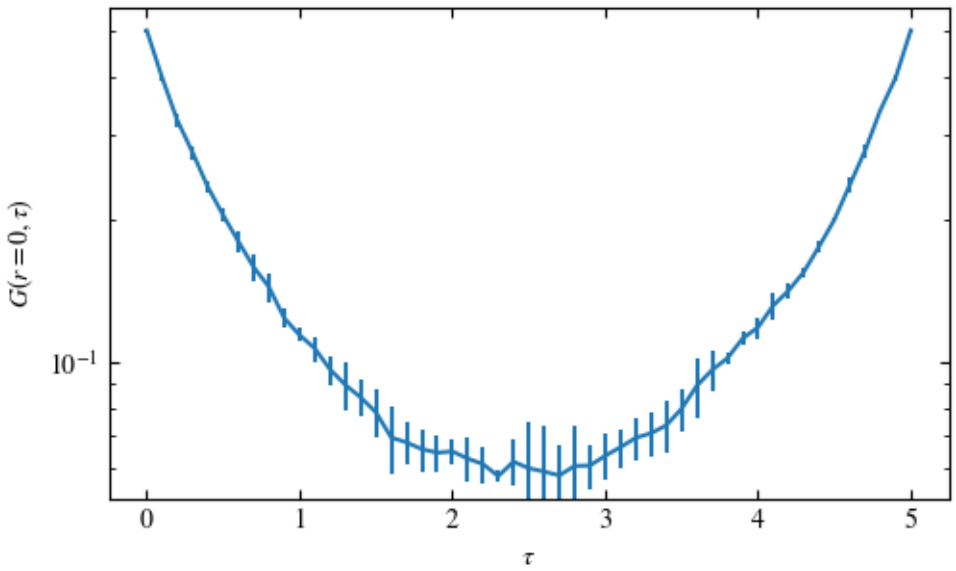
Here we plot the time-displaced Greens function at $\boldsymbol{r} = 0$:

```
# 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 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
```

4.2.3.2 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 numpy as np                          # Numerical libary
import matplotlib.pyplot as plt             # Plotting library
from py_alf.analysis import analysis        # Analysis function
from py_alf.utils import find_sim_dirs      # Function for finding QMC bins
from py_alf.ana import load_res             # Function for loading analysis results
```

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',
 './ALF_data_back/Hubbard_Square']
```

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:

$$R(O, \boldsymbol{k}_*) = 1 - \frac{O(\boldsymbol{k}_* + \boldsymbol{\delta})}{O(\boldsymbol{k}_*)} \tag{4.2}$$

Where $O(\boldsymbol{k})$ is a correlation function that has a divergence at $\boldsymbol{k} = \boldsymbol{k}_*$ in the ordered phase and $\boldsymbol{\delta}$ scales with $1/L$, where L is the linear system size. A usual choice for $\boldsymbol{\delta}$ is the smallest $\boldsymbol{k}$ on the finite-sized Bravais lattice. With these properties, $R(O, \boldsymbol{k}_*)$ 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 $\boldsymbol{k}_i$ and distances from those points $\boldsymbol{\delta}_j$, which results in:

$$R = \frac{1}{N_k} \sum_{i=1}^{N_k} \left(1 - \frac{\frac{1}{N_\delta} \sum_{j=1}^{N_\delta} O(\boldsymbol{k}_i + \boldsymbol{\delta}_j)}{O(\boldsymbol{k}_i)} \right) \tag{4.3}$$

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

$$O(\boldsymbol{k}) = \sum_{n,m} \tilde{O}(\boldsymbol{k})_{n,m} M_{n,m} \tag{4.4}$$

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 $\boldsymbol{k}$ points, the orbital structure and $\boldsymbol{\delta}_j$ 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 printout has again been truncated for brevity.

```
for directory in dirs:
    analysis(directory, custom_obs=custom_obs, always=True)
```

```
### Analyzing ./ALF_data/Hubbard ###
/home/jonas/dissertation/jb/chap4_pyalf/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
### Analyzing ./ALF_data/Hubbard_L1=4_L2=4_U=1.0 ###
/home/jonas/dissertation/jb/chap4_pyalf/usage
Custom observables:
...
...
...
```

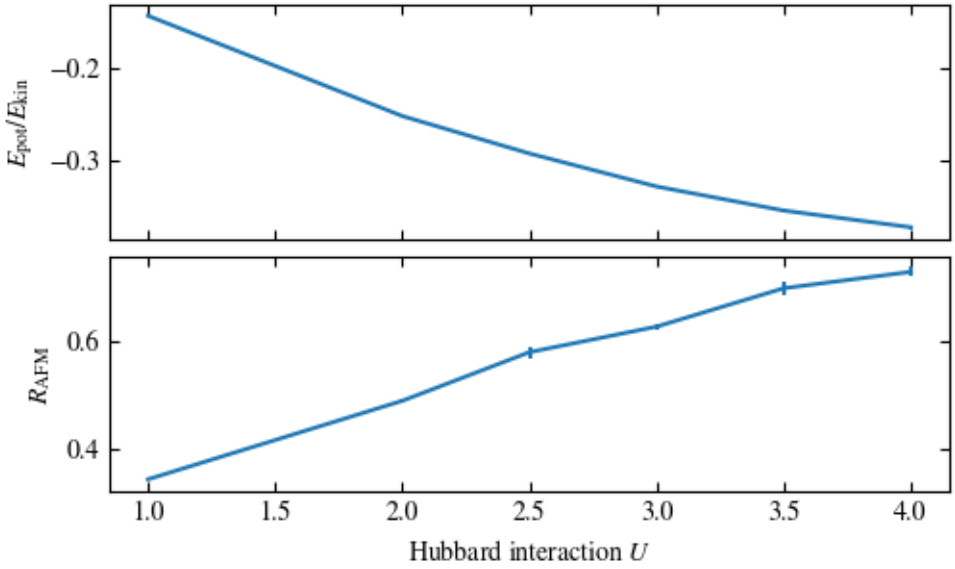
The results are loaded the same way as in the previous section:

```
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.
./ALF_data_back/Hubbard_Square
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);
```



4.2.3.3 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.

4.2.3.3.1 Preparations

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

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

import numpy as np                          # Numerical libary
import matplotlib.pyplot as plt             # Plotting library
from py_alf.analysis import analysis        # Analysis function
from py_alf.utils import find_sim_dirs      # Function for finding QMC bins
from py_alf.ana import load_res             # Function for loading analysis results
```

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_warmup, check_rebin
```

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
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/dissertation/jb/chap4_pyalf/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',
 './ALF_data_back/Hubbard_Square']
```

4.2.3.3.2 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[38]). 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:

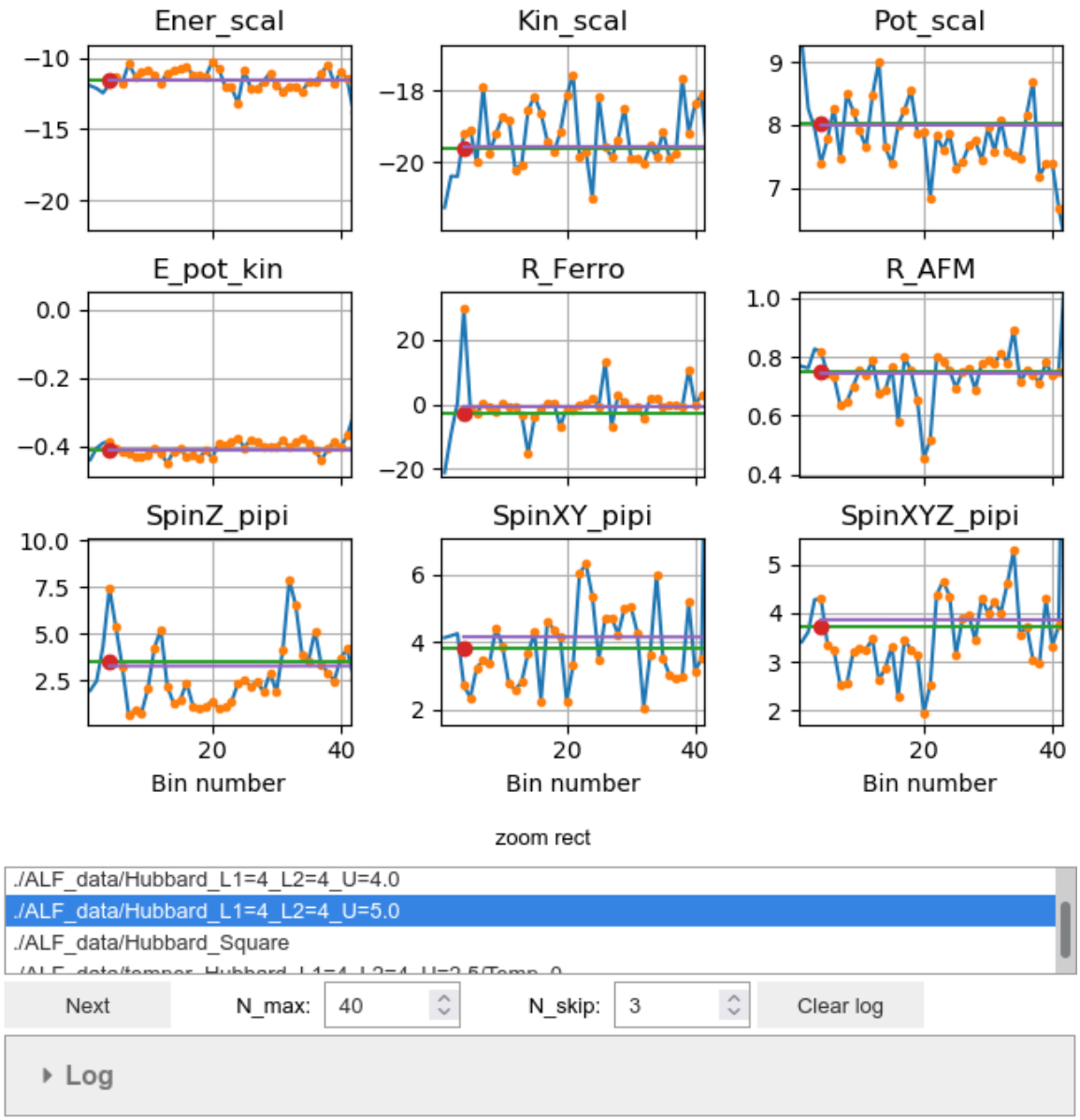
```
warmup_widget = check_warmup(
    dirs,
    ['Ener_scal', 'Kin_scal', 'Pot_scal',
     'E_pot_kin', 'R_Ferro', 'R_AFM',
     'SpinZ_pipi', 'SpinXY_pipi', 'SpinXYZ_pipi'],
    custom_obs=custom_obs, gui='ipy'
)
```

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[39], which allows to seamlessly work within Jupyter. Another option would be `gui='tk'`, which opens a separate window using `tkinter`[40]. 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
```

[38] https://git.physik.uni-wuerzburg.de/ALF/ALF/-/jobs/artifacts/master/raw/Documentation/doc.pdf?job=create_doc
[39] https://ipywidgets.readthedocs.io
[40] https://docs.python.org/3/library/tkinter.html#module-tkinter



4.2.3.3.3 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()`*.

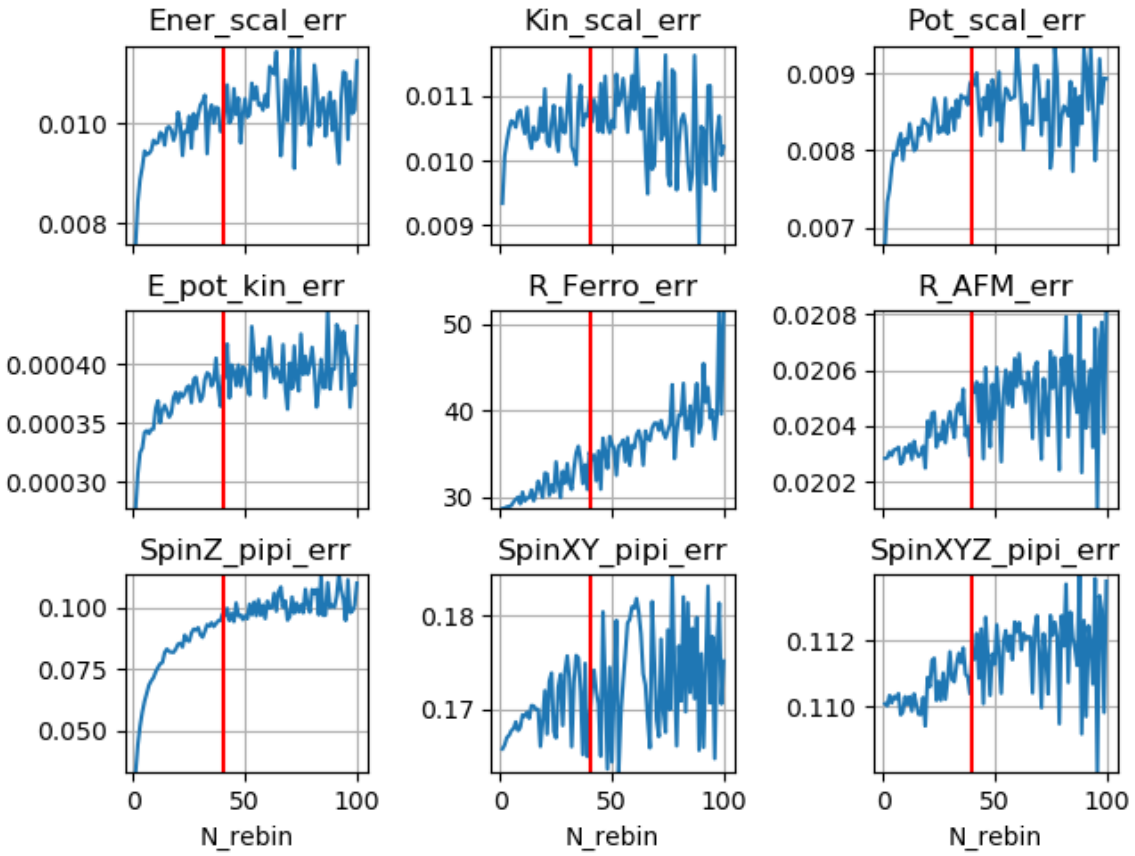
```
rebin_widget = check_rebin(
    dirs,
    ['Ener_scal', 'Kin_scal', 'Pot_scal',
     'E_pot_kin', 'R_Ferro', 'R_AFM',
     'SpinZ_pipi', 'SpinXY_pipi', 'SpinXYZ_pipi'],
    custom_obs=custom_obs, gui='ipy'
)
```

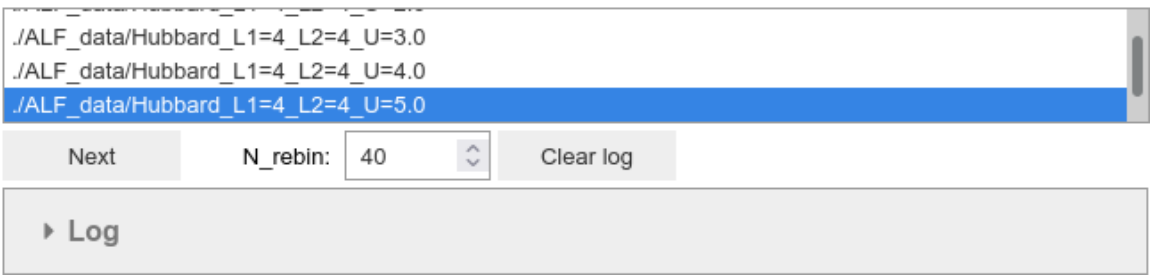
Below, we can see how different observables have different autocorrelation times. While the error of the kinetic energy saturates already at $N_{\text{rebin}} = 3$, the correlations of the z component of the spin at (π, π) (`SpinZ_pipi`) need $N_{\text{rebin}} \sim 40$. For the correlations of the total 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 4.2.2) 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.

4.2.3.4 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 $\boldsymbol{k}$-point of the Bravais lattice and returning an integer corresponding to the transformed $\boldsymbol{k}$-point of the Bravais lattice. The analysis then averages the correlation over all transformations:

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

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 numpy as np                          # Numerical libary
import matplotlib.pyplot as plt             # Plotting library
from py_alf.analysis import analysis        # Analysis function
from py_alf.ana import load_res             # Function for loading analysis results
from py_alf import Lattice                  # Defines Bravais lattice object
from custom_obs import custom_obs           # Custom observable specifications
                                            # 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
def sym_c4_0(latt, i): return i
def sym_c4_1(latt, i): return latt.rotate(i, np.pi*0.5)
def sym_c4_2(latt, i): return latt.rotate(i, np.pi)
def sym_c4_3(latt, i): return latt.rotate(i, np.pi*1.5)

sym_c4 = [sym_c4_0, sym_c4_1, sym_c4_2, sym_c4_3]
```

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/dissertation/jb/chap4_pyalf/usage
Custom observables:
```

(continues on next page)

(continued from previous page)

```
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/dissertation/jb/chap4_pyalf/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
```

(continues on next page)

(continued from previous page)

```
./ALF_data/Hubbard
No orbital locations saved.
```

We now compare results for the points $(\pi, \pi) + \boldsymbol{b}_1$ and $(\pi, \pi) + \boldsymbol{b}_2$, where $\boldsymbol{b}_1 = (2\pi/L, 0)$ and $\boldsymbol{b}_2 = (0, 2\pi/L)$ are the primitive vectors of the Bravais lattice in k-space, with and without symmetrization.

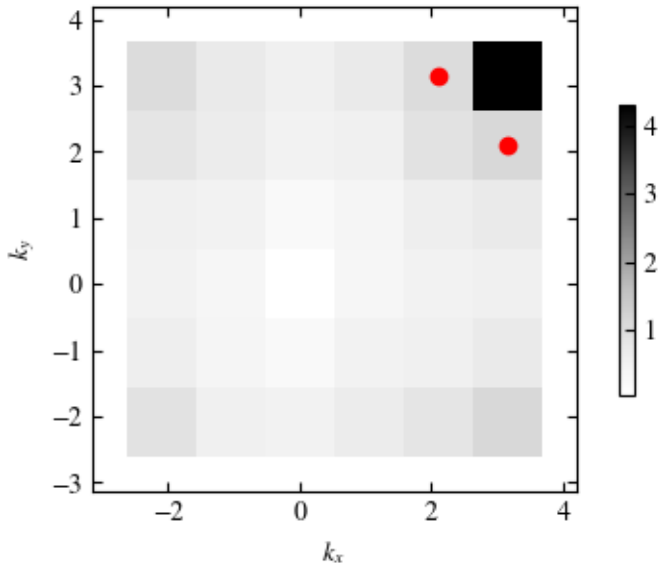
```
latt = Lattice(res_nosym['SpinT_eq_lattice'])
n = latt.k_to_n((np.pi, np.pi))
n1 = latt.nnlistk[n, -1, 0]
n2 = latt.nnlistk[n, 0, -1]
```

```
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
```

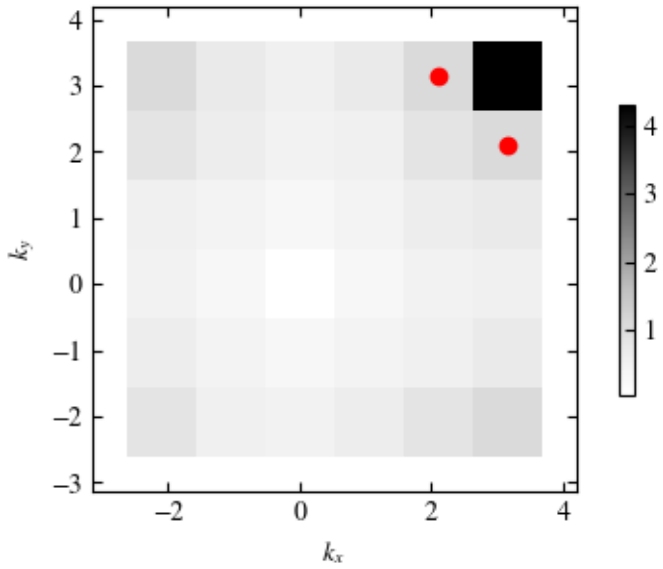
```
latt.plot_k(res_nosym.SpinT_eqK_sum)
plt.plot(*latt.k[n1], 'or')
plt.plot(*latt.k[n2], 'or')
```

```
[<matplotlib.lines.Line2D at 0x7f6da8c85f70>]
```



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

```
[<matplotlib.lines.Line2D at 0x7f6da097a2a0>]
```



4.2.4 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` and, as mentioned in Section 4.1, it is recommended to add this folder to the `$PATH` environment variable, to conveniently use the scripts:

```
export PATH="/path/to/pyALF/py_alf/cli:$PATH"
```

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.py` and `alf_postprocess.py`.

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.

4.2.4.1 alf_run.py

The script `alf_run.py` enables most of the features displayed in Section 4.2.2 to be used directly from the shell. The help text lists all possible arguments:

```
!alf_run.py -h
```

```
usage: alf_run.py [-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.

optional arguments:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --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 4.2.2 might help. The following printout is truncated for brevity.

```
!alf_run.py --sims_file ./Sims_Kondo --mpi --n_mpi 4 --mpiexec orterun
```

```
Number of simulations: 4
Compiling ALF...
Cleaning up Prog/
Cleaning up Libraries/
Cleaning up Analysis/
Compiling Libraries
ar: creating modules_90.a
ar: creating libqrref.a
Compiling Analysis
Compiling Program
Parsing Hamiltonian parameters
filename: Hamiltonians/Hamiltonian_Kondo_smod.F90
filename: Hamiltonians/Hamiltonian_Hubbard_smod.F90
filename: Hamiltonians/Hamiltonian_Hubbard_Plain_Vanilla_smod.F90
filename: Hamiltonians/Hamiltonian_tV_smod.F90
filename: Hamiltonians/Hamiltonian_LRC_smod.F90
filename: Hamiltonians/Hamiltonian_Z2_Matter_smod.F90
Compiling program modules
Link program
Done.
Prepare directory "/scratch/pyalf-docu/doc/source/usage/ALF_data/Kondo_L1=4_
 ↪L2=4_JK=0.5" for Monte Carlo run.
Create new directory.
Run /home/jschwab/Programs/ALF/Prog/ALF.out
 ALF Copyright (C) 2016 - 2021 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
Compiling ALF...
...
...
...
```

4.2.4.2 alf_postprocess.py

The script `alf_postprocess.py` enables most of the features discussed in Section 4.2.3, except for plotting capabilities, to be used directly from the shell. The help text lists all possible arguments:

```
!alf_postprocess.py -h
```

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

optional arguments:
  -h, --help            show this help message and exit
  --check_warmup, --warmup
                        Check warmup.
  --check_rebin, --rebin
                        Check rebinning for controlling autocorrelation.
  -l CHECK_LIST [CHECK_LIST ...], --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 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
```

(continues on next page)

(continued from previous page)

```
# 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. The printout has again been truncated.

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

```
### Analyzing ALF_data/Kondo_L1=4_L2=4_JK=0.5 ###
/scratch/pyalf-docu/doc/source/usage
Custom observables:
custom E_squared ['Ener_scal']
custom E_pot_kin ['Pot_scal', 'Kin_scal']
custom SpinZ_pipi ['SpinZ_eq']
Scalar observables:
Constraint_scal
Ener_scal
Kin_scal
Part_scal
Pot_scal
Histogram observables:
Equal time observables:
Den_eq
Dimer_eq
Green_eq
SpinZ_eq
Time displaced observables:
Den_tau
Dimer_tau
Green_tau
Greenf_tau
SpinZ_tau
### Analyzing ALF_data/Kondo_L1=4_L2=4_JK=1.0 ###
/scratch/pyalf-docu/doc/source/usage
...
...
...
ALF_data/Kondo_L1=4_L2=4_JK=0.5
ALF_data/Kondo_L1=4_L2=4_JK=1.0
ALF_data/Kondo_L1=4_L2=4_JK=1.5
ALF_data/Kondo_L1=4_L2=4_JK=2.0
```

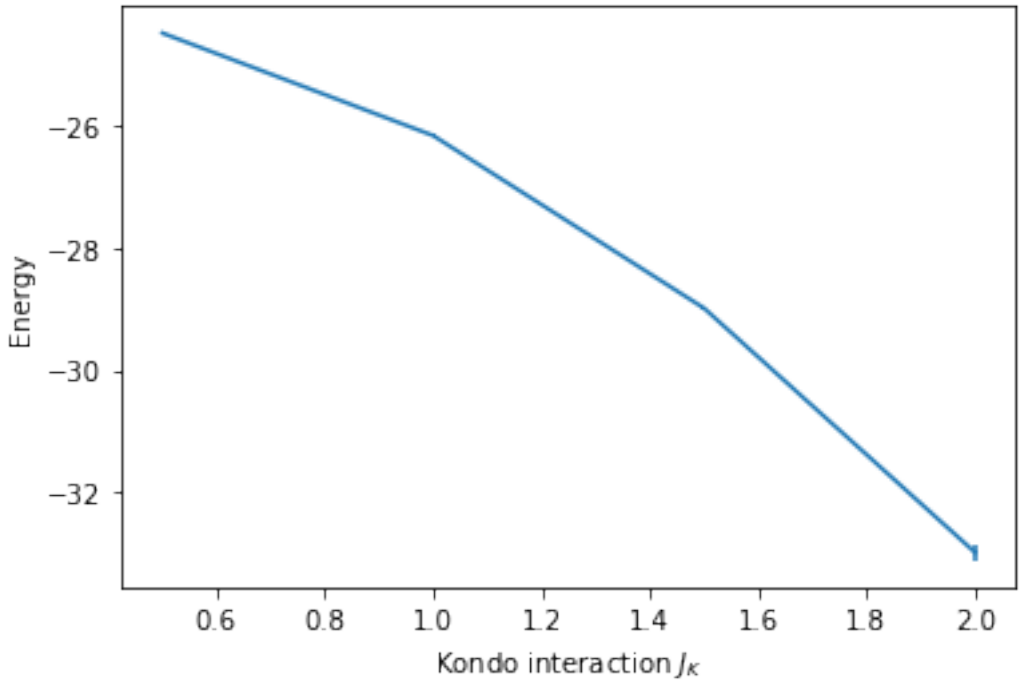
The data from `gathered.pkl` can, for example, be read and plotted like this:

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

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

# Create figure with axis labels
```

(continues on next page)

(continued from previous page)

```
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);
```



4.3 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()`[41].

Table of contents

[41] https://docs.python.org/3/library/functions.html#help

4.3.1 Class ALF_source

class py_alf.**ALF_source**(*alf_dir=None*, *branch=None*, *url='https://git.physik.uni-wuerzburg.de/ALF/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://git.physik.uni-wuerzburg.de/ALF/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.

4.3.2 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.

4.3.3 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

4.3.4 Class Lattice

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

Finite size Bravais lattice object.

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 vector 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 behaviour is to first try compiled Fortran and fall back to Python if that fails.

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.

4.3.5 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.

4.3.6 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.

4.3.7 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.

4.3.7.1 minimal_ALF_run

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

4.3.7.2 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]
```

4.3.7.2.1 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.

4.3.7.3 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 ...]
```

4.3.7.3.1 Positional Arguments

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

4.3.7.3.2 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))

4.3.7.4 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 ...]
```

4.3.7.4.1 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.

4.3.7.5 alf_show_obs

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

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

4.3.7.5.1 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.

4.3.7.6 alf_del_bins

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

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

4.3.7.6.1 Positional Arguments

filename Name of HDF5 file.

4.3.7.6.2 Named Arguments

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

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

4.3.7.7 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]
```

4.3.7.7.1 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

FIVE

CONCLUSIONS

In this thesis, I have presented three major projects that collectively contribute to advancing our understanding of highly correlated quantum systems. Each project was centered around the use of auxiliary field quantum Monte Carlo, leveraging the capabilities of the Algorithms for Lattice Fermions (ALF) package.

The first project (Chapter 2), conducted in collaboration with renormalization group (RG) experts, investigated nematic quantum phase transitions in Dirac fermions, where the Dirac points are pinned in the disordered phase and start to meander in the ordered phase. While there have been previous analytical investigations of this kind of transition, there has been no clear consensus whether it is of first or second order [37, 50, 51, 52, 53]. Set out to solve this long-standing problem, we conducted the first exact numerical investigation: We designed two sign-problem-free models exhibiting the intended phase transition. In our QMC studies combined with ϵ-expansion, we found the transitions to be continuous, with a quantum critical regime that is characterized by large velocity anisotropies of the Dirac cones. These velocity anisotropies turned out to have a very slow RG flow (for one of the two models, we could explicitly show that it diverges logarithmically with the RG parameter). As a result, finitely sized systems, in both experimental and numerical investigations, will not be representative of the infrared fixed point, but of a quasiuniversal regime where the drift of the exponents tracks the velocity anisotropy. Notably, even though the ϵ-expansion finds qualitatively distinct fixed points for the two investigated models, the numerical investigation finds no distinction in the exponents, within the margin of errors. Therefore, it seems that the quasiuniversial regime is —at least close to the ultraviolet beginning— identical for both models, even though their infrared universality is different. A particular challenge in this project was the broken Lorentz symmetry caused by both the Fermi velocity anisotropy and the meandering Dirac points. In particular, the variable zero-dimensional Fermi surface on a finite lattice lead to confusing artifacts that could be misinterpreted as sings of a first order phase transition. The findings of this project have already been published in Phys. Rev. Lett. [14].

In my second project (Chapter 3), we set out to perform the first exact numerical investigation to complement the seminal work of Read and Sachdev [18, 20, 21]. Hence, I simulated a generalized Heisenberg model on a square lattice, where each site hosts an irreducible representation of SU(N) described by a square Young tableau with $N/2$ rows and $2S$ columns. With my QMC simulations, we were able to map out its ground state phase diagram for $S \in \{1/2, 1, 3/2, 4\}$ and $N \in \{2, 4, \ldots, 22\}$. Confirming the analytical work by Read and Sachdev, we found antiferromagnetic order for big S and dimerized order for big N. Along a line defined by $N = 8S + 2$ in the S versus N phase diagram, we observe a rich variety of phases. For $S = 1/2$ and $3/2$, the system forms a four-fold degenerate VBS state, while for $S = 1$, we identify a two-fold degenerate spin nematic state that breaks the C_4 lattice symmetry down to C_2. At $S = 2$, we observe a unique symmetry-protected topological state, characterized by a dimerized SU(18) boundary state, reminiscent of the two-dimensional Affleck-Kennedy-Lieb-Tasaki (AKLT) state. These phases proximate to the Néel state align with the theoretical framework of monopole condensation of the antiferromagnetic order parameter, with degeneracies following a mod$(4, 2S)$ rule. The findings of this project have already been published in [17].

The third project is in a sense a meta-project, since it is –at least in parts– the result of my adaptions and extensions of ALF for working more efficiently on the first two projects. In Chapter 4 this is represented by the documentation of pyALF, a Python library for running and analyzing ALF simulations, but I also supplied significant contributions to the ALF Fortran code [13]. My most noteworthy contributions to ALF include an improved encapsulation of model definitions, implementing HDF5 for observables and an extension of the automatic test executed by our development platform, GitLab.

All in all, I believe my research offers new insights into highly correlated quantum systems and both my contributions to ALF and the development of pyALF provide significant advances in tooling for the numerical study of condensed matter models.

5.1 Outlook for (py)ALF

Going forward, there are still many possibilities to further improve ALF and pyALF. What might come to mind first, are new features for models and enhancements on the QMC algorithm, such as the new type of interaction vertex for implementing general gauge theories the ALF collaboration is currently working on[1]. Furthermore, ALF will have to start leveraging GPUs, if we want to keep on track with advances in HPC computing. But there is also a big potential for improving the structure of the code and its usability.

In particular, further development of pyALF has a big potential for improving usability. Lowering the barrier of entry can open ALF to a whole new host of users. I would like to group such improvements into two categories:

1. Reducing the needed IT knowledge, i.e. keeping the users away from Fortran code, compilers, terminal shells, etc.
2. Reduce the needed QMC knowledge, by automating common accuracy checks and detection of failing simulations.

Here are three partly interconnected measures from the first category:

- Exposing ALF directly to Python through a module, instead of executing the ALF binary through a system call –as pyALF does it currently–, would generate a smoother user experience and eliminate many potential points of failure.
- Furthermore, a possibility to define Hamiltonians directly in Python instead of Fortran would also lower the barrier of entry significantly. Here, the biggest challenge would be to find a way for implementing new observables.
- Lastly, shipping binaries with the Python package would also improve the user experience, since they would not have to deal with compilers and the ALF source code any more, on the other hand, this may come at the expense of performance losses.

To let users treat (py)ALF as a black box that returns accurate results with error bars, we will have to automate a number of common checks. Properties to test are the Green function precision (i.e. numerical stability of the algorithm), warmup times, autocorrelation times, spikes from fat-tailed distributions and systematic errors from the imaginary time step Δ_τ. Although the last one might also be kept with the user. Furthermore, estimating simulation times and issuing warnings if jobs might take longer than expected will also be a great help for novice users.

Checking the Green function precision will be relatively easy. One could issue a warning (pronounced enough to prompt most users to investigate) if the average or maximal deviation are above a certain threshold, e.g. 10^{-5} and 0.1, respectively. Alternatively, one could take the management of stabilization completely out of the hand of (novice) users, by auto-tuning stabilization intervals and even switching between stabilization schemes if e.g. the scales become too big.

The correct estimation of warmup and autocorrelation times poses a bigger challenge. The general strategy for warmup estimations would most certainly involve a linear fit of the observable time series $O(t)$ and dismissal of the leading elements until the slope is zero within some bounds. Though finding good criteria might be tricky, since for fluctuating data, a fit could be flat within error bars but still show a significant drift. For autocorrelation time, one could fit $\bar{\gamma}_\tau(O) = \exp\left(-\frac{\tau}{\tau_\gamma(O)}\right)$ (cf. Eq. (1.7)), but results from this still have to be treated carefully. Overall, one can never be entirely sure the Markov Chain is not stuck in a local maximum of the weight, or if there is a very slow mode that can not be resolved yet.

To detect whether the simulation might sample a fat-tailed distribution, the observable time series has to be scanned for outliers –so-called spikes–, e.g. by comparing the deviation of individual bins from the mean to the average fluctuation, or by arranging the bins into a histogram.

Implementing these checks to at least work in many cases should be feasible, but extensive tests will be in order. Generally, my approach here would be to err on the side of caution and prompt the user to look directly at the data if the data quality looks ambiguous to the algorithm.

With these ideas for making (py)ALF more approachable, I am concluding my thesis.

[1] https://git.physik.uni-wuerzburg.de/ALF/ALF/-/issues/297

Appendix

APPENDIX

A

APPENDIX TO "NEMATIC QUANTUM CRITICALITY IN DIRAC SYSTEMS"

The appendix for Chapter 2.

- *Renormalization group flow*
- *pyALF Example*
- *Source code of data collapse functions*
- *Source code for exponential fit of Green function*
- *Other values for N_σ and ξ*

A.1 Renormalization group flow

In this appendix, we present details of the renormalization group (RG) analysis of the continuum field theories. Due to the lack of Lorentz and continuous spatial rotational symmetries in the low-energy models, the Fermi and bosonic velocities, as well as their anisotropies, will in general receive different loop corrections. In order to appropriately take this multiple dynamics [60, 61] into account, it is useful to employ a regularization in the frequency only, which preserves the property that the different momentum components can be rescaled independently. This allows us to keep the boson velocities $c \equiv c_+ = c_-$ fixed, i.e., we measure the Fermi velocities in units of $c = 1$. Integrating over the "frequency shell" $\Lambda/b \leq |\omega| \leq \Lambda$ with $b > 1$ and *all* momenta causes the velocities and couplings to flow at criticality $r = 0$ as

$$\begin{aligned}
\frac{\mathrm{d}v_\parallel}{\mathrm{d}\ln b} &= \frac{1}{2}(\eta_\phi - \eta_+ - 2\eta_\psi)v_\parallel - F(v_\parallel, v_\perp)g^2, \\
\frac{\mathrm{d}v_\perp}{\mathrm{d}\ln b} &= \frac{1}{2}(\eta_\phi - \eta_- - 2\eta_\psi)v_\perp + F(v_\perp, v_\parallel)g^2, \\
\frac{\mathrm{d}g^2}{\mathrm{d}\ln b} &= \left(\epsilon - \frac{\eta_+ + \eta_-}{2} - 2\eta_\psi\right) g^2 - 2G(v_\parallel, v_\perp)g^4, \\
\frac{\mathrm{d}\lambda}{\mathrm{d}\ln b} &= \left(\epsilon - \frac{\eta_+ + \eta_-}{2} - \eta_\phi\right) \lambda - 18\lambda^2 + \frac{N' g^4}{16 v_\parallel v_\perp},
\end{aligned} \tag{1.1}$$

with the anomalous dimensions $\eta_\psi = g^2 H(v_\parallel, v_\perp)$, $\eta_\phi = N' g^2/(12 v_\parallel v_\perp)$, and $\eta_\pm = a_\pm N' g^2 v_\perp/(12 v_\parallel)$, to the one-loop order. Here, the angular integrals are performed in $d = 2$, while the dimensions of the couplings are counted in general d [51, 117]. At the present order, the flows of the two models differ only in the definition of the coefficients $a_\pm$, with $a_+ = 0$, $a_- = 2$ ($a_+ = a_- = 1$) in the C_{2v} (C_{4v}) model, and the number of spinor components $N' = 4N_\sigma$ ($N' = 8N_\sigma$). Our regularization scheme allows the evaluation of the one-loop integrals in closed form, leading to

the functions

$$
\begin{aligned}
F(v_1,v_2) &= \frac{1}{\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{v_1 q_1^2 \mathrm{d}q_1 \mathrm{d}q_2}{\left(1+q_1^2+q_2^2\right)^2\left(1+v_1^2q_1^2+v_2^2q_2^2\right)} \\
&= \frac{v_1\left[v_1\left(v_2^2-1\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}+\left(v_1+v_2\right)\sin^{-1}\left(v_1\sqrt{\frac{v_2^2-1}{v_2^2-v_1^2}}\right)-\left(v_1+v_2\right)\csc^{-1}\left(\sqrt{\frac{v_2^2-v_1^2}{v_2^2-1}}\right)\right]}{\left(v_1^2-1\right)\left(v_2^2-1\right)\left(v_1+v_2\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}}, \qquad (1.2)\\
G(v_1,v_2) &= \frac{1}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{1-v_1^2q_1^2+v_2^2q_2^2}{\left(1+q_1^2+q_2^2\right)\left(1+v_1^2q_1^2+v_2^2q_2^2\right)^2}\mathrm{d}q_1\mathrm{d}q_2 \\
&= \frac{\left(v_2^2-1\right)\left(v_1v_2-1\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}+\left(v_1^2+v_2^2-2\right)\csc^{-1}\left(\frac{1}{v_1}\sqrt{\frac{v_2^2-v_1^2}{v_2^2-1}}\right)-\left(v_1^2+v_2^2-2\right)\csc^{-1}\left(\sqrt{\frac{v_2^2-v_1^2}{v_2^2-1}}\right)}{2\left(v_1^2-1\right)\left(v_2^2-1\right)^2\sqrt{\frac{1-v_1^2}{v_2^2-1}}} \\
&\quad+\frac{v_1\left[\left(v_2^2-1\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}-v_1(v_1+v_2)\sin^{-1}\left(\sqrt{\frac{v_2^2-1}{v_2^2-v_1^2}}\right)+v_1(v_1+v_2)\csc^{-1}\left(\frac{1}{v_1}\sqrt{\frac{v_2^2-v_1^2}{v_2^2-1}}\right)\right]}{2\left(v_1^2-1\right)\left(v_2^2-1\right)\left(v_1+v_2\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}} \\
&\quad+\frac{v_2^4(v_1+v_2)\sin^{-1}\left(\sqrt{\frac{v_2^2-1}{v_2^2-v_1^2}}\right)}{2v_2^2\left(v_2^2-1\right)^2\left(v_1+v_2\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}}+\frac{\left(v_1^3+v_1^2v_2+v_1v_2^2+v_2^3\right)\sin^{-1}\left(v_1\sqrt{\frac{v_2^2-1}{v_2^2-v_1^2}}\right)-2\left(v_1^2-1\right)v_2^3\sqrt{\frac{1-v_1^2}{v_2^2-1}}}{4\left(v_1^2-1\right)v_2^2\left(v_2^2-1\right)\left(v_1+v_2\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}} \\
&\quad-\frac{\left(v_2^2+1\right)\left[v_1^3\left(2v_2^2-1\right)+v_1^2v_2\left(2v_2^2-1\right)-v_1v_2^2-v_2^3\right]\csc^{-1}\left(\frac{1}{v_1}\sqrt{\frac{v_2^2-v_1^2}{v_2^2-1}}\right)}{4\left(v_1^2-1\right)v_2^2\left(v_2^2-1\right)^2\left(v_1+v_2\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}}, \\
H(v_1,v_2) &= \frac{1}{\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{\mathrm{d}q_1\mathrm{d}q_2}{\left(1+q_1^2+q_2^2\right)^2\left(1+v_1^2q_1^2+v_2^2q_2^2\right)} \\
&= \frac{\left(v_1^2+v_2^2-2v_1^2v_2^2\right)\left[\csc^{-1}\left(\sqrt{\frac{v_2^2-v_1^2}{v_2^2-1}}\right)-\csc^{-1}\left(\frac{1}{v_1}\sqrt{\frac{v_2^2-v_1^2}{v_2^2-1}}\right)\right]}{\left(v_1^2-1\right)\left(v_2^2-1\right)^2\sqrt{\frac{1-v_1^2}{v_2^2-1}}}-\frac{\left(v_1v_2-1\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}}{\left(v_1^2-1\right)\left(v_2^2-1\right)\sqrt{\frac{1-v_1^2}{v_2^2-1}}}.
\end{aligned}
$$

(1.3)

The above one-loop flow equations admit a nontrivial fixed point that is characterized by anisotropic Fermi velocities $v_\parallel^* = 0$ and $v_\perp^* = 1/\sqrt{a_-} > 0$, and vanishing g_*^2 and λ^*, but finite ratio $(g^2/v_\parallel)_* = 12\sqrt{a_-}\epsilon/N' + \mathcal{O}(\epsilon^2)$. Perturbations of the couplings g^2 and λ and the Fermi velocity $v_\perp$ away from this fixed point turn out to be irrelevant; however, the flow of $v_\parallel$ near the fixed point is

$$
\left.\frac{\mathrm{d}v_\parallel}{\mathrm{d}\ln b}\right|_{g_*^2,v_\perp^*} = \frac{\epsilon}{2}(1-a_+)v_\parallel - \frac{20\epsilon}{N'}v_\parallel^2 + \mathcal{O}(v_\parallel^3). \qquad (1.4)
$$

Hence, in the C_{4v} model with $a_+ = a_- = 1$, $v_\parallel$ is marginally irrelevant, rendering the fixed point stable. The fixed point represents a quantum critical point with maximally anisotropic Fermi velocities $(v_\parallel^*, v_\perp^*) = (0,1)$ and boson anomalous dimensions, describing the temporal and spatial decays of the order-parameter correlations, as $\eta_\phi = \epsilon$ and $\eta_+ = \eta_- = \epsilon$, respectively. The fermion anomalous dimension becomes $\eta_\psi = 0$. In the vicinity of this fixed point, the flow of $v_\parallel$ can be integrated out analytically, reading

$$
v_\parallel(b) \simeq \frac{N'}{20\epsilon\ln b}, \qquad (1.5)
$$

where we have assumed $b \gg 1$ for simplicity. This demonstrates that the Fermi velocity flow in the vicinity of the C_{4v} fixed point is logarithmically slow, reflecting the fact that $v_\parallel$ is marginally irrelevant at this fixed point. This indicates that exponentially large lattice sizes are needed to ultimate reach the fixed point. By contrast, in the C_{2v} model with $a_+ = 0$ and $a_- = 2$, $v_\parallel$ is a relevant parameter near the maximal-anisotropy fixed point and flows to larger values. By numerically integrating out the flow, we find that the parameters $v_\perp$, $v_\parallel$, and g^2 flow to a new nontrivial stable fixed point at which the boson anomalous dimensions satisfy a sum rule, $\eta_+ + \eta_- + 2\eta_\phi = 2\epsilon$ with $0 = \eta_+ < \eta_-, \eta_\phi < \epsilon$. The fixed point is located at $(v_\parallel^*, v_\perp^*) = (0.1611, 0.5942)$ and $(g_*^2, \lambda^*) = (0.2123, 0.1755)\epsilon + \mathcal{O}(\epsilon^2)$ for $N' = 4N_\sigma = 4$. We find the corresponding anomalous dimensions as $(\eta_\phi, \eta_+, \eta_-, \eta_\psi) = (0.7391, 0, 0.5219, 0.1643)\epsilon + \mathcal{O}(\epsilon^2)$, reflecting again the fact that the character of the stable fixed point in the C_{2v} model is different from the one of

the C_{4v} model. The different behaviors of the Fermi velocities in the two models is illustrated in Fig. A1, which shows the renormalization group flow in the $v_\parallel$-$v_\perp$ plane. For visualization purposes, we have fixed the ratios $g^2/(v_\perp v_\parallel)$ to their values at the respective stable fixed points in these plots. We have explicitly verified that $g^2/(v_\perp v_\parallel)$ corresponds to an irrelevant parameter near these fixed points (marked as red dots in Fig. A1).

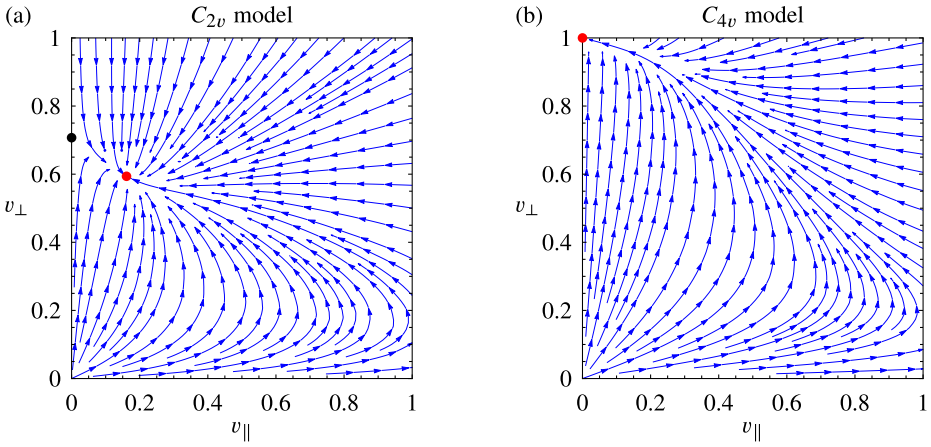


Fig. A1: Renormalization group flow in the $v_\parallel$-$v_\perp$ plane for (a) the C_{2v} model and (b) the C_{4v} model. Arrows denote flow towards infrared. The fixed point at $(v_\parallel^*, v_\perp^*) = (0, 1/\sqrt{a_-})$ and $(g^2/v_\parallel)_* = 12\sqrt{a_-}\epsilon/N'$ is unstable in the C_{2v} model [black dot in (a)], but stable in the C_{4v} model [red dot in (b)]. In the C_{2v} model, there is a nontrivial stable fixed point at $(v_\parallel^*, v_\perp^*) = (0.1611, 0.5942)$, with $g_*^2 = 0.2123\epsilon$ for $N' = 4$ [red dot in (a)]. For visualization purposes, we have fixed the ratio $g^2/(v_\perp v_\parallel)$ to its value at the respective stable fixed point (red dots) in these plots.

To make further contact with the QMC data displayed in Fig. 2.14(e), we show in Fig. A2(a,b) the Fermi velocity ratio $v_\perp/v_\parallel$ as function of RG scale $1/b$ in the two models, assuming an isotropic ratio $v_\perp/v_\parallel = 1$ at the ultraviolet scale $b = 1$, for different initial values of the interaction parameter $g^2/(v_\parallel v_\perp)$. We emphasize that a sizable deviation between the two models is observable only at very low energies $1/b \lesssim 0.01$, while the RG flows in the high-energy regime are very similar for the employed starting values. Identifying the RG energy scale $1/b$ roughly with the inverse lattice size $1/L$, this result explains why the lattice sizes available in our simulations are too small to detect a substantial difference in the finite-size scaling of $v_\perp/v_\parallel$. This also implies that the estimates for the critical exponent obtained from the finite-size analysis of the QMC data describes only an intermediate regime, in which the RG flow is not yet fully integrated out. Let us illustrate this point further for the case of the C_{4v} model. In this case, we can define a scale-dependent *effective* correlation-length exponent by using the scaling relation

$$1/\nu_{\text{eff}}(b) = 2 - \eta_\phi^{\text{eff}}(b), \tag{1.6}$$

where $\eta_\phi^{\text{eff}}(b) = N' g^2(b)/[12 v_\perp(b) v_\perp(b)]$ is the effective boson anomalous dimension. This relation becomes exact in the vicinity of the C_{4v} fixed point, for which $\lambda^* = 0$. The effective correlation-length exponent $1/\nu_{\text{eff}}$ is plotted as function of the RG scale $1/b$ in Fig. A2(c) for different values of the initial interaction parameter $g^2/(v_\parallel v_\perp)$. We note that the approach to $\nu_{\text{eff}} \to 1$ in the deep infrared is extremely slow, with sizable deviations from the fixed-point value at intermediate scales. Interestingly, while the behavior in the high-energy regime $1/b \gtrsim 0.05$ is nonuniversal and strongly depends on the particular starting values of the RG flow, a quasiuniversal regime emerges at intermediate energy $1/b \lesssim 0.05$, in which the exponents still drift, but have only a very weak dependence on the initial interaction parameters. This quasiuniversal behavior is a characteristic feature of systems with marginal or close-to-marginal operators [65, 66]. Here, it arises from the slow flow of the velocity anisotropy ratio $v_\perp/v_\parallel$, which implies that the effective exponents will become functions of $v_\perp/v_\parallel$ only, but not of the ultraviolet starting values of the interaction parameters. The quasiuniversality reflects the fact that there is only *one* slowly decaying perturbation to the fixed point (i.e., the leading irrelevant operator), whereas all other perturbations decay quickly, and hence have died out once $1/b \lesssim 0.05$. Importantly, the largest lattice sizes available in the QMC simulations appear to be just large enough to approach the quasiuniversal regime, if we again identify $1/b$ roughly with $1/L$. Reassuringly, for $L = 20$, we therewith obtain the RG estimate $1/\nu_{\text{eff}} \simeq 1.20 \ldots 1.25$, which is in the same ballpark as the estimate from the finite-size scaling analysis of the QMC data discussed in the main text.

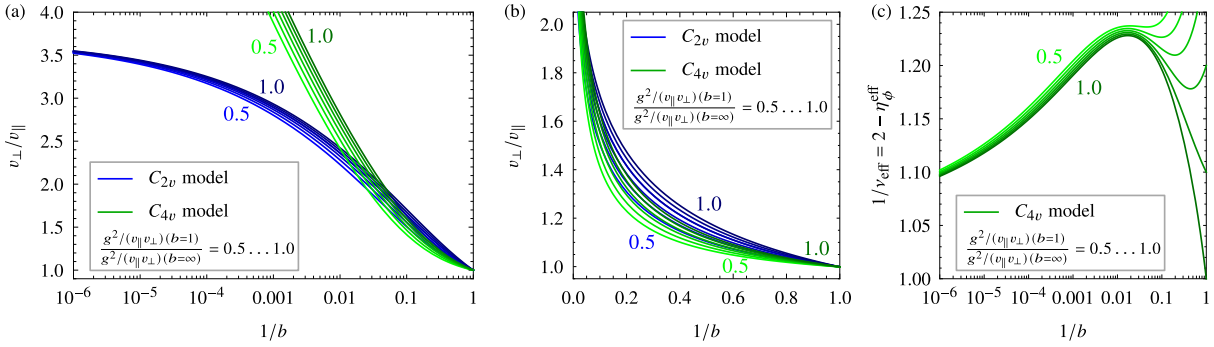


Fig. A2: (a,b) Ratio of Fermi velocities $v_\perp/v_\parallel$ as function of RG scale $1/b$ in the C_{2v} model (blue) and C_{4v} model (green) for different starting values of the interaction parameter $g^2/(v_\parallel v_\perp)$ at the ultraviolet scale $b = 1$. Here, we have numerically integrated out the full RG flow in the $(v_\parallel, v_\perp, g^2)$ parameter space, assuming initial velocities $v_\parallel(b = 1) = v_\perp(b = 1) = 0.25$, and $g^2/(v_\parallel v_\perp)(b = 1)$ between 50% and 100% of the value at the respective stable fixed point. (a) Semilogarithmic plot, demonstrating the finite infrared anisotropy in the C_{2v} model and the logarithmic divergence in the C_{4v} model. (b) Same data as in (a), but using a linear plot, illustrating the similarity of the anisotropy flows in the two models on the high-energy scale, to be compared with the QMC data shown in Fig. 2.14. (c) Effective correlation-length exponent $1/\nu_{\text{eff}}$ as function of RG scale $1/b$ in the C_{4v} model in semilogarithmic plot, defined according to Eq. (1.6), illustrating the drifting of the exponents and the quasiuniversal behavior for $1/b \lesssim 0.05$. We have used the same ultraviolet starting values as in (a,b).

A.2 pyALF Example

This section shows, in a concise form, how to use pyALF to get some of the results shown in Chapter 2, by demonstrations on a small amount of data. It is split in two parts: The first part consists of running ALF simulations, it is designed to run a total of approximately 26 hours on a quad-core machine. The second part showcases some postprocessing of the produced data, e.g. through data collapse.

A.2.1 Running ALF

```
from pprint import pprint  # Pretty print
from py_alf import ALF_source, Simulation  # Interface with ALF
```

A.2.1.1 ALF_source

Create instance of `ALF_source`, choosing to checkout the git branch `'211-add-nematic-dirac-hamiltonian'`, since the Nematic Dirac Hamiltonian is not on master.

Reminder: Directory containing the ALF code is taken from environment variable `$ALF_DIR`, if present.

```
alf_src = ALF_source(
    branch='211-add-nematic-dirac-hamiltonian',
)
```

```
Checking out branch 211-add-nematic-dirac-hamiltonian
Your branch is up to date with 'origin/211-add-nematic-dirac-hamiltonian'.
```

Check available Hamiltonians:

```
alf_src.get_ham_names()
```

```
['Kondo',
 'Hubbard',
 'Hubbard_Plain_Vanilla',
```

(continues on next page)

(continued from previous page)

```
 'tV',
 'LRC',
 'Z2_Matter',
 'Nematic_Dirac']
```

Print valid parameters and their defaults for Nematic Dirac Hamiltonian:

```
pprint(alf_src.get_default_params('Nematic_Dirac', include_generic=False))
```

```
OrderedDict([('VAR_Nematic_Dirac',
              {'Global_J': {'comment': 'J for proposing global updates',
                            'defined_in_base': False,
                            'value': 1.0},
               'Global_h': {'comment': 'h for proposing global updates',
                            'defined_in_base': False,
                            'value': 3.0},
               'Global_type': {'comment': 'Type of global update. Possible '
                                          "values: 'Wolff', 'Geo', 'switch', "
                                          "'flip'",
                               'defined_in_base': False,
                               'value': ''},
               'Ham_J': {'comment': 'Ferromagnetic Ising interaction',
                         'defined_in_base': False,
                         'value': 1.0},
               'Ham_chem': {'comment': 'Chemical potential',
                            'defined_in_base': False,
                            'value': 0.0},
               'Ham_h': {'comment': 'Ising transverse field',
                         'defined_in_base': False,
                         'value': 3.0},
               'Ham_xi': {'comment': 'Coupling strength Ising spins <-> '
                                     'fermions',
                          'defined_in_base': False,
                          'value': 1.0},
               'Ham_xi2': {'comment': 'Static fermion hopping "distortion"',
                           'defined_in_base': False,
                           'value': 0.0},
               'L1': {'comment': 'Size of lattice in a1 direction',
                      'defined_in_base': False,
                      'value': 4},
               'L2': {'comment': 'Size of lattice in a2 direction',
                      'defined_in_base': False,
                      'value': 4},
               'Model_vers': {'comment': 'Version of model. 1: C_2v model, 2: '
                                         'C_4v model',
                              'defined_in_base': False,
                              'value': 1},
               'N_SUN': {'comment': 'SU(N) symmetry',
                         'defined_in_base': True,
                         'value': 2},
               'Phi_1': {'comment': 'Twisted boundary in a1 direction',
                         'defined_in_base': False,
                         'value': 0.0},
               'Phi_2': {'comment': 'Twisted boundary in a2 direction',
                         'defined_in_base': False,
                         'value': 0.0},
               'beta': {'comment': 'Reciprocal temperature',
                        'defined_in_base': False,
                        'value': 10.0},
               'dtau': {'comment': 'Imaginary time step size',
                        'defined_in_base': False,
```

(continues on next page)

(continued from previous page)

```
                'value': 0.1},
 'ham_t': {'comment': 'Hopping amplitude of fermions',
           'defined_in_base': False,
           'value': 1.0},
 'init_type': {'comment': 'How to initialize Ising field. '
                          "Possible values: 'random', 'up', "
                          "'down', 'updown'",
               'defined_in_base': False,
               'value': 'random'}})])
```

A.2.1.2 Perform simulations

The loop shown below performs a parameter sweep over transverse field strength $h \in \{2.5, 3.0, 3.5, 4.0\}$ for system sizes $L \in \{4, 6, 8, 10\}$. Parallel Tempering is used to run simulations with different h in parallel. This method is actually intended for addressing ergodicity issues, but in this case it is only used to more conveniently perform a parameter sweep in parallel.

The simulations are set to take 0.2, 1, 7 and 18 hours.

```
for L, time in zip([4, 6, 8, 10], [.2, 1., 7., 18.]):
    print(f'====== L={L} ======')
    sim = Simulation(
        alf_src,
        'Nematic_Dirac',
        [{
            # Model specific parameters
            'Model_vers': 1,  # C_2v model
            'L1': L,
            'L2': L,
            'beta': L*4.,
            'Ham_xi': 0.25,
            'Ham_h': h,
            # QMC parameters
            'Ltau': 1,
            'CPU_MAX': time,
            'NSweep': 20,
            'mpi_per_parameter_set': 1,
            # Only put Tempering_calc_det=False if you know what you're doing.
            'Tempering_calc_det': False,
        } for h in [2.5, 3.0, 3.5, 4.0]],
        machine='intel',
        mpi=True,
        n_mpi=4,
    )
    if L == 4:
        # We only need to compile once
        sim.compile()
    sim.run()
    sim.print_info_file()
```

The text printed through these simulations is very long and has therefore been hidden, but it can be viewed in the website version[1] of this document.

The info files produced by ALF show a "Precision Green" and "Precision Phase" of order 10^{-14} which is very good and much smaller than 10^{-8}. In fact one might consider increasing the stabilization interval `Nwrap` to speed up the simulation.

The produced QMC data will be postprocessed in the next section.

[1] https://purl.org/diss-jschwab

A.2.1.3 Prepare directories for simulation

pyALF can also be use to prepare simulation directories without executing ALF, for example to copy the prepared directories to another machine or execute ALF in another way like a scheduler.

The following example prepares the directory `for_hpc` with subdirectories `L{L}/h{h}`.

```
for L in [4, 6, 8, 10]:
    for h in [2.5, 3.0, 3.5, 4.0]:
        sim = Simulation(
            alf_src,
            'Nematic_Dirac',
            {
                # Model specific parameters
                'Model_vers': 1,  # C_2v model
                'L1': L,
                'L2': L,
                'beta': L*4.,
                'Ham_xi': 0.25,
                'Ham_h': h,
                # QMC parameters
                'Ltau': 1,
                'CPU_MAX': 24,
                'NSweep': 20,
                'mpi_per_parameter_set': 1,
                # Only put Tempering_calc_det=False if you know what you're doing.
                'Tempering_calc_det': False,
            },
            machine='intel',
            mpi=True,
            n_mpi=4,
            sim_root='for_hpc',
            sim_dir=f'L{L}/h{h}'
        )
        sim.run(only_prep=True)
```

```
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L4/h2.5" for Monte Carlo run.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L4/h3.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L4/h3.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L4/h4.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L6/h2.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L6/h3.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L6/h3.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L6/h4.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L8/h2.5" for Monte Carlo run.
Create new directory.
```

(continues on next page)

(continued from previous page)

```
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L8/h3.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L8/h3.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L8/h4.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L10/h2.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L10/h3.0" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L10/h3.5" for Monte Carlo run.
Create new directory.
Prepare directory "/home/jonas/dissertation/jb/appendix_nematic_pyalf/for_hpc/
 ↪L10/h4.0" for Monte Carlo run.
Create new directory.
```

```
!tree for_hpc
```

```
for_hpc
├── L10
│   ├── h2.5
│   │   ├── parameters
│   │   └── seeds
│   ├── h3.0
│   │   ├── parameters
│   │   └── seeds
│   ├── h3.5
│   │   ├── parameters
│   │   └── seeds
│   └── h4.0
│       ├── parameters
│       └── seeds
├── L4
│   ├── h2.5
│   │   ├── parameters
│   │   └── seeds
│   ├── h3.0
│   │   ├── parameters
│   │   └── seeds
│   ├── h3.5
│   │   ├── parameters
│   │   └── seeds
│   └── h4.0
│       ├── parameters
│       └── seeds
├── L6
│   ├── h2.5
│   │   ├── parameters
│   │   └── seeds
│   ├── h3.0
│   │   ├── parameters
│   │   └── seeds
│   ├── h3.5
│   │   ├── parameters
```

(continues on next page)

(continued from previous page)

```
│   │   └── seeds
│   └── h4.0
│       ├── parameters
│       └── seeds
└── L8
    ├── h2.5
    │   ├── parameters
    │   └── seeds
    ├── h3.0
    │   ├── parameters
    │   └── seeds
    ├── h3.5
    │   ├── parameters
    │   └── seeds
    └── h4.0
        ├── parameters
        └── seeds

20 directories, 32 files
```

A.2.2 Postprocessing

```
# Enable Jupyter Widget support for matplotlib.
%matplotlib widget

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

A.2.2.1 Find QMC data

The variable `dirs` is a list of all directories containing an ALF results file `data.h5`.

```
from py_alf.utils import find_sim_dirs
dirs = find_sim_dirs()
dirs
```

```
['./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_
↪h=2.5/Temp_0',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_
↪h=2.5/Temp_1',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_
↪h=2.5/Temp_2',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_
↪h=2.5/Temp_3',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.
↪5/Temp_0',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.
↪5/Temp_1',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.
↪5/Temp_2',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.
↪5/Temp_3',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.
↪5/Temp_0',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.
↪5/Temp_1',
```

(continues on next page)

(continued from previous page)

```
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.
↪5/Temp_2',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.
↪5/Temp_3',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.
↪5/Temp_0',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.
↪5/Temp_1',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.
↪5/Temp_2',
'./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.
↪5/Temp_3']
```

A.2.2.2 Custom observables

Defining some observables derived from measurements done during the simulations (cf. Section 4.2.3.2).

Correlation ratio $1 - \frac{O(\boldsymbol{k}+\boldsymbol{\delta})}{O(\boldsymbol{k})}$. RG-invariant quantity

```
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)]):
    """RG-invariant quantity derived from a correlation function.

    obs.shape = (N_orb, N_orb, N_tau, latt.N)
    back.shape = (N_orb,)
    """
    if mat is None:
        mat = np.identity(N_orb, dtype=np.double)
    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)
```

Binder cumulant $\left(3 - \frac{\langle s^4 \rangle}{\langle s^2 \rangle^2}\right)/2$. RG-invariant quantity

```
def binder(obs, sign, N_obs):
    return (3 - obs[2] / obs[1]**2 * sign)/2
```

Susceptibility $\int_0^\beta \mathrm{d}\tau\, C\left(\boldsymbol{k}, \tau\right)$

```
def susceptibility(obs, back, sign, N_orb, N_tau, dtau, latt,
                   ks=[[0., 0.]]):
    """Susceptibility of a time-displaced correlation function.

    obs.shape = (N_orb, N_orb, N_tau, latt.N)
    back.shape = (N_orb,)
    """
    out = 0.
    for k in ks:
        n = latt.k_to_n(k)
        out += dtau*obs[..., n].trace().sum() / sign
    return out / len(ks)
```

Fermionic single particle gap determined through a quick and dirty fit of the time-displaced Green function (cf. Section 2.7.2.1).

```
from scipy.optimize import curve_fit

def fit_gap_lazy(obs, back, sign, N_orb, N_tau, dtau, latt):
    """Lazily fit Green function to determin gap.

    obs.shape = (N_orb, N_orb, N_tau, latt.N)
    """
    i1 = (2*N_tau)//24
    i2 = (4*N_tau)//24
    green = obs[0, 0, i1:i2, :]/sign

    def func(x, a, b):
        return a*np.exp(-b*x)
    taus = np.arange(0., N_tau*dtau, dtau)
    res = np.empty((latt.N,), dtype=np.cdouble)
    for n in range(latt.N):
        popt, pcov = curve_fit(func, taus[i1:i2], green[:, n])
        res[n] = popt[1]

    return res
```

```
custom_obs = {}
```

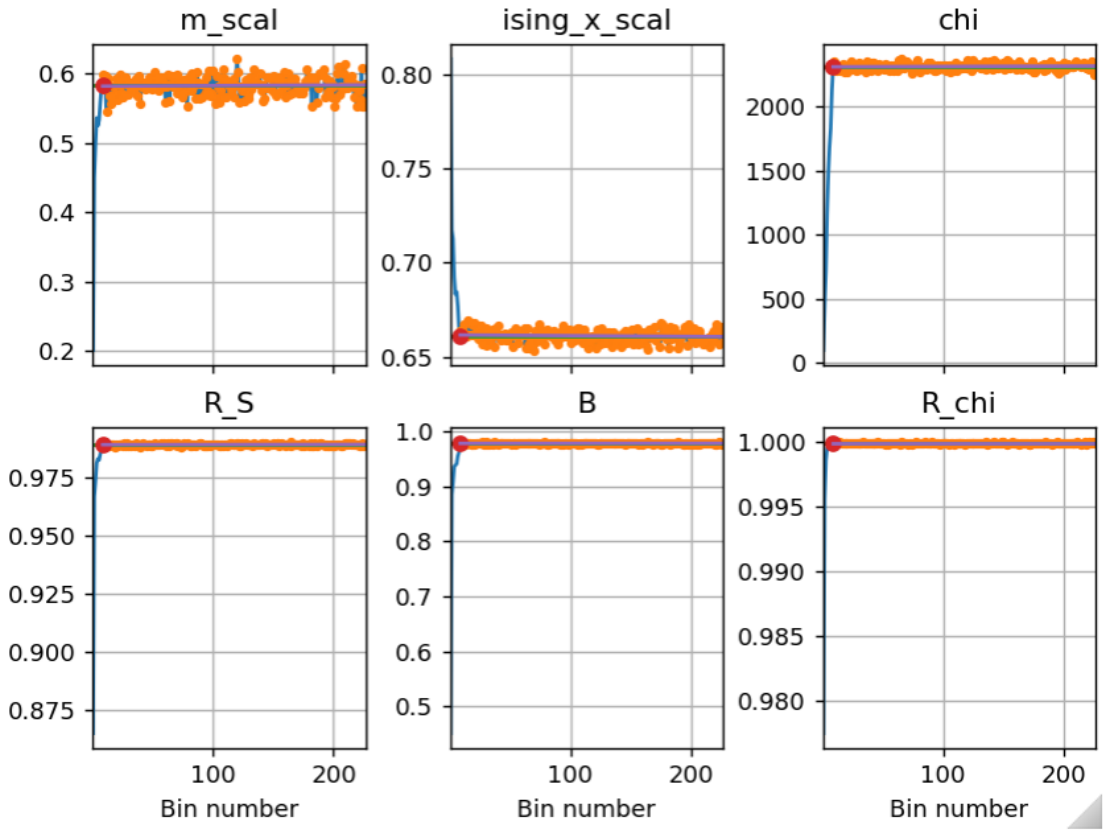
```
# Structure factor correlation ratio
custom_obs['R_S']= {
    'needs': ['IsingZ_eq'],
    'function': R_k,
    'kwargs': {}
}

# Susceptibility correlation ratio
custom_obs['R_chi']= {
    'needs': ['IsingZT_tau'],
    'function': R_k,
    'kwargs': {}
}

# Binder cumulant
custom_obs['B']= {
    'needs': ['m_scal'],
    'function': binder,
    'kwargs': {}
}

# Susceptibility
custom_obs['chi']= {
    'needs': ['IsingZT_tau'],
    'function': susceptibility,
    'kwargs': {}
}

# Susceptibility
custom_obs['gap_lazy']= {
    'needs': ['Green_tau'],
    'function': fit_gap_lazy,
    'kwargs': {}
}
```

A.2.2.3 Check warmup and autocorrelation times

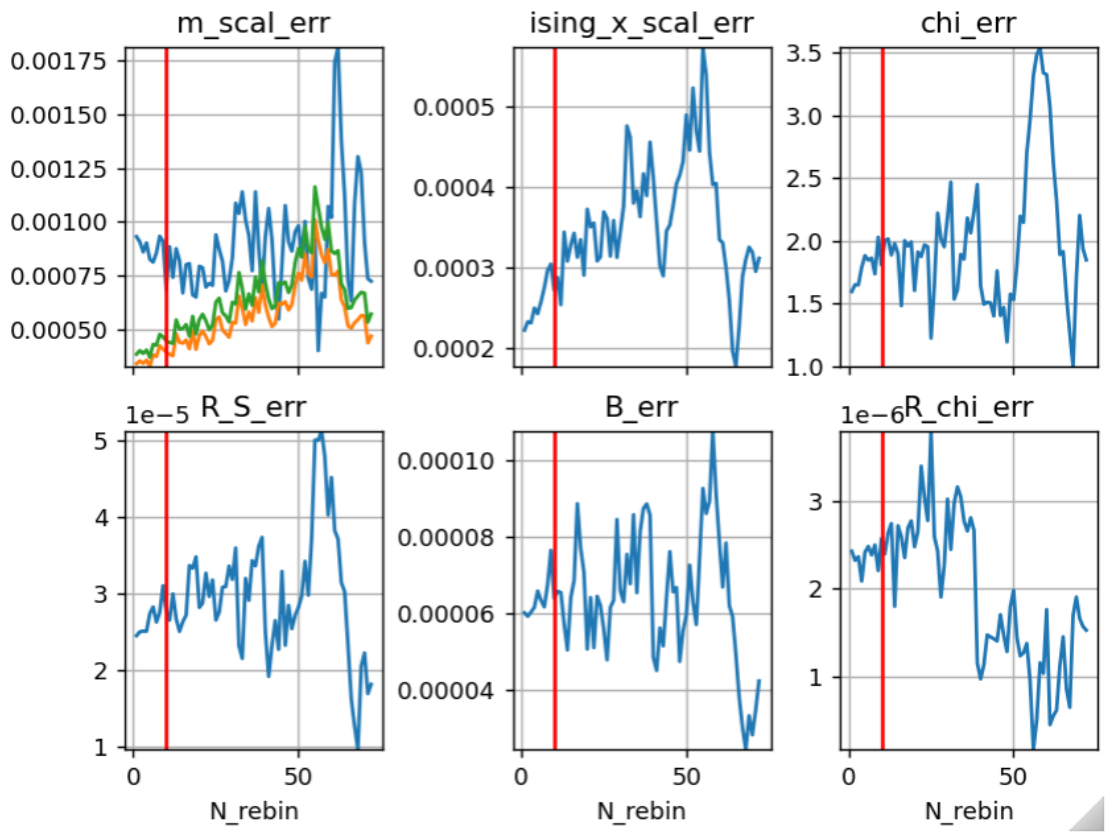
C.f. Section 4.2.3.3.

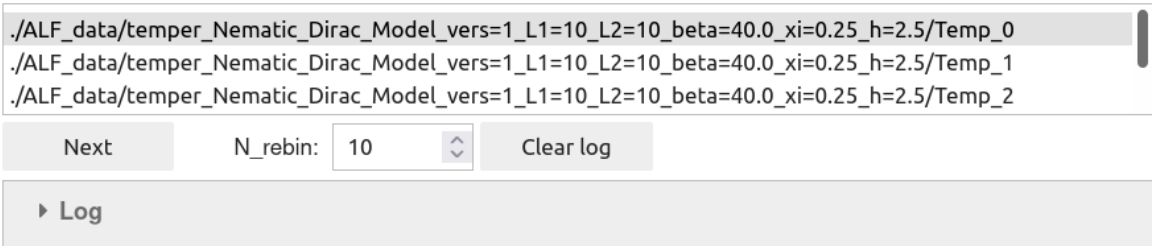
```
from py_alf import check_warmup, check_rebin
```

```
check_warmup(dirs, ['m_scal', 'ising_x_scal', 'chi', 'R_S', 'B', 'R_chi'],
             custom_obs=custom_obs, gui='ipy')
```




```
check_rebin(dirs, ['m_scal', 'ising_x_scal', 'chi', 'R_S', 'B', 'R_chi'],
            custom_obs=custom_obs, gui='ipy')
```





A.2.2.4 Error analysis

The analysis results are saved in each simulation directory, both in plain text in the folder `res` and as a `pickled`[2] Python dictionary in the file `res.pkl`.

Due to its length, the text printed out during the analysis is hidden, but can be viewed in the website version[3] of this document.

```
from py_alf.analysis import analysis
for directory in dirs:
    analysis(directory, custom_obs=custom_obs, always=True)
```

[2] https://docs.python.org/3/library/pickle.html#module-pickle
[3] https://purl.org/diss-jschwab

A.2.2.5 Read analysis results

Read all the `res.pkl` files and combine them in a single `pandas DataFrame`[4], called `res`.

```
from py_alf.ana import load_res
res = load_res(dirs)
```

```
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
 ↪5/Temp_0
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
 ↪5/Temp_1
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
 ↪5/Temp_2
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
 ↪5/Temp_3
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
 ↪Temp_0
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
 ↪Temp_1
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
 ↪Temp_2
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
 ↪Temp_3
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
 ↪Temp_0
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
 ↪Temp_1
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
 ↪Temp_2
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
 ↪Temp_3
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
 ↪Temp_0
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
 ↪Temp_1
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
 ↪Temp_2
No orbital locations saved.
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
 ↪Temp_3
No orbital locations saved.
```

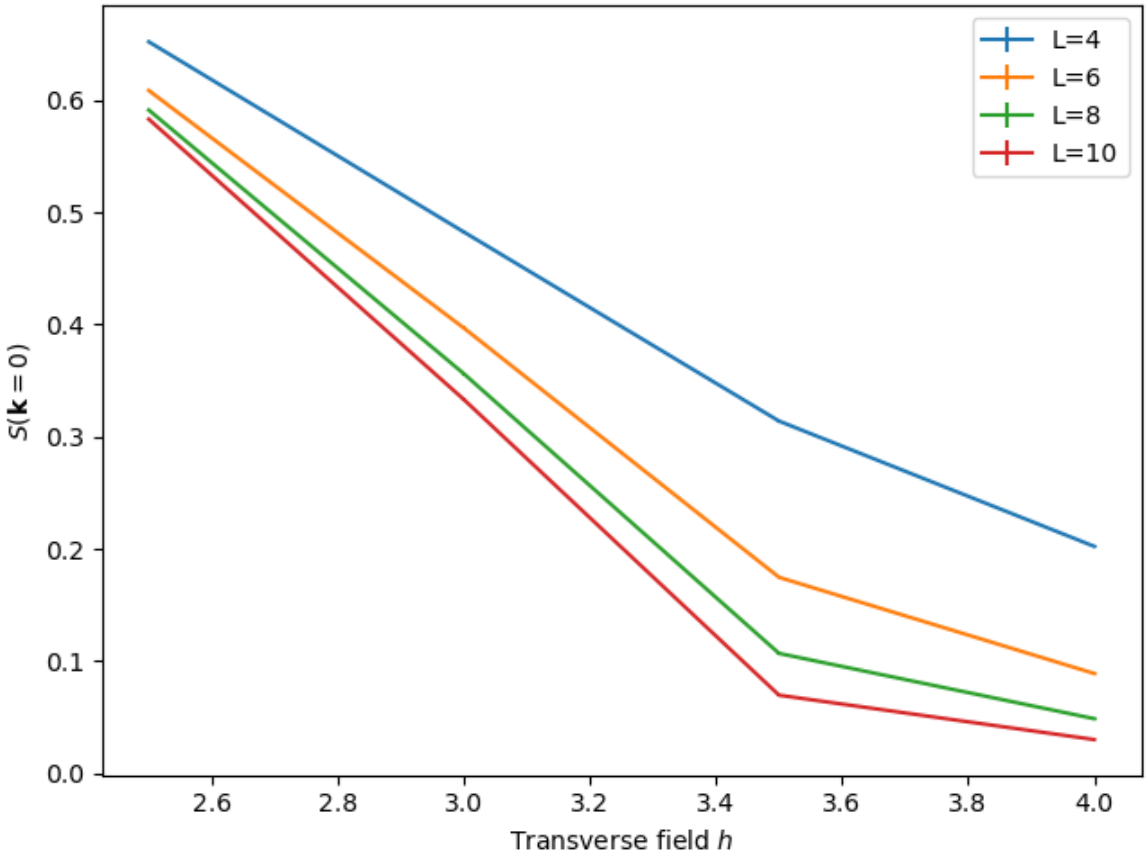
The printout of the following command is again hidden, but can be viewed in the website version[5] of this document.

```
res
```

[4] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html#pandas.DataFrame
[5] https://purl.org/diss-jschwab

A.2.2.6 Plot order parameter

```
fig, ax = plt.subplots(constrained_layout=True)
for L in [4, 6, 8, 10]:
    df = res[res.l1 == L].sort_values(by='ham_h')
    ax.errorbar(df.ham_h, df.m_scal0, df.m_scal0_err, label=f'L={L}')
ax.legend()
ax.set_xlabel('Transverse field $h$')
ax.set_ylabel('$S(\boldsymbol{k}=0)$');
```



A.2.2.7 Plot RG-invariant quantities

RG-invariant quantities behave at critical point as:

$$R = f(L^z/\beta, (h - h_c)L^{1/\nu}, L^{-\omega})$$

Dismissing dependence on β and finite size corrections:

$$R = f((h - h_c)L^{1/\nu}) \tag{1.7}$$

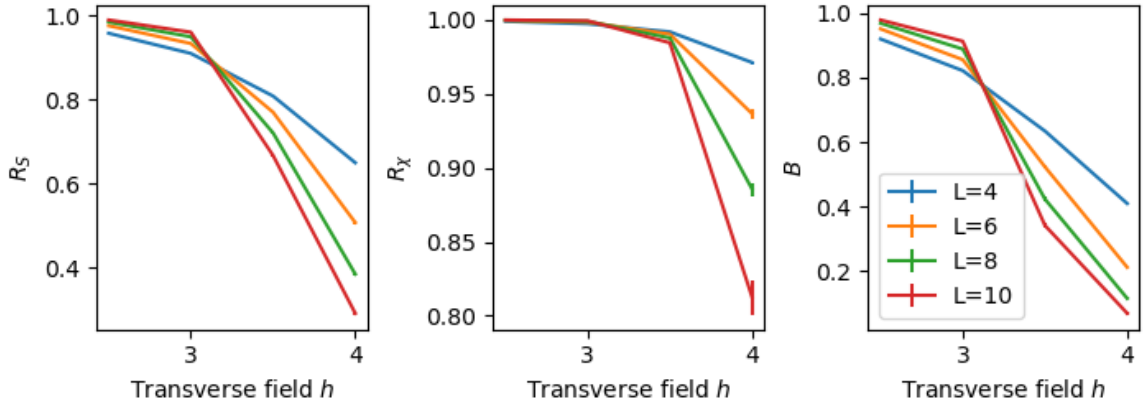
Therefore, they should cross for different system sizes at $h = h_c$.

```
fig, axs = plt.subplots(1, 3, constrained_layout=True, figsize=(7, 2.5))
for L in [4, 6, 8, 10]:
    df = res[res.l1 == L].sort_values(by='ham_h')
    for obs_name, ylabel, ax in zip(['R_S', 'R_chi', 'B'],
                                    [r'$R_S$', r'$R_\chi$', r'$B$'],
                                    axs):
        ax.errorbar(
            df.ham_h, df[obs_name], df[obs_name+'_err'], label=f'L={L}')
```

(continues on next page)

(continued from previous page)

```
        ax.set_xlabel('Transverse field $h$')
        ax.set_ylabel(ylabel)
ax.legend();
```



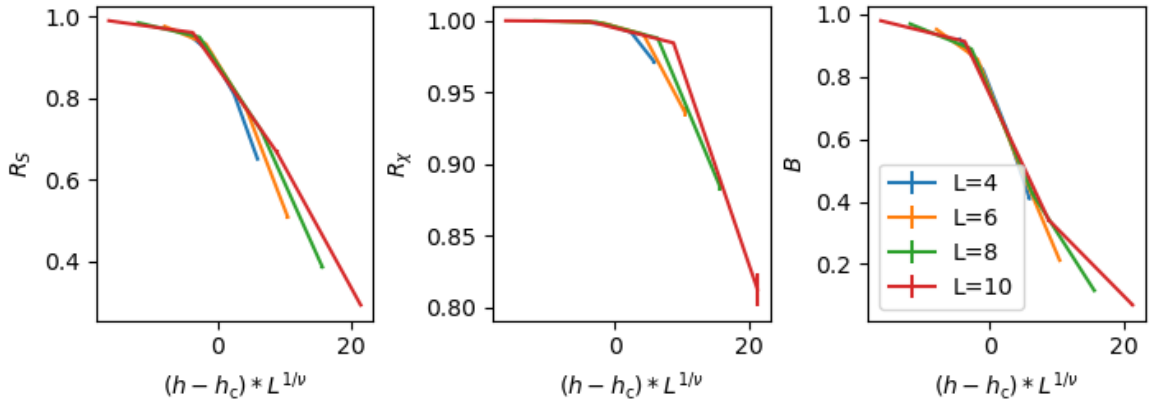
A.2.2.8 Data collapse

I demonstrate here briefly how one can perform a data collapse. The amount of data is too little and the system sizes too small to get meaningful results, but the overall approach can still be presented well.

A.2.2.8.1 Manual data collapse

The first step is to manually vary the parameters, in this case h_c and $a = 1/\nu$, to collapse the data. The results can then be used as starting parameters of the automatic data collapse in the next step.

```
hc = 3.15
a = 1.4  # a=1/nu
fig, axs = plt.subplots(1, 3, constrained_layout=True, figsize=(7, 2.5))
for L in [4, 6, 8, 10]:
    df = res[res.l1 == L].sort_values(by='ham_h')
    for obs_name, ylabel, ax in zip(['R_S', 'R_chi', 'B'],
                                    [r'$R_S$', r'$R_\chi$', r'$B$'],
                                    axs):
        ax.errorbar((df.ham_h-hc)*L**a, df[obs_name],
                    df[obs_name+'_err'], label=f'L={L}')
        ax.set_xlabel(r'$(h-h_{\rm c})*L^{1/\nu}$')
        ax.set_ylabel(ylabel)
ax.legend();
```



A.2.2.8.2 Data collapse fit

See *Appendix A.3* for the source code of `collapse`.

```
from collapse import collapse
```

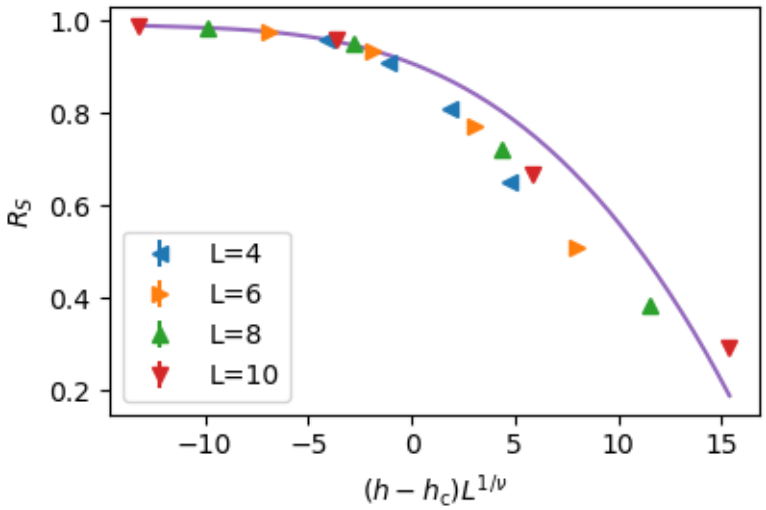
The function `func` defines the scaling assumption of Eq. (1.7).

```
def func(L, x, dx, y, dy, par):
    """Scaling assumption of RG-invariant quantities
    without corrections to scaling."""
    xc = par[0]
    a = par[1]  # a=1/nu
    x_scaled = (x-xc) * L**a
    if dx is None:
        dx_scaled = None
    else:
        dx_scaled = (x-xc) * L**a
    y_scaled = y
    dy_scaled = dy
    return x_scaled, dx_scaled, y_scaled, dy_scaled
```

Performing automatic data collapse of R_S for system sizes $L \in \{4, 6, 8, 10\}$ and starting parameters $h_c = 3.15$, $1/\nu = 1.4$ from the manual data collapse.

```
fig = plt.figure(constrained_layout=True, figsize=(4, 2.7))
plt.xlabel(r'$(h-h_{\rm c})L^{1/\nu}$')
plt.ylabel(r'$R_S$')
collapse(func, 'ham_h', 'R_S', res, Ls=[4, 6, 8, 10] , par0=[3.15, 1.3])
```

```
{'Ls': [4, 6, 8, 10],
 'L0': 4,
 'NL': 4,
 'popt': array([ 3.19239096e+00,  1.28013715e+00,  9.08264521e-01, -1.67682563e-
↪02,
        -1.24065522e-03, -3.29972113e-05]),
 'perr': array([8.89780198e-03, 4.91240345e-03, 2.45520108e-03, 6.27747079e-04,
        5.22775205e-05, 1.42967090e-06]),
 'S': 1136.7062136799598}
```



The resulting data collapse does not look very good and the quality of fit function $S \approx 1500$, which should be of order 1, is much too big.

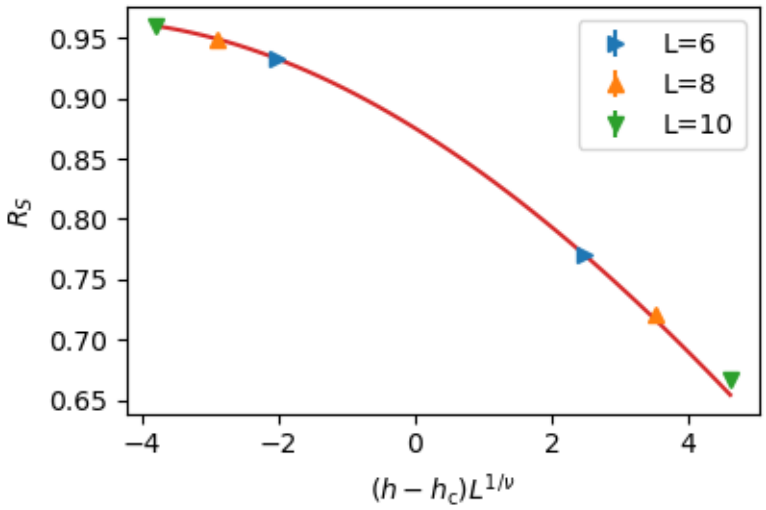
Some likely reasons for this are:

1. Too small system sizes.
2. Too few data points.
3. Too large fitting range, meaning to big values of $\left|(h-h_\mathrm{c})L^{1/\nu}\right|$.

We dismiss system sizes $L = 4$ and restrict the data to $(h-h_{\mathrm{c},0})L^{1/\nu_0} \in [-5, 10]$. This leads to a much more agreeable, but still too large $S \approx 24$. Furthermore, there are only 6 data points left and the result $1/\nu = 1.2 \pm 0.1$ is not in agreement with Section 2.8. Nevertheless, this might be the best that can be done with the available data in terms of data collapses.

```
hc0 = 3.15
a0 = 1.3
df = res[(res['ham_h']-hc)*res.l1**a > -5]
df = df[(df['ham_h']-hc)*df.l1**a < 10]
fig = plt.figure(constrained_layout=True, figsize=(4, 2.7))
plt.xlabel(r'$(h-h_{\rm c})L^{1/\nu}$')
plt.ylabel(r'$R_S$')
collapse(func, 'ham_h', 'R_S', df, Ls=[6, 8, 10] , par0=[hc0, a0])
```

```
{'Ls': [6, 8, 10],
 'L0': 6,
 'NL': 3,
 'popt': array([ 3.22588756e+00,  1.22585025e+00,  8.74205610e-01, -3.71090125e-
↪02,
        -3.65818772e-03,  2.59712699e-05]),
 'perr': array([0.04638255, 0.10616724, 0.0102565 , 0.00745803, 0.00196077,
        0.00023591]),
 'S': 26.630914312767914}
```



A.2.2.9 Plot correlation

A.2.2.9.1 Accessing elements of the dataframe

```
res.columns
```

```
Index(['beta', 'dtau', 'global_h', 'global_j', 'global_type', 'ham_chem',
       'ham_h', 'ham_j', 'ham_t', 'ham_xi',
       ...
       'IsingXT_tauK_err', 'IsingXT_tauR', 'IsingXT_tauR_err',
       'IsingXT_tau_lattice', 'IsingZT_tauK', 'IsingZT_tauK_err',
```

(continues on next page)

(continued from previous page)

```
       'IsingZT_tauR', 'IsingZT_tauR_err', 'IsingZT_tau_lattice', 'lattice'],
      dtype='object', length=128)
```

```
res.index
```

```
Index(['./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.
 ↪25_h=2.5/Temp_0',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.
 ↪25_h=2.5/Temp_1',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.
 ↪25_h=2.5/Temp_2',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.
 ↪25_h=2.5/Temp_3',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.
 ↪25_h=2.5/Temp_0',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.
 ↪25_h=2.5/Temp_1',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.
 ↪25_h=2.5/Temp_2',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.
 ↪25_h=2.5/Temp_3',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.
 ↪25_h=2.5/Temp_0',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.
 ↪25_h=2.5/Temp_1',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.
 ↪25_h=2.5/Temp_2',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.
 ↪25_h=2.5/Temp_3',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.
 ↪25_h=2.5/Temp_0',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.
 ↪25_h=2.5/Temp_1',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.
 ↪25_h=2.5/Temp_2',
       './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.
 ↪25_h=2.5/Temp_3'],
      dtype='object')
```

```
item = res.loc['./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.
 ↪0_xi=0.25_h=2.5/Temp_0']
item
```

```
beta                                                              40.0
dtau                                                               0.1
global_h                                                           3.0
global_j                                                           1.0
global_type                                                        b''
                                          ...
IsingZT_tauK_err       [[0.0008932302580056016, 0.0011906138182334284...
IsingZT_tauR           [[0.5773744618389179, 0.5776504791744279, 0.57...
IsingZT_tauR_err       [[0.0005925670020223004, 0.000599508043614475,...
IsingZT_tau_lattice    {'L1': [7.071067811865475, 7.071067811865475],...
lattice                {'L1': [7.071067811865475, 7.071067811865475],...
Name: ./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.
 ↪25_h=2.5/Temp_0, Length: 128, dtype: object
```

```
item['IsingZ_eqK']
```

```
array([[[ 0.32638655,  0.34595176,  0.37585751,  0.40512372,
          0.41848529,  0.40560553,  0.3758137 ,  0.34585338,
          0.32686855,  0.32000094,  0.34582087,  0.36959162,
          0.40548   ,  0.44284267,  0.4599152 ,  0.44257476,
          0.40533533,  0.36922645,  0.3460411 ,  0.33724708,
          0.37373131,  0.40447458,  0.45249519,  0.50672501,
          0.53239187,  0.50668498,  0.4533181 ,  0.4044087 ,
          0.37437927,  0.36392956,  0.40205248,  0.43922089,
          0.50464484,  0.58204502,  0.62283904,  0.58161175,
          0.50320508,  0.44007743,  0.40266863,  0.38967683,
          0.41550708,  0.45576157,  0.52808854,  0.6214426 ,
         58.38260793,  0.6214426 ,  0.52808854,  0.45576157,
          0.41550708,  0.40164649,  0.40266863,  0.44007743,
          0.50320508,  0.58161175,  0.62283904,  0.58204502,
          0.50464484,  0.43922089,  0.40205248,  0.38967683,
          0.37437927,  0.4044087 ,  0.4533181 ,  0.50668498,
          0.53239187,  0.50672501,  0.45249519,  0.40447458,
          0.37373131,  0.36392956,  0.3460411 ,  0.36922645,
          0.40533533,  0.44257476,  0.4599152 ,  0.44284267,
          0.40548   ,  0.36959162,  0.34582087,  0.33724708,
          0.32686855,  0.34585338,  0.3758137 ,  0.40560553,
          0.41848529,  0.40512372,  0.37585751,  0.34595176,
          0.32638655,  0.32000094,  0.31925968,  0.33890868,
          0.36577437,  0.3931343 ,  0.40544295,  0.3931343 ,
          0.36577437,  0.33890868,  0.31925968,  0.31334392]]])
```

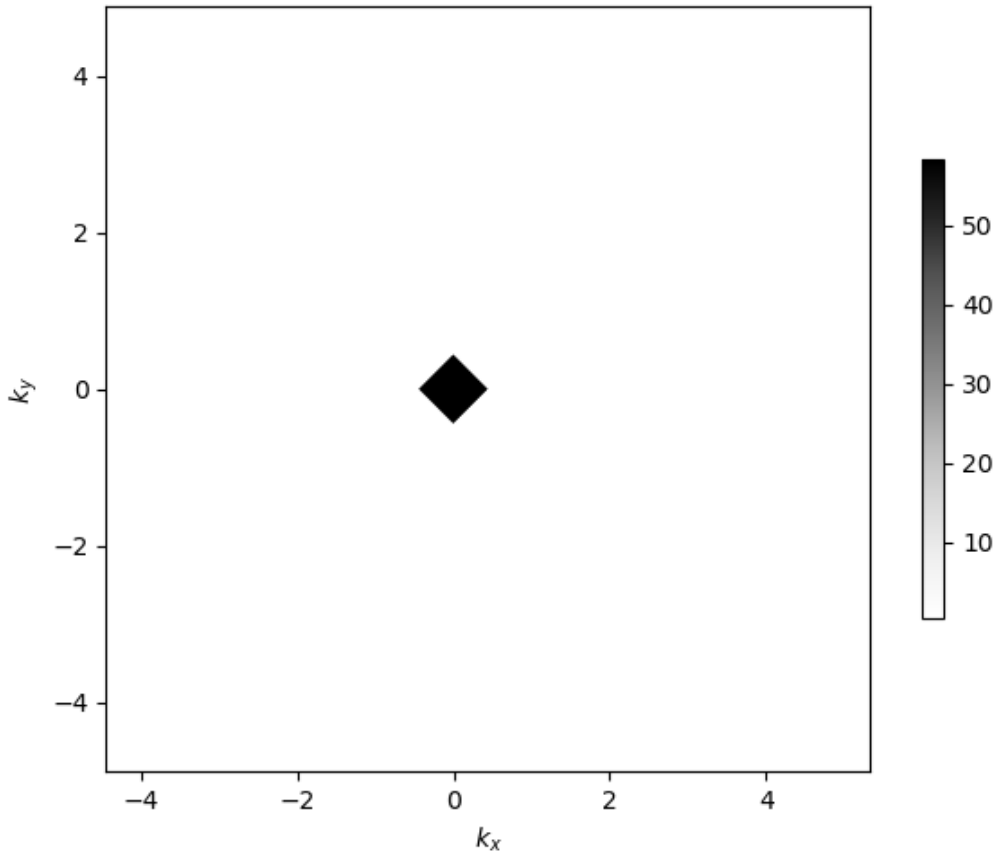
A.2.2.9.2 Creating Lattice object

```
from py_alf import Lattice
```

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

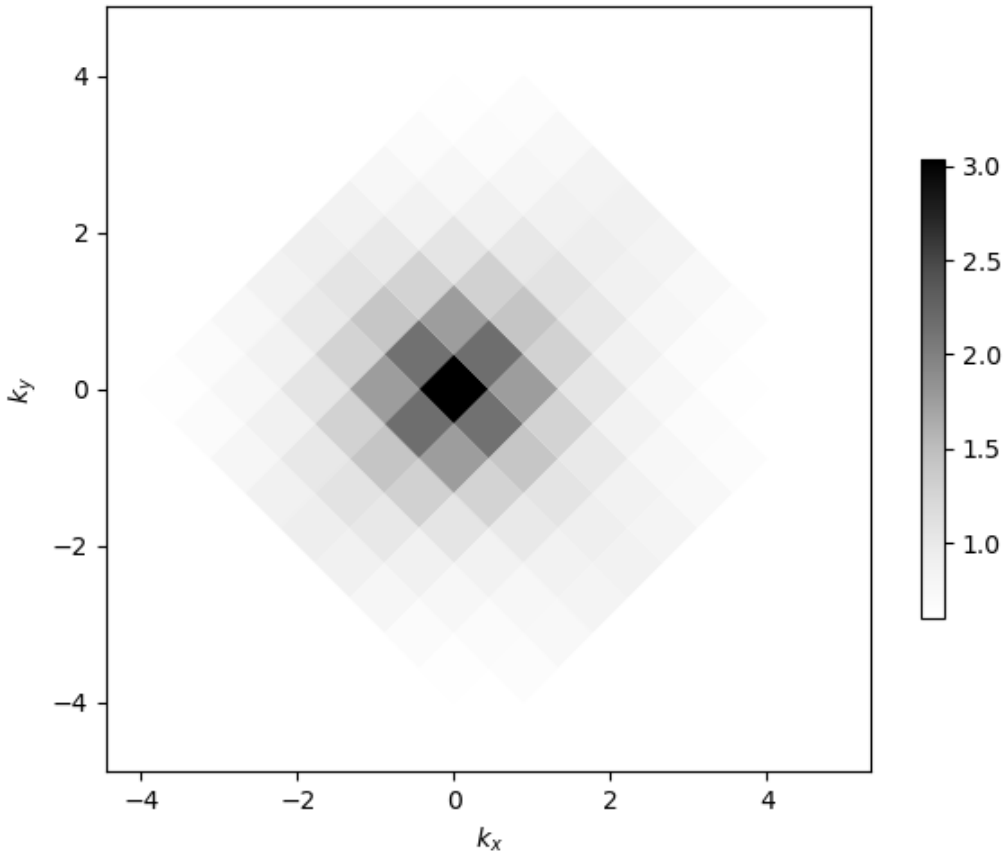
A.2.2.9.3 Spin-Spin correlation deep in ordered phase

```
latt.plot_k(item['IsingZ_eqK'][0,0])
```



A.2.2.9.4 Spin-Spin correlation in disordered phase

```
latt.plot_k(res.loc[
    './ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_
↪h=2.5/Temp_3',
    'IsingZ_eqK'][0,0])
```



A.2.2.10 Fermionic dispersion

See *Appendix A.4* for source code of `fit_green_tau`.

```
from fit_green_tau import fit_green_tau
```

```
dic = {}
for i in res.index:
    print(i)
    dtau = res.loc[i, 'dtau']
    Green = res.loc[i, 'Green_tauK']
    dGreen = res.loc[i, 'Green_tauK_err']
    (N_tau, N) = Green.shape
    taus = np.arange(0., N_tau*dtau, dtau)
    dic[i] = np.empty((N, 2))
    for n in range(N):
        # print(f'{n} out of {N}')
        G = Green[:, n]
        dG = dGreen[:, n]
        dic[i][n] = fit_green_tau(taus, G, dG, plot=False)
res['gap'] = pd.Series(dic)
```

```
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
↪5/Temp_0
```

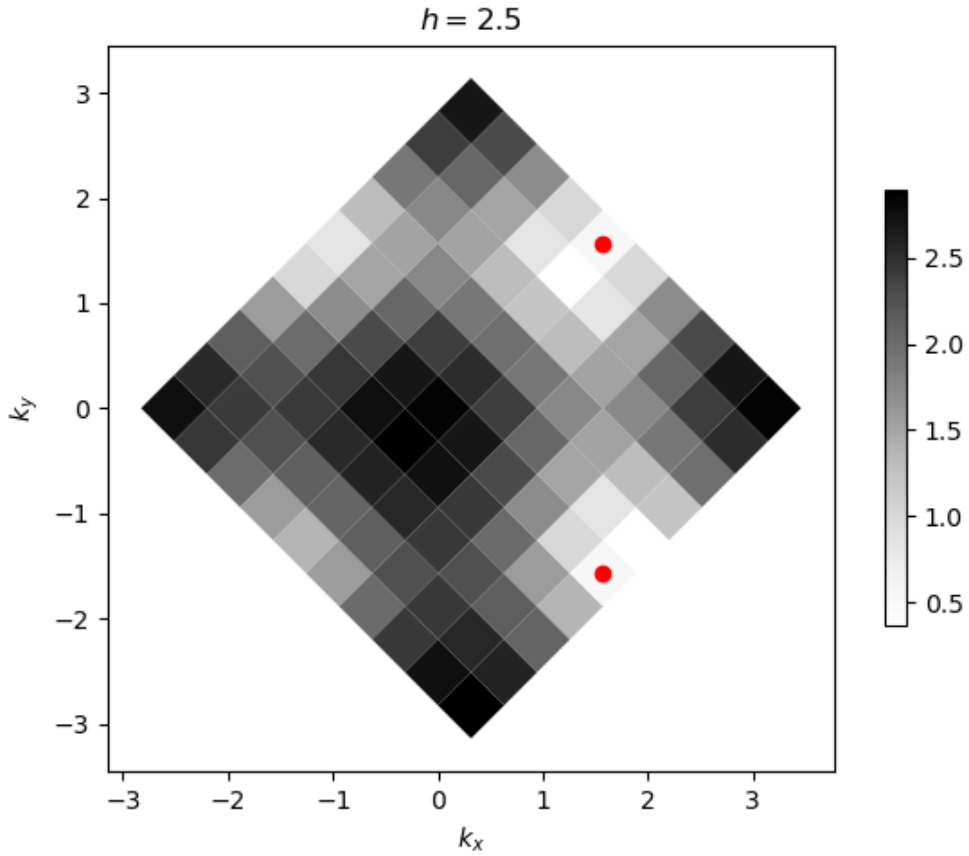
```
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
↪5/Temp_1
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
↪5/Temp_2
```

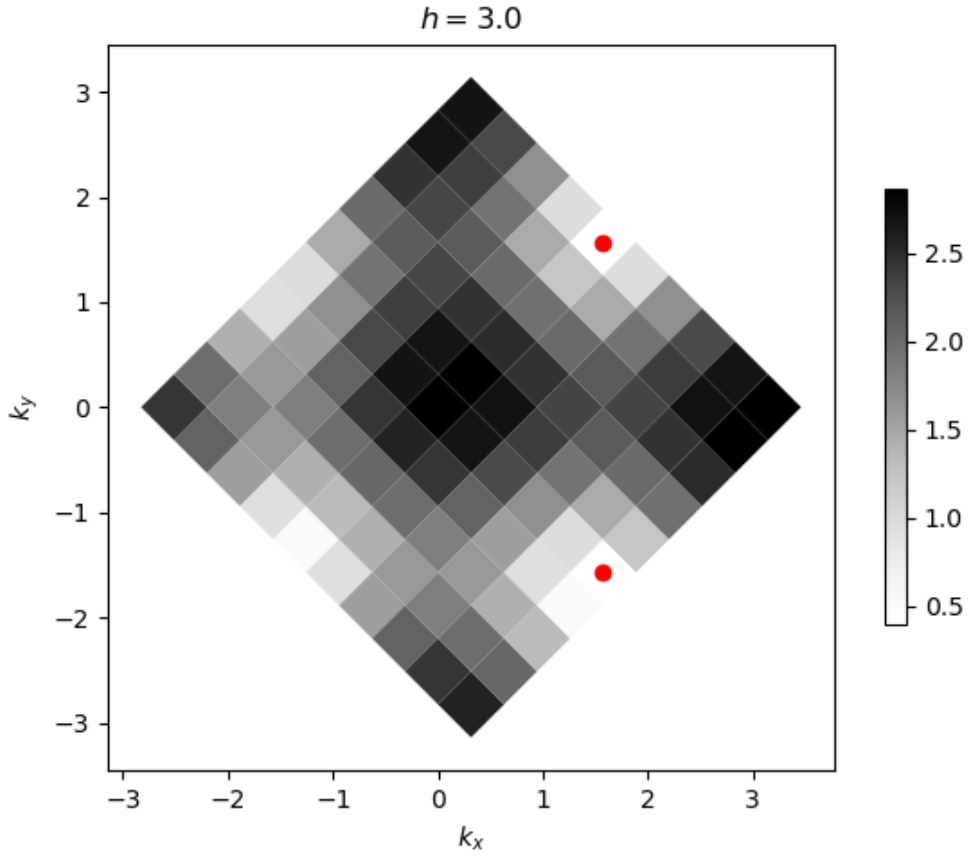
```
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=10_L2=10_beta=40.0_xi=0.25_h=2.
↪5/Temp_3
```

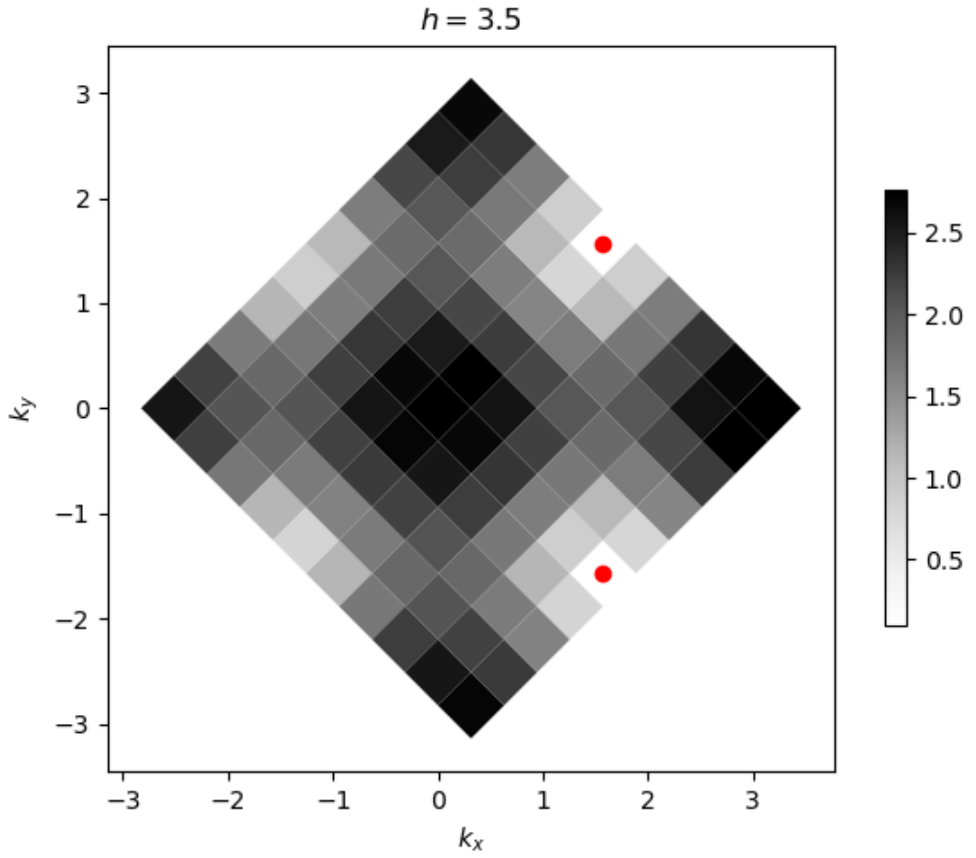
```
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
↪Temp_0
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
↪Temp_1
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
↪Temp_2
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=4_L2=4_beta=16.0_xi=0.25_h=2.5/
↪Temp_3
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
↪Temp_0
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
↪Temp_1
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
↪Temp_2
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=6_L2=6_beta=24.0_xi=0.25_h=2.5/
↪Temp_3
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
↪Temp_0
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
↪Temp_1
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
↪Temp_2
./ALF_data/temper_Nematic_Dirac_Model_vers=1_L1=8_L2=8_beta=32.0_xi=0.25_h=2.5/
↪Temp_3
```

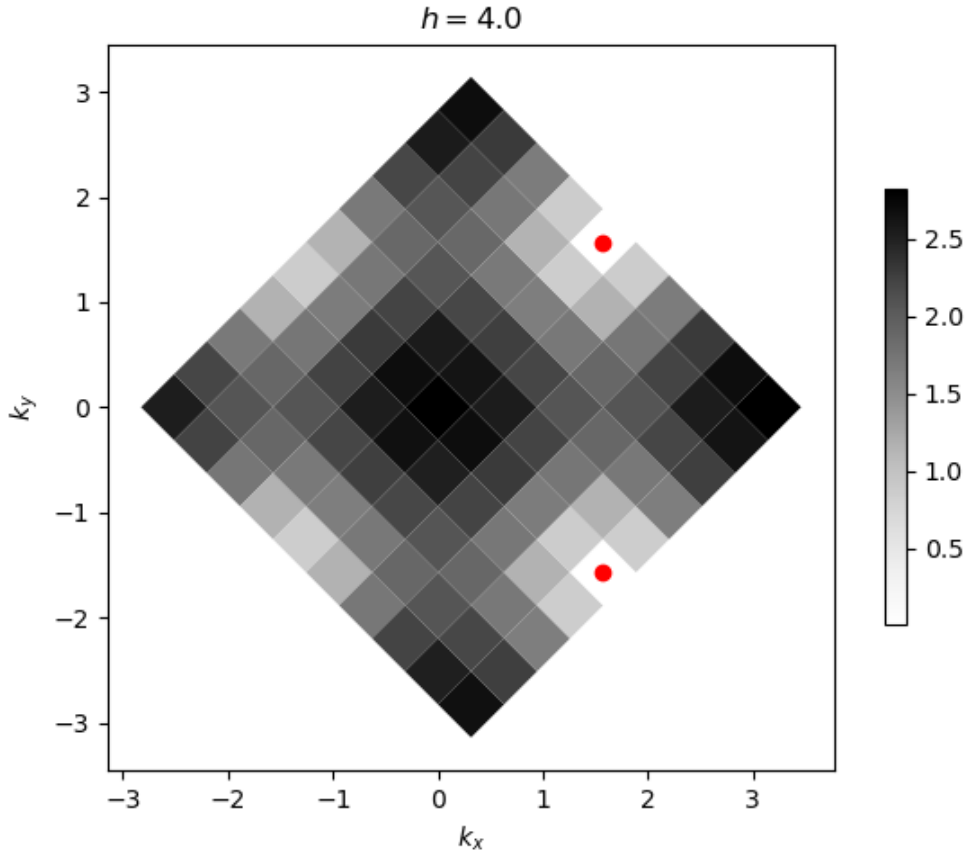
We plot the determined dispersion for system sizes $L = 10$ and mark the points $\boldsymbol{k} = (\pi/2, \pi/2)$ and $\boldsymbol{k} = (\pi/2, -\pi/2)$, which are the locations of the Dirac points in the disordered phase. One can see how the Dirac cones are displaced in the ordered phase at $h = 2.5, 3.0$, while they remain in place in the ordered phase at $h = 3.5, 4.0$. Notably, the simulation at $h = 2.5$ features $\langle \hat{s}^z \rangle < 0$ and $h = 3.0$ features $\langle \hat{s}^z \rangle > 0$ (cf. Fig. 2.1(a1)), which is random. The displaced Dirac cones are only observable because the simulation is not fully ergodic and randomly "chooses" one of the two symmetry broken phases.

```
df = res[res.l1 == 10]
latt = Lattice([10, 10], [10, -10], [1, 1], [1, -1])
for i in df.index:
    latt.plot_k(res.loc[i, 'gap_lazy'])
    plt.title(f'$h = {res.loc[i, "ham_h"]}$')
    p = np.pi/2
    plt.plot([p, p], [p, -p], 'o', color='red')
```


$h = 2.5$
k_y
k_x
3
2
1
0
−1
−2
−3
2.5
2.0
1.5
1.0
0.5


$h = 3.0$
k_y
k_x
3
2
1
0
−1
−2
−3
2.5
2.0
1.5
1.0
0.5



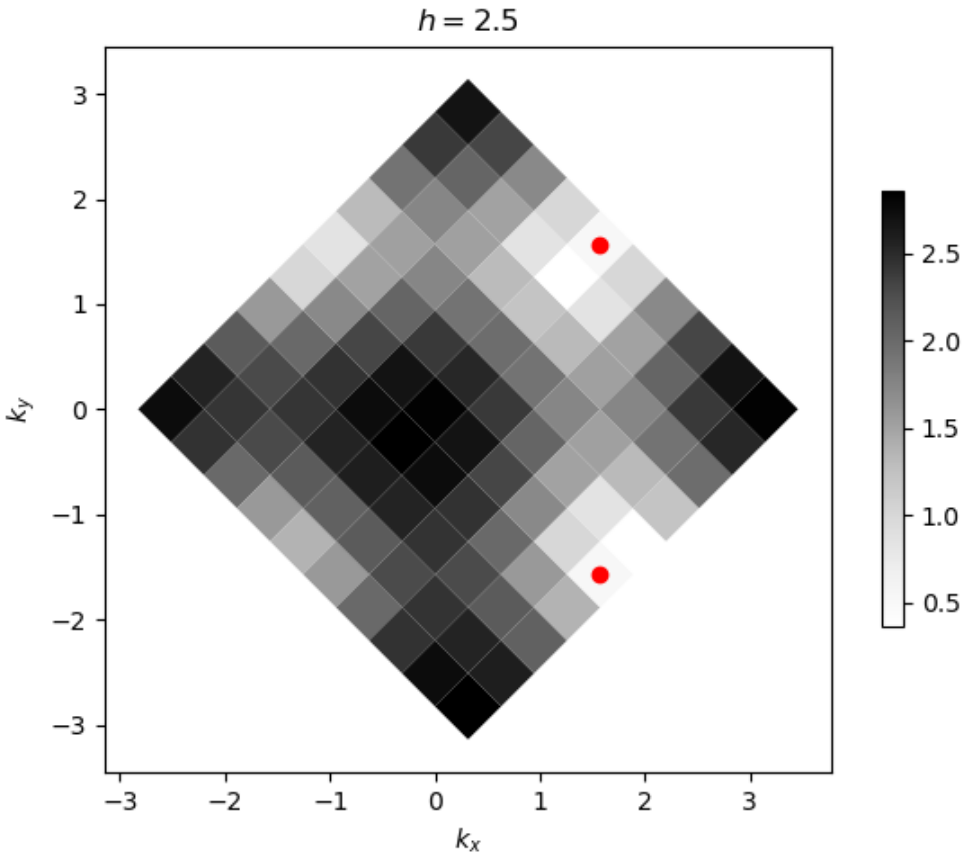


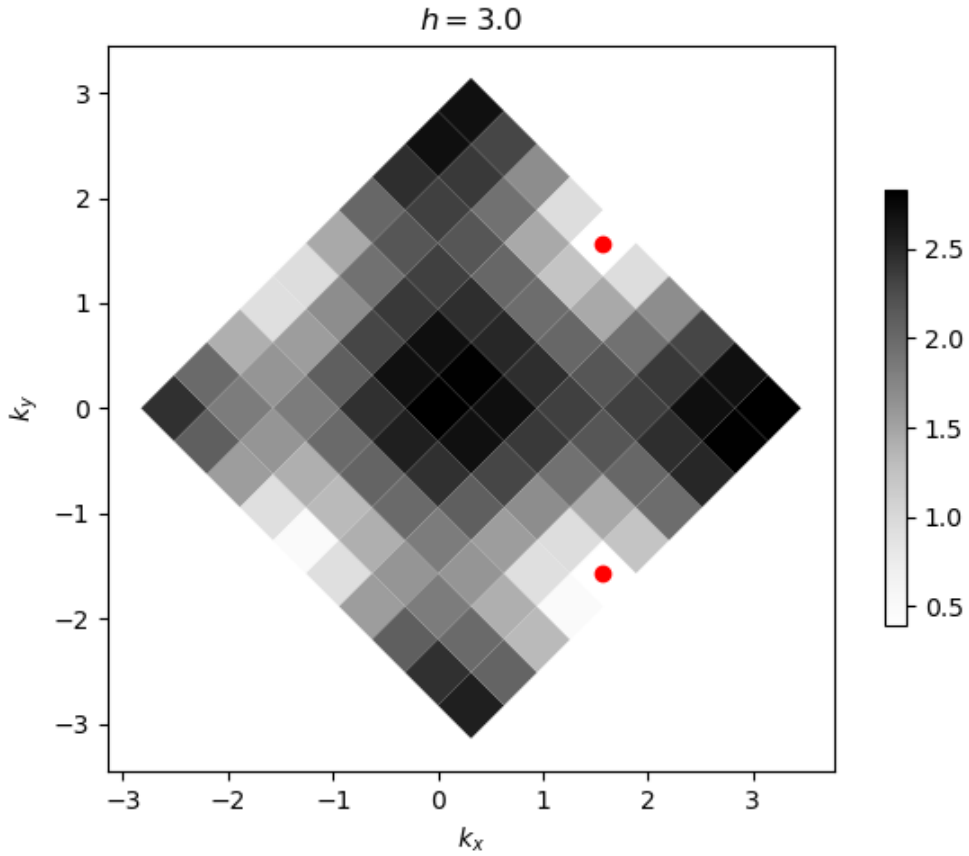

```
df = res[res.l1 == 10]
latt = Lattice([10, 10], [10, -10], [1, 1], [1, -1])
```

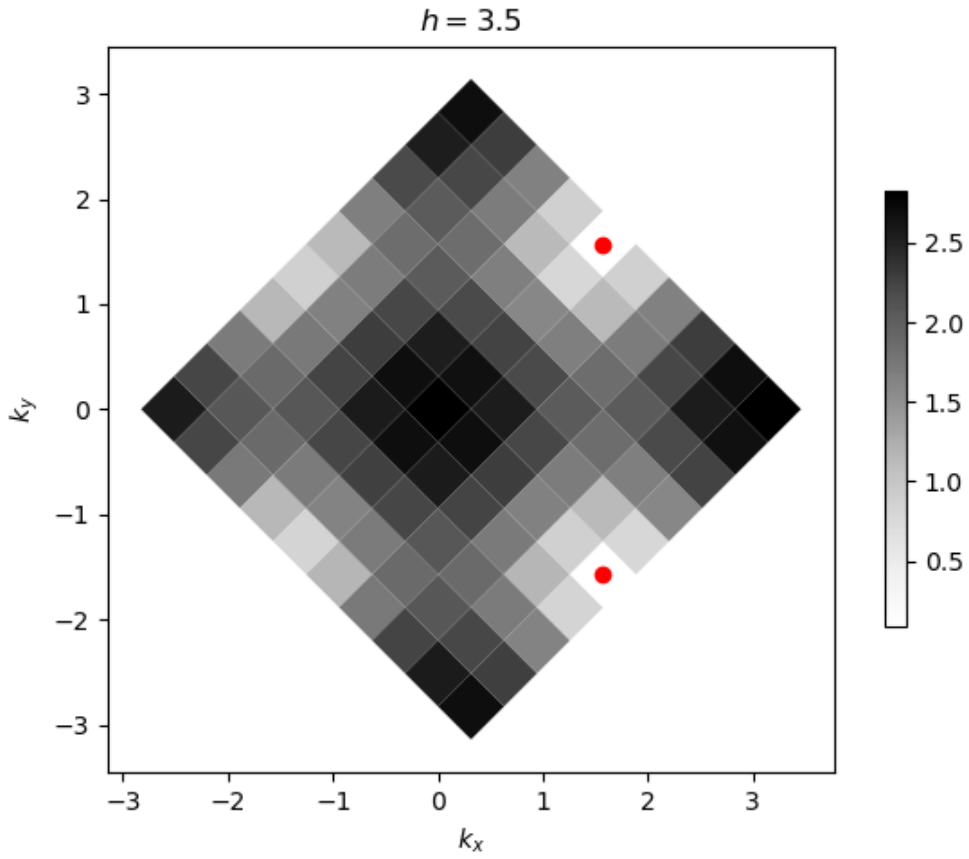
(continues on next page)

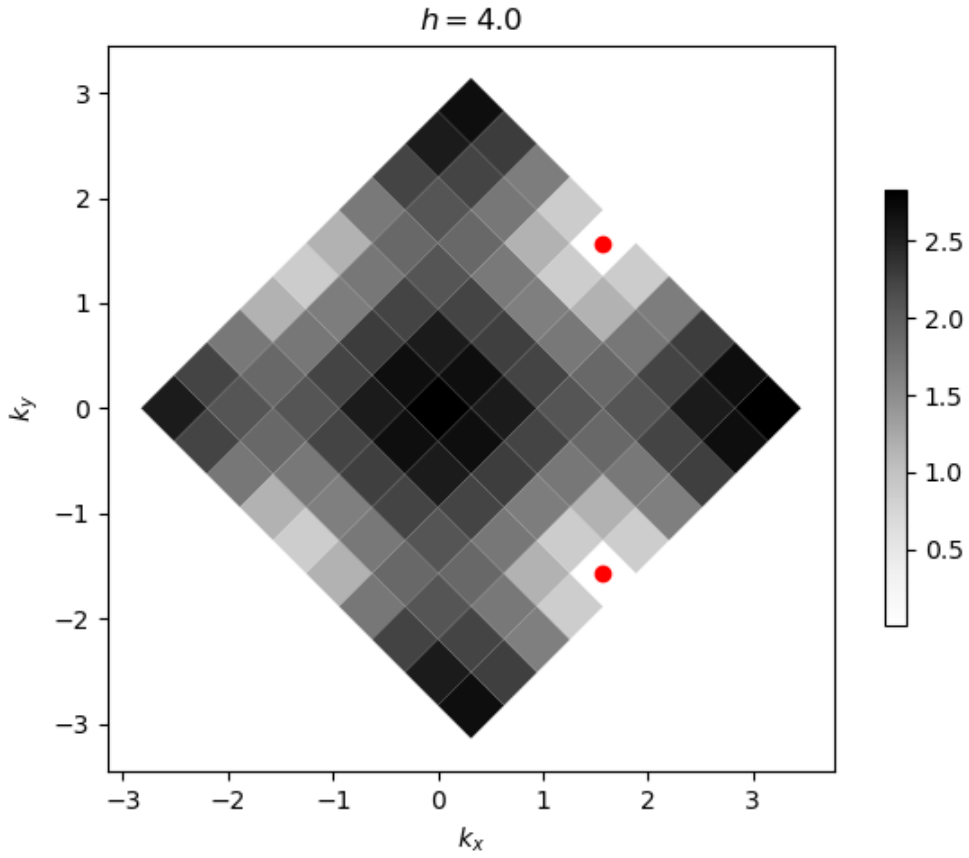
(continued from previous page)

```
for i in df.index:
    latt.plot_k(res.loc[i, 'gap'][:, 0])
    plt.title(f'$h = {res.loc[i, "ham_h"]}$')
    p = np.pi/2
    plt.plot([p, p], [p, -p], 'o', color='red')
```




$h = 3.0$
k_y
k_x
3
2
1
0
−1
−2
−3
2.5
2.0
1.5
1.0
0.5


$h = 3.5$
k_y
k_x
3
2
1
0
−1
−2
−3
2.5
2.0
1.5
1.0
0.5



A.3 Source code of data collapse functions

Listing 1.1 implements a data collapse of the form

$$\tilde{y}_i = p\left(\tilde{x}_i\right),$$

where p is a polynomial and $\tilde{x}_i$, $\tilde{y}_i$ are obtained with the scaling assumption

$$\tilde{x}_i, \sigma_{\tilde{x}_i}, \tilde{y}_i, \sigma_{\tilde{y}_i} = f\left(L_i, x_i, \sigma_{x_i}, y_i, \sigma_{y_i}, par\right)$$

By minimizing the quality of fit function

$$S = \frac{1}{N} \sum_i \frac{\left[p\left(\tilde{x}_i\right) - \tilde{y}_i\right]^2}{\sigma_{\tilde{y}_i}^2 + p'\left(\tilde{x}_i\right)^2 \sigma_{\tilde{x}_i}^2}$$

trough varying p and par. The uncertainty of the results are estimated with a bootstrap resampling scheme [118].

Listing 1.1: Content of file `collapse.py` that implements an automatic data collapse.

```
"""Provides functions for data collapse."""
# pylint: disable=invalid-name
# pylint: disable=redefined-outer-name
# pylint: disable=too-many-arguments

import numpy as np
from numpy.random import default_rng
from numpy.polynomial.polynomial import polyval, polyfit, polyder
from scipy.optimize import minimize
```

(continues on next page)

(continued from previous page)

```python
import matplotlib.pyplot as plt


def collapse(func, x_name, y_name, df, Ls, par0,
             N_boot=100, plot=True, rank=3, x_err=False,
             verbose=False, size_name='l1'):
    """Perform a data collapse.

    Rescales data according to `func` and fits polynomial of rank `rank`.
    Minimizes a quality of fit function to achieve data collapse and
    estimates error through bootstrap method.

    Parameters
    ----------
    func : function
        Function func(L, x, dx, y, dy, par) defining the rescaling to achieve
        data collapse, with:

        Parameters
        ----------
        L: int
            system size.
        x: array of floats
            x values.
        dx: array of float or None
            Error of x values
        y: array of floats
            y values.
        dy: array of float or None
            Error of y values.
        par: List of float
            parameters for data collapse, e.g. critical field, exponents.

        Returns
        -------
        x_scaled : array of float
            Scaled x values.
        dx_scaled : array of float or None
            Scaled x error.
        y_scaled : array of float
            Scaled y values.
        dy_scaled : array of float
            Scaled y error.
    x_name : str
        Name of x variable.
    y_name : str
        Name of y variable.
    df : pandas DataFrame
        Contains data.
    Ls : list of int
        System sizes to consider.
    par0 : List of floats
        Starting parameters for data collapse.
    N_boot : int, default=100
        Number of bootstrap bins to generate for error analysis.
    plot : bool, default=True
        Plot data collapse via matplotlib.
    rank : integer, default=3
        Rank of polynomial to fit.
    x_err : bool, default=False
        Consider x errors.
```

(continues on next page)

(continued from previous page)

```
    verbose : bool, default=False
        Be verbose.
    size_name : str, default="l1"
        Name of parameter corresponding to system size.

    Returns
    -------
    {
        'Ls': Ls,
        'L0': Ls[0],
        'NL': len(Ls),
        'popt': popt,
        'perr': perr,
        'S': S,
    }
    with :
        Ls : list of int
            Input argument Ls
        popt : array of float
            Best parameters found for data collapse, including parameters for
            polynomial fit function.
        perr : array of float
            Standard error of data collapse parameters.
        S : float
            Quality of data collapse. Smaller is better, should be of order 1.
    """
    data = Data_obj(df, x_name, y_name, Ls, x_err=x_err, size_name=size_name)

    if x_err:
        def quality(par, Npar, L, x, dx, y, dy):
            """The quality of fit function."""
            der = polyder(par[Npar:])
            S = 0.
            n = 0
            for L_, x_, dx_, y_, dy_ in zip(L, x, dx, y, dy):
                x_scaled, dx_scaled, y_scaled, dy_scaled = \
                    func(L_, x_, dx_, y_, dy_, par[:Npar])
                # Alternative quality of fit function.
                # S += ((polyval(x_scaled, par[Npar:]) - y_scaled)**2
                #       / (dy_scaled**2 + (polyval(x_scaled, der)*dx_scaled)**2)).
↪mean()
                # n += 1
                S += ((polyval(x_scaled, par[Npar:]) - y_scaled)**2
                      / (dy_scaled**2 + (polyval(x_scaled, der)*dx_scaled)**2)).
↪sum()
                n += len(x_scaled)
            return S / n
    else:
        def quality(par, Npar, L, x, dx, y, dy):
            """The quality of fit function."""
            S = 0.
            n = 0
            for L_, x_, dx_, y_, dy_ in zip(L, x, dx, y, dy):
                x_scaled, dx_scaled, y_scaled, dy_scaled = \
                    func(L_, x_, dx_, y_, dy_, par[:Npar])
                # Alternative quality of fit function.
                # S += ((polyval(x_scaled, par[Npar:]) - y_scaled)**2
                #       / dy_scaled**2).mean()
                # n += 1
                S += ((polyval(x_scaled, par[Npar:]) - y_scaled)**2
                      / dy_scaled**2).sum()
```

(continues on next page)

(continued from previous page)

```
                n += len(x_scaled)
            return S / n

    # Initial fit.
    Npar = len(par0)
    x_scaled, dx_scaled, y_scaled, dy_scaled = func(*data.get_data_flat(), par0)
    if x_err:
        der = polyder(p)
        p = polyfit(x_scaled, y_scaled,
                    w=1/(dy_scaled**2 + (polyval(x, der)*dx_scaled)**2),
                    deg=rank, full=False)
    else:
        p = polyfit(x_scaled, y_scaled, w=1/dy_scaled**2, deg=rank, full=False)
    L, x, dx, y, dy = data.get_data()
    res0 = minimize(quality, (*par0, *p), args=(Npar, L, x, dx, y, dy))
    if verbose:
        print(res0.fun)

    # Return results of initial fit, if no bootstrap error estimation.
    if N_boot == 0:
        if plot:
            plot_collapse(func, x_name, y_name, df, Ls,
                          res0.x[:Npar], rank=rank, x_err=x_err, size_name=size_
↪name)
        return {'Ls': Ls, 'L0': Ls[0], 'NL': len(Ls),
                'popt': res0.x, 'perr': [None]*len(res0.x), 'S': res0.fun}

    # Execute bootstrap error estimation.
    xs = np.empty((N_boot, len(res0.x)))
    for n in range(N_boot):
        L, x, dx, y, dy = data.generate_data()
        res = minimize(quality, res0.x, args=(Npar, L, x, dx, y, dy))
        xs[n] = res.x
    popt = np.mean(xs, 0)
    perr = np.std(xs, 0)
    L, x, dx, y, dy = data.get_data()
    S = quality(popt, Npar, L, x, dx, y, dy)

    if plot:
        plot_collapse(func, x_name, y_name, df, Ls, popt[:Npar],
                      rank=rank, x_err=x_err, size_name=size_name)
    return {'Ls': Ls, 'L0': Ls[0], 'NL': len(Ls),
            'popt': popt, 'perr': perr, 'S': S}


def plot_collapse(
    func, x_name, y_name, df, Ls, par, rank=3, x_err=False,
    fmts=None, size_name='l1'):
    """Plot data collapse via matplotlib. See :func:`collapse`."""
    if fmts is None:
        fmts = {4: '<', 6: '>', 8: '^', 10: 'v', 12: '*',
                14: 'x', 16: 'o', 18: '+', 20: '.'}
    data = Data_obj(df, x_name, y_name, Ls, x_err=x_err, size_name=size_name)
    x_min = np.inf
    x_max = -np.inf

    for L, x, dx, y, dy in zip(*data.get_data()):
        x_scaled, dx_scaled, y_scaled, dy_scaled = func(L, x, dx, y, dy, par)
        x_min = min(x_min, x_scaled.min())
        x_max = max(x_max, x_scaled.max())
        plt.errorbar(x_scaled, y_scaled, dy_scaled, xerr=dx_scaled,
```

(continues on next page)

(continued from previous page)

```
                      fmt=fmts[L], label=f'L={L}')

    L, x, dx, y, dy = data.get_data_flat()
    x_scaled, dx_scaled, y_scaled, dy_scaled = func(L, x, dx, y, dy, par)
    if x_err:
        der = polyder(p)
        p = polyfit(x_scaled, y_scaled,
                    w=1/(dy_scaled**2 + (polyval(x_scaled, der)*dx_scaled)**2),
                    deg=rank, full=False)
    else:
        p = polyfit(x_scaled, y_scaled, w=1/dy_scaled**2, deg=rank, full=False)

    x = np.linspace(x_min, x_max)
    plt.plot(x, polyval(x, p))
    plt.legend()


class Data_obj:
    """Encapsulate data for bootstrap."""

    def __init__(self, df, x_name, y_name, Ls, x_err=False, size_name='l1'):
        self.x_err = x_err
        self.len = 0
        self.x = []
        self.dx = []
        self.y = []
        self.dy = []
        self.Ls = Ls
        self.rng = default_rng()
        for L in Ls:
            df1 = df[df[size_name] == L]
            self.len += len(df1)
            self.x.append(np.array(df1[x_name]))
            if x_err:
                self.dx.append(np.array(df1[x_name+'_err']))
            else:
                self.dx.append(None)
            self.y.append(np.array(df1[y_name]))
            self.dy.append(np.array(df1[y_name+'_err']))

    def get_data(self):
        return self.Ls, self.x, self.dx, self.y, self.dy

    def get_data_flat(self):
        L = np.empty((self.len,))
        x = np.empty((self.len,))
        if self.x_err:
            dx = np.empty((self.len,))
        else:
            dx = None
        y = np.empty((self.len,))
        dy = np.empty((self.len,))
        i = 0
        for L_temp, x_temp, dx_temp, y_temp, dy_temp in \
                zip(self.Ls, self.x, self.dx, self.y, self.dy):
            le = len(x_temp)
            L[i:i+le] = L_temp
            x[i:i+le] = x_temp
            if self.x_err:
                dx[i:i+le] = dx_temp
            y[i:i+le] = y_temp
```

(continues on next page)

(continued from previous page)

```
            dy[i:i+le] = dy_temp
            i += le
        return L, x, dx, y, dy

    def generate_data(self):
        y_generated = []
        for y, dy, in zip(self.y, self.dy):
            y_generated.append(self.rng.normal(loc=y, scale=dy))

        if self.x_err:
            x_generated = []
            for x, dx, in zip(self.x, self.dx):
                x_generated.append(self.rng.normal(loc=x, scale=dx))
        else:
            x_generated = self.x

        return self.Ls, x_generated, self.dx, y_generated, self.dy
```

A.4 Source code for exponential fit of Green function

Listing 1.2: Content of file `fit_green_tau.py` containing the source code used to fit the time-displaced Green function with an exponential decay for determining the single particle gap. See also Section 2.7.2.1 for a derivation of the exponential decay and *Appendix A.2.2.10* demonstrates how to apply the fitting function.

```
"""Fit time-displaced Green function exponentially to determine
single particle gap."""
# pylint: disable=invalid-name

import numpy as np
from scipy.optimize import curve_fit


def fit_green_tau(taus, G, dG, plot=False):
    """Fit time-displaced Green function exponentially to determine
    single particle gap.

    Parameters
    ----------
    taus : array-like object
        Tau values.
    G : array-like object
        Green function values
    dG : array-like object
        Standard errors of Green functions
    plot : bool, default=False
        Plot G(tau).

    Returns
    -------
    gap, dgap : floats
        Single particle gap and standard error.
    """
    N_tau = len(G)

```

(continues on next page)

(continued from previous page)

```
    def func1(x, a, b):
        return a*np.exp(-b*x)

    def func2(x, a, b):
        return a*np.exp(b*(x-taus[-1]))

    if plot:
        import matplotlib.pyplot as plt
        fig, (ax, ax2) = plt.subplots(1, 2, figsize=(14, 5))
        ax.errorbar(x=taus, y=G, yerr=dG)
        ax.set_yscale('log')

    W = N_tau//2
    for i in range(N_tau//2):
        if G[i]-dG[i] < 1E-7:
            W = i
            # print(i)
            break
    i1 = W//2 - 5
    i2 = i1+10
    popt, pcov, i1_out, i2_out, chi_squared, reduced_chi_squared, accepted = \
        try_fit(func1, taus, G, dG, i1, i2)
    gap1  = popt[1]
    dgap1 = np.sqrt(pcov[1,1])
    if plot:
        # print('left')
        ax.errorbar(
            x=taus[i1_out:i2_out],
            y=G[i1_out:i2_out],
            yerr=dG[i1_out:i2_out]
            )
        title = f'{accepted} {i1_out} {i2_out} {i2_out-i1_out} {reduced_chi_
↪squared}'
        ax2.errorbar(x=[1], y=[gap1], yerr=[dgap1])

    W = N_tau//2
    for i in range(N_tau//2):
        if G[-i]-dG[-i] < 1E-7:
            W = i
            # print(i)
            break
    i2 = N_tau-W//2+5
    i1 = i2-10
    out = try_fit(func2, taus, G, dG, i1, i2)
    # print(out)
    if out is not None:
        popt, pcov, i1_out, i2_out, chi_squared, reduced_chi_squared, accepted =␣
↪out
        gap2  = popt[1]
        dgap2 = np.sqrt(pcov[1,1])
    gap  = np.mean([gap1, gap2])
    dgap = (np.max([gap1+dgap1, gap2+dgap2]) - np.min([gap1-dgap1, gap2-dgap2]))/2
    if plot:
        # print('right')
        ax.errorbar(x=taus[i1_out:i2_out],
                    y=G[i1_out:i2_out],
                    yerr=dG[i1_out:i2_out])
        ax.set_title(f'{title}\n{accepted} {i1_out} {i2_out} ' +
                     f'{i2_out-i1_out} {reduced_chi_squared}')
        ax2.errorbar(x=[2], y=[gap2], yerr=[dgap2])
        ax2.errorbar(x=[3], y=[gap], yerr=[dgap])
```

(continues on next page)

(continued from previous page)

```
        plt.show()
        plt.close()

    return gap, dgap


def try_fit(func, x, y, dy, i1, i2, popt=None):
    """Fit function func to data x[i1:i2], y[i1:i2], dy[i1:i2] and alternately
    decrease i1 and increase i2 until some standards for the quality of fit are
    no more fulfilled."""
    this_side = 1
    side1_fin = False
    side2_fin = False
    accepted = False

    while True:
        if i1 < 0 or i2 >= len(y):
            return popt, pcov, i1, i2, chi_squared, reduced_chi_squared, accepted
        try:
            popt, pcov = curve_fit(func, x[i1:i2], y[i1:i2], sigma=dy[i1:i2],
                                   absolute_sigma=True, p0=popt)
        except RuntimeError as curve_fit_failed:
            try:
                return popt, pcov, i1, i2, chi_squared, reduced_chi_squared,␣
↪accepted
            except NameError:
                raise Exception("curve_fit did not converge") from curve_fit_failed

        # ================== Estimate quality of fit ==================
        chi_squared = np.sum(((func(x[i1:i2], *popt)-y[i1:i2])/dy[i1:i2])**2)
        reduced_chi_squared = (chi_squared)/(len(x[i1:i2])-len(popt))

        n_ges  = 0
        n_in_h = 0
        n_in   = 0
        n_in_4 = 0
        n_end1 = 0
        n_end2 = 0

        for i in range(i1, i2):
            n_ges = n_ges + 1
            if abs( func(x[i], *popt) - y[i] ) < 0.7 * dy[i]:
                n_in_h = n_in_h + 1
            if abs( func(x[i], *popt) - y[i] ) < 4 * dy[i]:
                n_in_4 = n_in_4 + 1
            if abs( func(x[i], *popt) - y[i] ) < dy[i]:
                n_in = n_in + 1

        for i in range(i1,i1+3):
            if abs( func(x[i], *popt) - y[i] ) < 2*dy[i]:
                n_end1 = n_end1 + 1

        for i in range(i2-3,i2):
            if abs( func(x[i], *popt) - y[i] ) < 2*dy[i]:
                n_end2 = n_end2 + 1

        ratio_h = n_in_h / n_ges
        ratio   = n_in / n_ges
        ratio_4 = n_in_4 / n_ges

        # line = "Fitting from {} to {}: {:5.2f} {:5.2f} "
```

(continues on next page)

(continued from previous page)

```
        #         "{} {:5.2f} {:5.2f} {:5.2f} {} {}"
        # line = line.format(i1, i2, chi_squared, reduced_chi_squared,
        #                    n_ges, ratio_h, ratio, ratio_4, n_end1, n_end2)
        # print(line)

        # ====== Set minimal standard for quality of fit ======
        success = ((ratio > 0.68) and (ratio_4 > 0.99)
                   and (n_end1 > 1) and (n_end2 > 1))

        if not success:
            # print('rejected!')
            if this_side == 1:
                side1_fin = True
                i1 = i1 + 1
            elif this_side == 2:
                side2_fin = True
                i2 = i2 - 1
        else:
            # print('accepted')
            accepted = True

        if i1 == 0:
            side1_fin = True
        if i2 == len(x)-1:
            side2_fin = True

        if this_side == 1 and side2_fin is False:
            i2 = i2+1
            this_side = 2
        elif this_side == 2 and side1_fin is False:
            i1 = i1-1
            this_side = 1
        elif side2_fin is False:
            i2 = i2+2
            this_side = 2
        elif side1_fin is False:
            i1 = i1-1
            this_side = 1
        else:
            return popt, pcov, i1, i2, chi_squared, reduced_chi_squared, accepted
```

A.5 Other values for N_σ and ξ

In this appendix, we report the result of additional simulations for different values of N_σ and ξ. For the C_{2v} model, we show how at higher couplings, ξ, discontinuities occur due to level crossings, as already described in Section 2.3. For the C_{4v} model, we show that the transition stays continuous for all considered parameters.

A.5.1 The C_{2v} model

Fig. A3 shows the structure factor correlation ratio and derivative of free energy for the C_{2v} model at $N_\sigma = 4$ and $\xi \in \{0.25, 0.4, 0.5\}$. For these parameters we observe a continuous phase transition. Fig. A4 plots the same observables for $N_\sigma = 2$. For lower values of the coupling ξ the curves are also smooth, but at $\xi = 0.5$ discontinuities appear, which get more pronounced at $\xi = 0.75$. At $\xi = 0.75$, one can observe multiple discontinuities for a single system size, e.g. at $h \approx 4.2$ and $h \approx 4.4$ for $L = 20$. These discontinuities occur due to level crossings, as already described in the mean field part in Section 2.3. As shown in Fig. A4(d,f) and elaborated in the mean field section, they can be avoided by twisting the boundary conditions of the fermionic degrees of freedom.

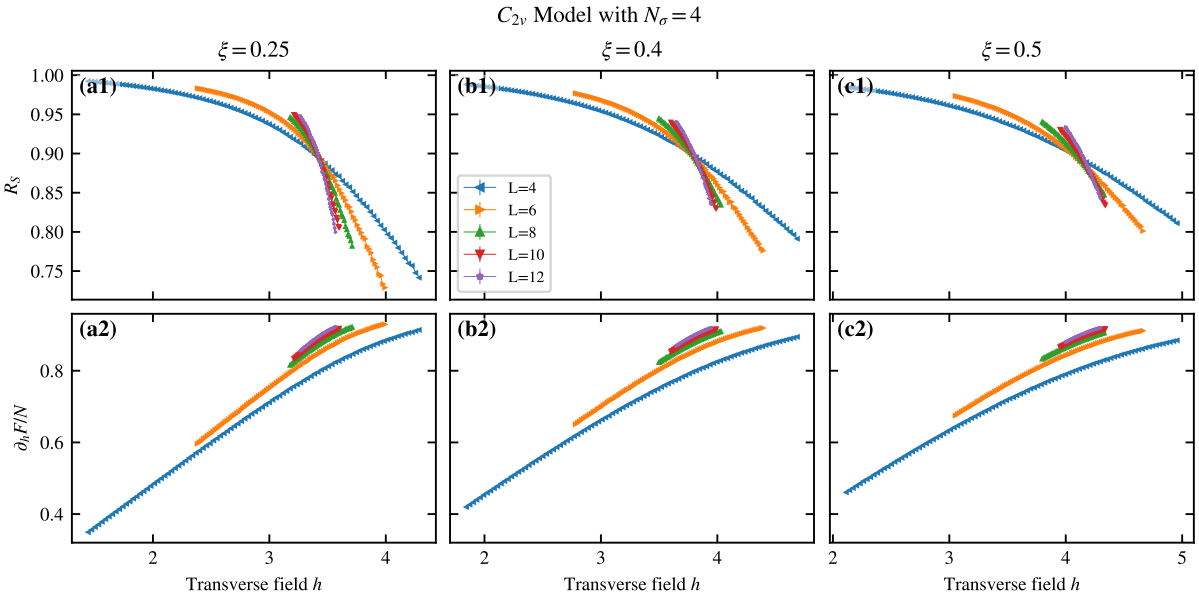


Fig. A3: Structure factor correlation ratio and derivative of free energy for the C_{2v} model at $N_\sigma = 4$ and $\xi \in \{0.25, 0.4, 0.5\}$. The data is consistent with continuous transitions.

A.5.2 The C_{4v} model

Fig. A5 and Fig. A6 have the same layout as the previous figures and show only continuous transitions for various combinations of $N_\sigma \in \{1, 2\}$, $\xi \in \{0.5, 0.75, 1, 2\}$. We also show data at $\xi = 0$, which corresponds to the transverse-field Ising model.

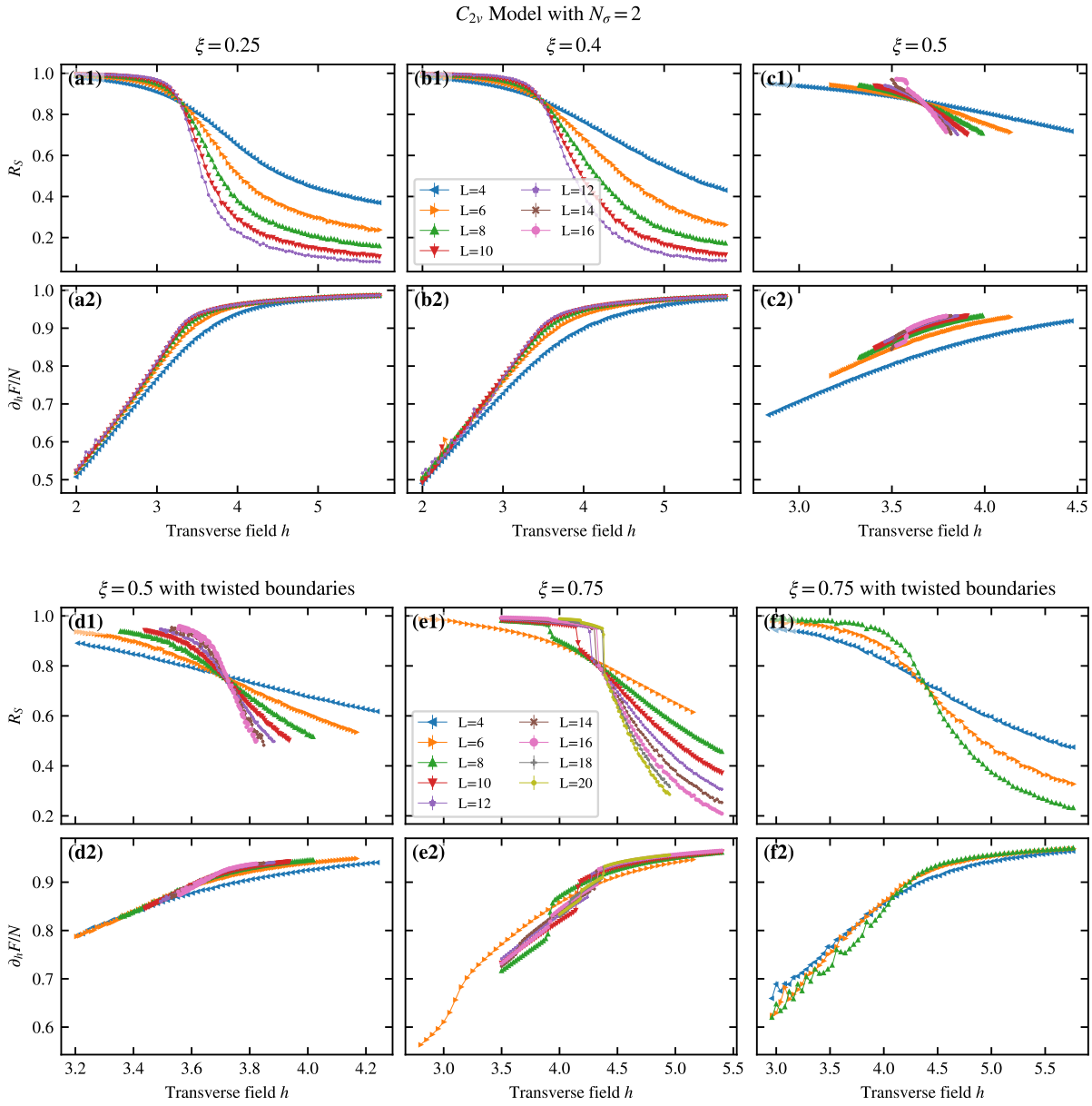


Fig. A4: Structure factor correlation ratio and derivative of free energy for the C_{2v} model at $N_\sigma = 2$ and $\xi \in \{0.25, 0.4, 0.5, 0.75\}$. At $\xi = 0.5$ and $\xi = 0.75$ discontinuities due to level crossing emerge. They can be avoided by twisting the boundary conditions of the fermionic degrees of freedom.

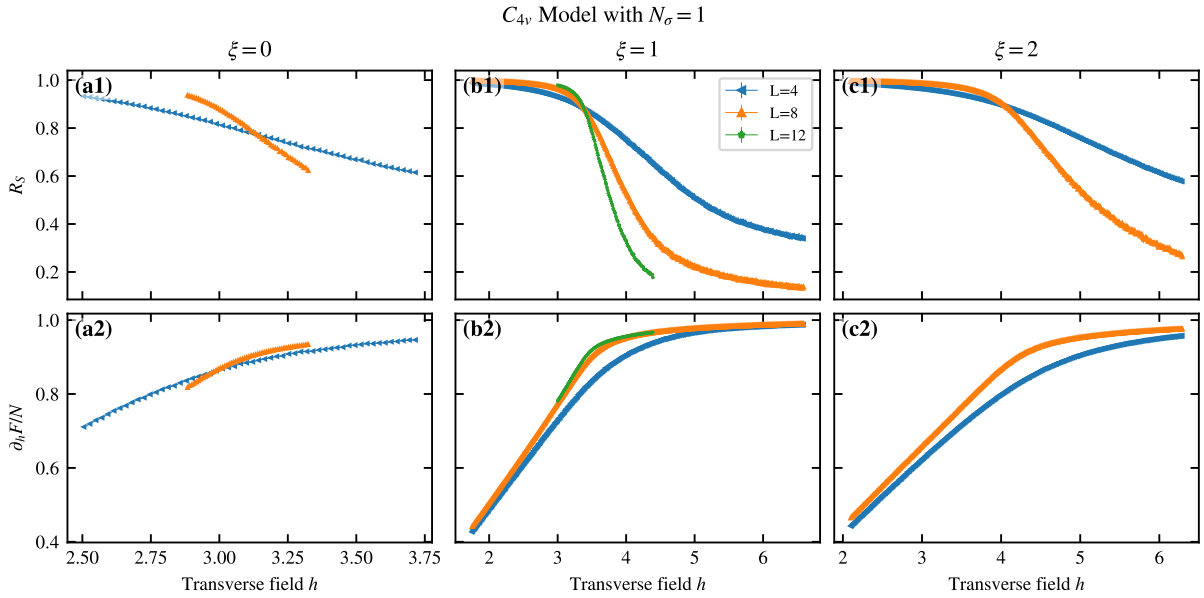


Fig. A5: Structure factor correlation ratio and derivative of free energy for the C_{4v} model at $N_\sigma = 1$ and $\xi \in \{0, 1, 2\}$. The data shows continuous transitions.

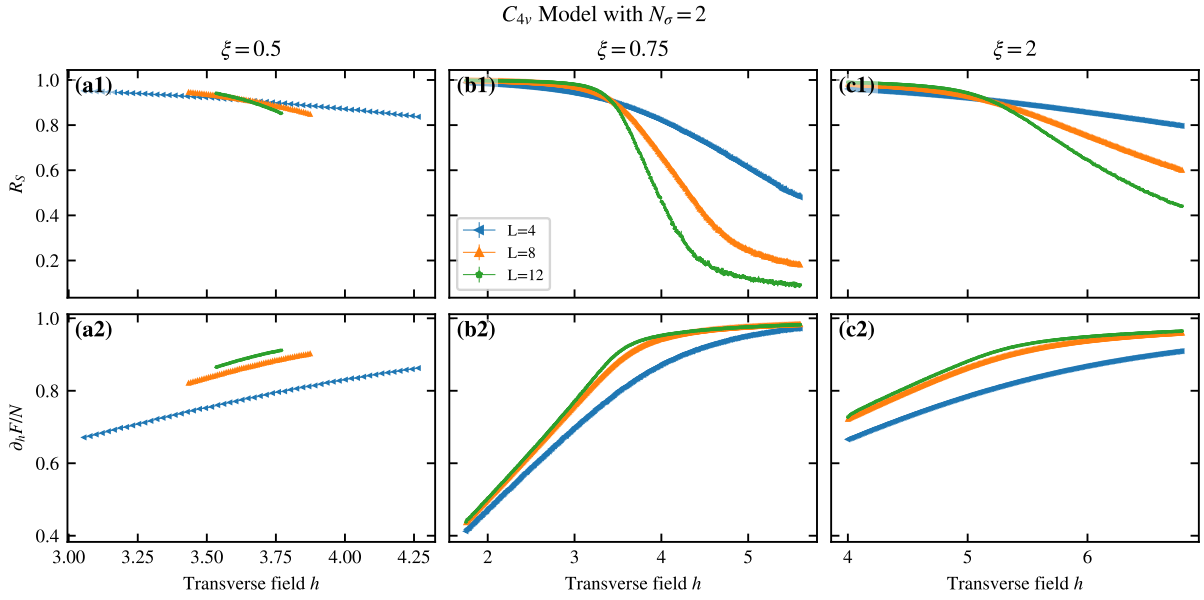


Fig. A6: Structure factor correlation ratio and derivative of free energy of the C_{4v} model at $N_\sigma = 2$ and $\xi \in \{0.5, 0.75, 2\}$. The data shows continuous transitions.

APPENDIX

B

APPENDIX TO "PHASE DIAGRAM OF THE SPIN S, SU(N) ANTIFERROMAGNET ON A SQUARE LATTICE"

The appendix for Chapter 3.

Contents

B.1 The quadratic Casimir eigenvalue in terms of the Young tableau

In this appendix, we discuss the relation between the Young tableau of an irreducible representation and the corresponding eigenvalue of the quadratic Casimir operator. For an irreducible representation, whose Young tableau has n_l rows of length $\{l_i\}$ and n_c columns of length $\{c_i\}$, the eigenvalue of the quadratic Casimir operator is [99]

$$C = \frac{1}{2}\left[r\left(N - \frac{r}{N}\right) + \sum_i^{n_l} l_i^2 - \sum_i^{n_c} c_i^2\right], \tag{2.1}$$

where $r = \sum_i l_i = \sum_i c_i$ is the total number of boxes.

In this appendix, we also derive Eq. (2.1), which is stated in the Appendix of Ref. [99] without an explicit proof. First, we notice that in Eq. (2.1) there is an implicit choice of normalization. As we show below, such a normalization is consistent with Eq. (3.10).

For an irreducible representation, the value of C can be easily computed with Weyl's formula [94][1],

$$C = \langle \Lambda | \Lambda + 2\delta \rangle, \tag{2.2}$$

where Λ is the maximum weight of the representation and δ the Weyl vector. In the Dynkin representation the metric tensor of the scalar product is, up to a normalization N, the inverse of the transpose of the Cartan matrix A [94],

$$\begin{aligned} G^{ij} &= N\left[\left(A^T\right)^{-1}\right]_{ij}, \\ \left[\left(A^T\right)^{-1}\right]_{ij} &= \min(i,j) - \frac{ij}{N}, \end{aligned} \tag{2.3}$$

[1] The Weyl's formula for the Casimir element, Eq. (2.2), should not be confused with the Weyl's formula for the dimension of an irreducible representation, Eq. (3.4).

and the Weyl vector is $\delta = (1, 1, \dots, 1)$. To fix the normalization N, we compute C for the defining representation, and match it with Eq. (3.10). For the defining representation, the maximum Dynkin weight is $\Lambda_{\alpha_i} = \delta_{i,1}$, hence

$$\begin{aligned} C &= N \sum_{i,j=1}^{N-1} \delta_{i,1} \left[\min(i,j) - \frac{ij}{N}\right] (\delta_{j,1} + 2) \\ &= N \frac{N^2 - 1}{N}. \end{aligned} \tag{2.4}$$

On the other hand, by taking the trace on both hand sides of Eq. (3.10), C is readily computed as $C = (N^2-1)/(2N)$. Therefore the normalization constant is $N = 1/2$.

Employing Eq. (2.2), we first compute $\langle\Lambda|\Lambda\rangle$. Using Eq. (3.2)

$$\langle\Lambda|\Lambda\rangle = \sum_{i,j=1}^{N-1} (l_i - l_{i+1}) G^{ij} (l_j - l_{j+1}), \tag{2.5}$$

where the metric tensor G^{ij} is given in Eq. (2.3). By developing the products and employing change of variables $i \to i-1$, $j \to j-1$, Eq. (2.5) can be written as

$$\begin{aligned} \langle\Lambda|\Lambda\rangle =& l_1 G^{11} l_1 \\ &+ l_1 \sum_{j=2}^{N-1} (G^{1,j} - G^{1,j-1}) l_j \\ &+ \sum_{i=2}^{N-1} l_i (G^{i,1} - G^{i-1,1}) l_1 \\ &+ \sum_{i,j=2}^{N-1} l_i (G^{i,j} - G^{i-1,j} - G^{i,j-1} + G^{i-1,j-1}) l_j, \end{aligned} \tag{2.6}$$

where we have used that $l_N = 0$ for Young tableaux of $\mathfrak{su}(N)$ representations. Using Eq. (2.3), we have

$$G^{1,j} - G^{1,j-1} = G^{i,1} - G^{i-1,1} = -\frac{1}{2N}, \tag{2.7}$$

where we have employed the normalization $N = 1/2$ obtained after Eq. (2.4). Further, using Eq. (2.3), the difference in the parenthesis in the last term of Eq. (2.6) is computed as

$$\begin{aligned} &G^{i,j} - G^{i-1,j} - G^{i,j-1} + G^{i-1,j-1} = \\ &\frac{1}{2}\Big[\min(i,j) - \min(i-1,j) \\ &- \min(i,j-1) - \min(i-1,j-1) - \frac{1}{N}\Big]. \end{aligned} \tag{2.8}$$

By enumerating the various cases, it is easy to see that

$$\min(i,j) - \min(i-1,j) - \min(i,j-1) - \min(i-1,j-1) = \delta_{ij}. \tag{2.9}$$

Using Eqs. (2.7), (2.8) and (2.9) in Eq. (2.6), we obtain the first term in Weyl's formula,

$$\begin{aligned} \langle\Lambda|\Lambda\rangle =& \frac{l_1^2}{2}\left(1 - \frac{1}{N}\right) - \frac{1}{N} l_1 (r - l_1) \\ &+ \frac{1}{2} \sum_{i=1}^{N-1} l_i^2 - \frac{1}{2} l_1^2 + \frac{1}{2N} (r - l_1)^2 \\ =& \frac{1}{2}\left(\sum_{i=1}^{n_l} l_i^2 - \frac{r^2}{N}\right), \end{aligned} \tag{2.10}$$

where $r = \sum_i l_i$ is the total number of boxes and the sum over l_i^2 can be restricted to the n_l nonzero row lengths.

To compute the second term $\langle\Lambda|2\delta\rangle$ in Weyl's formula, we use a different parametrization of Λ. Since in the Dynkin representation the components of Λ_{α_i} are positive integers [see Eq. (3.2)], we can parametrize Λ_{α_i} as

$$\Lambda_{\alpha_i} = \sum_{a=1}^{n_c} \delta_{i,c_a}. \tag{2.11}$$

The set $\{c_a\}$ represents the position of the rows in the corresponding Young tableau where the number of boxes decreases on the following row. Such a decrease corresponds to the end of the column, hence $\{c_a\}$ are the column lengths. Using Eq. (2.11) and Eq. (2.3) we have

$$\begin{aligned}\langle\Lambda|2\delta\rangle &= \frac{1}{2}\sum_{i,j=1}^{N-1}\sum_{a,b=1}^{n_c}\left(\delta_{i,c_a}\min(i,j)2 - \delta_{i,c_a}\frac{ij}{N}2\right)\\ &= \sum_{a=1}^{n_c}\sum_{j=1}^{N-1}\min(c_a,j) - \sum_{a=1}^{n_c}\sum_{j=1}^{N-1}\frac{c_a j}{N}\end{aligned} \tag{2.12}$$

The first sum in Eq. (2.12) can be written as

$$\begin{aligned}\sum_{a=1}^{n_c}\sum_{j=1}^{N-1}\min(c_a,j) &= \sum_{a=1}^{n_c}\left(\sum_{j=1}^{c_a} j + \sum_{j=c_a+1}^{N-1} c_a\right)\\ &= \left(N-\frac{1}{2}\right)r - \frac{1}{2}\sum_{a=1}^{n_c} c_a^2,\end{aligned} \tag{2.13}$$

where we have used $\sum_a c_a = r$. The second sum in Eq. (2.12) can be computed as

$$\sum_{a=1}^{n_c}\sum_{j=1}^{N-1}\frac{c_a j}{N} = \frac{1}{2}r(N-1) \tag{2.14}$$

Inserting Eqs. (2.13) and (2.14) in Eq. (2.12), we obtain the second term of Weyl's formula

$$\langle\Lambda|2\delta\rangle = \frac{1}{2}\left(rN - \sum_{a=1}^{n_c} c_a^2\right) \tag{2.15}$$

Finally, employing Eqs. (2.10) and (2.15) in Eq. (2.2) one obtains Eq. (2.1).

As is known from the rules of Young tableaux of $\mathfrak{su}(N)$ representations, columns of length N can be deleted since they correspond to an invariant under SU(N). This is consistent with the formula of Eq. (2.1). Indeed, by adding to a Young tableau a column of length N, we have

$$\begin{aligned} r &\to r + N,\\ l_i &\to l_i + 1,\\ \sum_{i=1}^{n_c} c_i^2 &\to N^2 + \sum_{i=1}^{n_c} c_i^2.\end{aligned} \tag{2.16}$$

Inserting the substitutions of Eq. (2.16) in Eq. (2.1), one can check that the Casimir eigenvalue is left unchanged.

Finally, it is easy to check that in the case of the defining representation, whose Young tableau is a single box, Eq. (2.1) gives the expected result, with the normalization consistent with Eq. (3.10).

B.2 Bound on the eigenvalue of the quadratic Casimir operator

The tensor product of $2S$ self-adjoint antisymmetric representations given in Eq. (3.16) decomposes into different irreducible representations. In this appendix we prove that among those representations, the maximally symmetric one of Fig. 3.2 has the maximum Casimir eigenvalue, which we compute.

Due to the rules for the composition of Young tableaux, each of the irreducible representations arising from the tensor product has a Young tableau whose total number of boxes is $r \leq (2S)(N/2) = NS$ and whose row lengths cannot exceed $2S$, $l_i \leq 2S$. Thus an upper bound for $\sum_i l_i^2$ appearing in Eq. (2.1) is

$$\sum_i^{n_l} l_i^2 \leq \sum_i^{n_l} 2Sl_i = 2Sr. \tag{2.17}$$

This bound is saturated by

$$l_i = 2S, \qquad n_l = r/(2S). \tag{2.18}$$

On the other hand, an upper bound for the second sum in Eq. (2.1) is found using the Cauchy-Schwartz inequality on the n_c-component vectors $(c_1, \dots, c_n)$ and $(1, \dots, 1)$:

$$(c_1^2 + \dots + c_n^2)(1 + \dots + 1) \geq (c_1 + \dots + c_n)^2. \tag{2.19}$$

The number of columns in the Young tableau n_c is bounded by $n_c = \max(\{l_i\}) \leq 2S$, and their sum is $\sum c_i = r$. Hence, Eq. (2.19) gives

$$\sum_i^{n_c} c_i^2 \geq \frac{r^2}{n_c} \geq \frac{r^2}{2S}. \tag{2.20}$$

This bound is saturated by

$$c_i = r/(2S), \qquad n_c = 2S. \tag{2.21}$$

Inserting Eqs. (2.17) and (2.20) in Eq. (2.1) we get

$$C \leq \frac{N+2S}{2}\left(-\frac{r^2}{2SN} + r\right) \leq \frac{NS(2S+N)}{4}, \tag{2.22}$$

where the upper bound is obtained for $r = NS$. Together with Eqs. (2.18) and (2.21) this precisely corresponds to the Young tableau of Fig. 3.2 Its Casimir eigenvalue is most easily computed using Weyl's formula (Eq. (2.2)) and Eq. (3.3), obtaining Eq. (3.22), which saturates the upper bound of Eq. (2.22). Alternatively, the Casimir eigenvalue can be obtained from Eq. (2.1), and $r = NS$, $l_1 = l_2 = \dots = l_{N/2} = 2S$ and $c_1 = c_2 = \dots = c_{2S} = N/2$.

Finally, we observe that, since the variables $\{l_i\}$ and $\{c_i\}$ in Eq. (2.1) are positive integers, as soon as we deviate from the solution maximizing C, we decrease the Casimir eigenvalue by a *finite* integer amount. In other words, there is a finite gap $O(1)$ in the eigenvalues of the quadratic Casimir operators between the subspace of the representation of Fig. 3.2 and the other irreducible representations arising from the tensor product of $2S$ self-adjoint antisymmetric representations.

B.3 Systematic errors

In this appendix we show that there is no *explicit* dependence on the magnitude of the Trotter error as a function of N. To keep the notation simple, we will show this on the basis of the $S = 1/2$ Hamiltonian where $\hat{H}_{\text{Casimir}}$ [see Eq. (3.19))] as well as the orbital index can be omitted:

$$\begin{aligned}\hat{H}_{\text{QMC}} =& \hat{H}_J + \hat{H}_U \\ =& -\frac{J}{2N}\sum_{\langle i,j\rangle}\left\{\hat{D}_{i,j}, \hat{D}_{i,,j}^\dagger\right\} + \frac{U}{N}\sum_i\left(\hat{n}_i - \frac{N}{2}\right)^2\end{aligned} \tag{2.23}$$

In this appendix, we have normalized the Hamiltonian by the factor $\frac{1}{N}$, such that total energy differences defining e.g. the spin gap $[E_0(S=1) - E_0]$ remain constant in the large-N limit. In particular, with the mean-field ansatz $\chi_{i,j} = \frac{1}{N}\langle \hat{D}_{i,j}\rangle$ corresponding to the Affleck and Marston saddle point [119], the Hamiltonian reads:

$$\hat{H}_{\text{MF}} = -\frac{J}{2}\sum_{\langle i,j\rangle}\chi_{i,j}\hat{D}_{i,,j}^\dagger + \overline{\chi_{i,j}}\hat{D}_{i,,j}. \tag{2.24}$$

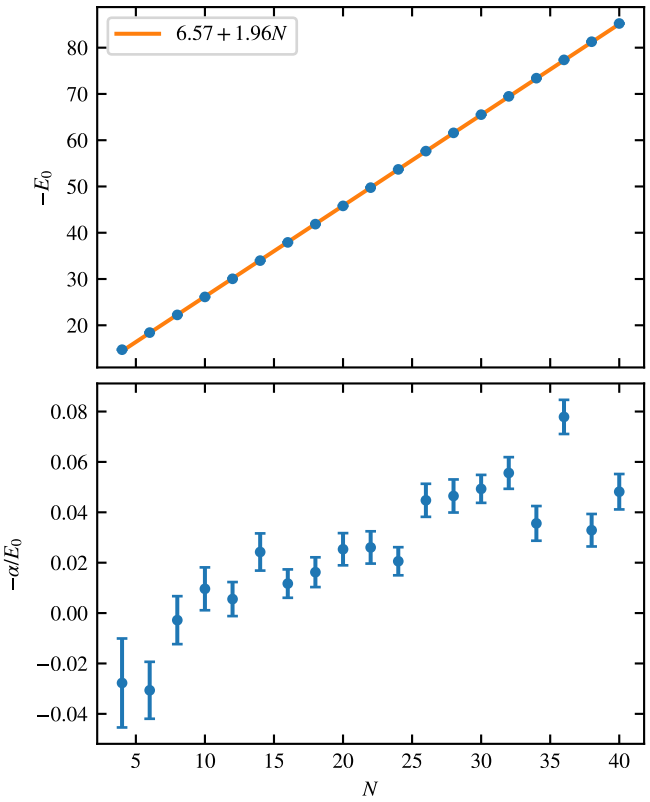


Fig. B1: Scaling of systematic Δ_τ error for $L = 4$, $\Theta = 2$, and $S = 1/2$. For each point, we simulated with a range of different values for $\Delta\tau$ and fitted the energy to $E(\Delta_\tau) = E_0 + \alpha{\Delta_\tau}^2$.

In this large-N limit, one will check explicitly that the spin gap on a finite lattice is N independent, and that the energy is extensive in the volume, V, and in N. Since gaps are N independent, at least in the large-N limit, it makes sense comparing results at different N but at constant temperature or projection parameter.

In the formulation of the AF QMC method, one introduces a checkerboard decomposition, where the interaction terms are grouped into disjoint families of commuting operators. This factorization introduces a Trotter discretization error, whose $N-$dependence we estimate as follows. To render the calculation as simple as possible, we will consider as an illustration a one-dimensional chain. In this case, the checkerboard decomposition in even and odd bonds, b, allows us to write the Hamiltonian as:

$$\hat{H} = \underbrace{\sum_{b\in A} \hat{h}_b}_{\equiv \hat{H}_A} + \underbrace{\sum_{b\in B} \hat{h}_b}_{\equiv \hat{H}_B}. \tag{2.25}$$

Both $\hat{H}_A$ and $\hat{H}_B$ are sums of commuting terms. $\hat{h}_b$ corresponds to a local Hamiltonian, such that it is extensive in N but intensive in volume.

An explicit form of $\hat{h}_b$ on a bond with legs i,j in the fermion representation would read: $\frac{1}{N}\left\{\hat{D}_{(i,j)}, \hat{D}^\dagger_{(i,j)}\right\}$. Note that to keep calculations as simple as possible, we implicitly consider a one-dimensional lattice in which $\hat{H}_A$ is a sum of commuting terms. For the two-dimensional case, the checkerboard bond decomposition necessitates four terms. We will use the symmetric Trotter decomposition,

$$e^{-\Delta\tau\hat{H}+\Delta\tau^3\hat{R}_3} = e^{-\frac{\Delta\tau}{2}\hat{H}_A} e^{-\Delta\tau\hat{H}_B} e^{-\frac{\Delta\tau}{2}\hat{H}_A} + O\left(\Delta\tau^5\right) \tag{2.26}$$

with $\hat{R}_3 = \left(\left[\hat{H}_A, \left[\hat{H}_A, \hat{H}_B\right]\right] + 2\left[\hat{H}_B, \left[\hat{H}_B, \hat{H}_A\right]\right]\right)/24$. Since $\hat{H}_A$ and $\hat{H}_B$ are sums of local operators, $\hat{R}_3$ is also a sum of local operators. Hence, $\hat{R}_3$ is extensive in the volume. By explicitly computing the commutators, one will also show, that $\hat{R}_3$ is extensive in N. Hence $\hat{R}_3$ scales as $\hat{H}$. Note that for non-local Hamiltonians considered in Ref. [120], this does not apply. We can now compute the corrections to the free energy:

$$F_{\text{QMC}} = F - \Delta\tau^2 \frac{\text{Tr} e^{-\beta\hat{H}}\hat{R}_3}{\text{Tr} e^{-\beta\hat{H}}} + O(\Delta\tau^4). \tag{2.27}$$

Hence the quantity plotted in Fig. B1 corresponds to

$$\langle \hat{R}_3 \rangle / \langle \hat{H} \rangle. \tag{2.28}$$

It is intensive in N and V, such that it has a well defined value in the large-N limit.

B.4 Bounds on the bond observable

In this appendix, we discuss a lower and upper bound for a bond observable $\sum_a \hat{S}_i^{(a)} \hat{S}_j^{(a)}$, where i and j are two distinct lattice sites, not necessarily nearest neighbor. The bond observable can be expressed as

$$\begin{aligned} \sum_a \hat{S}_i^{(a)} \hat{S}_j^{(a)} = \frac{1}{2} \sum_a \left(\hat{S}_i^{(a)} + \hat{S}_j^{(a)} \right) \left(\hat{S}_i^{(a)} + \hat{S}_j^{(a)} \right) \\ - \frac{1}{2} \sum_a \hat{S}_i^{(a)} \hat{S}_i^{(a)} - \frac{1}{2} \sum_a \hat{S}_j^{(a)} \hat{S}_j^{(a)}. \end{aligned} \tag{2.29}$$

With the choice of Eq. (3.9), the first term on the right-hand side of Eq. (2.29) is the quadratic Casimir element $\hat{C}_{2,\Gamma_i \otimes \Gamma_j}$ of the tensor product of the two $\mathfrak{su}(N)$ representations Γ_i and Γ_j, at lattice sites i and j [compare with Eq. (3.10)]. The spectrum of $\hat{C}_{2,\Gamma_i \otimes \Gamma_j}$ consists in the eigenvalues of the quadratic Casimir operator of all irreducible representations to which $\Gamma_i \otimes \Gamma_j$ reduces. An upper bound is readily found by the maximally symmetric composition of Γ_i and Γ_j, which corresponds to a Young tableau with $N/2$ rows and $4S$ columns; the proof is identical to that of *App. B.2*. Being a square of a hermitian operator, $\hat{C}_{2,\Gamma_i \otimes \Gamma_j} \geq 0$. Such lower bound is saturated by the totally antisymmetric composition of Γ_i and Γ_j, which corresponds to the trivial $S = 0$ representation. The second and third term on the right-hand side of Eq. (2.29) are the quadratic Casimir operator of the $\mathfrak{su}(N)$ representation considered here, and take the value given in Eq. (3.22). Inserting the bounds on $\hat{C}_{2,\Gamma_i \otimes \Gamma_j}$ discussed above in Eq. (2.29), we obtain

$$-C(N,S) \leq \left\langle \sum_a \hat{S}_i^{(a)} \hat{S}_j^{(a)} \right\rangle \leq C(N,2S)/2 - C(N,S) = \frac{NS^2}{2}. \tag{2.30}$$

ACKNOWLEDGMENTS

I am not big on giving speeches and the like, but still want to thank the people that helped and impacted me on the way of finally finishing this document.

Beginning with my advisor, Fakher Assaad, who has already been an excellent mentor to me even before I started my doctoral studies. He came up with very interesting projects for me, gave lots of liberty to explore my interests with e.g. pyALF and is generally a really cool guy.

I'd also like to thank the other two members of my advisory committee, Thorsten Ohl and Igor Herbut, for their easygoing cooperation and Mr. Schröder-Köhne from the GSST for his help and patience with the administrative aspects of my doctoral studies.

Furthermore, I thank my scientific collaborators for the pleasant and productive interaction: Lukas Janssen, Kain Sun, Zi Yang Meng, Igor Herbut, Matthias Vojta, Francesco Parisen Toldin and Fakher Assaad. I would like to thank Subir Sachdev for illuminating discussions on the project discussed in Chapter 3.

I also would like to thank all members and former members of AG Assaad, that made for a very pleasant working environment: Luis who thoroughly read most of this document and gave loads of constructive feedback. Jefferson, who read an early draft, was also an immense help in this thesis. Marcin, who has been sitting next to me for the last last few years, helped me a lot with literature. Flo, the go-to exert on numerical simulations and co-creator of the awesome HackyHour.
And all the others: Johannes, Stefan, Francesco, Manuel, Disha, Gabriel, Anika, Adrien, Maksim, Bimla, Zihong, Gaopei, Martin B., Martin H., Helke, Toshihiro, João, Sounak, Indra, Emilie, Zhenjiu, Fakher.

I am also grateful to the Fachschaft and the many awesome people there. There is almost always Club Mate in the fridge and a nice spot on the Couch to relax.

The ct.qmat AO, in particular in Würzburg, tha has been a great addition to my group of colleagues in the last two and a half years: Katha, Kathi, Maja-Lisa, Michi, Moni, Sabrina.

And many other great people at the physics faculty.

In acknowledgment of nice pastimes, I'll just mention the following: "T-Town", "Mr. Italy", "Wer lange sitzt muss rosten", "Edel(nice)-Rocker", "SBOA", "Segeln ohne Segeln".

My (former) flatmates made for a relaxed atmosphere to come home to: Jonas R., Sara, Fabio and René

My parents who probably did a pretty good job with raising me and my siblings were also a net positive.

For funding I would like to thank the Collaborative Research Center SFB 1170 ToCoTronics for contributing to bulk of my PhD salary and the Research Unit FOR 1807 which also contributed significantly. I also have to thank Unitary Fund[2] that supported my development of pyALF. Finally, the cluster of excellence ct.qmat and KONWIR took over my funding at the scheduled end of my doctoral training in 2022.

I am grateful to the Leibniz Supercomputing Centre[3] and the Erlangen National High Performance Computing Center (NHR@FAU[4]) for providing computing time on the Supercomputers SuperMUC-NG and Fritz, respectively, where the bulk of my simulations have been carried out.

[2] https://unitary.fund
[3] https://www.lrz.de
[4] https://hpc.fau.de/

BIBLIOGRAPHY

[1] Steve Jobs. Vintage Steve Jobs footage on Apple. Ca. 1980. Timestamp 5:27 to 7:00. URL: https://www.youtube.com/watch?v=GfxxRKBgos8&t=5m27s.

[2] The Walter J. Brown Media Archives & Peabody Awards Collection at the University of Georgia. Memory and Imagination: New Pathways to the Library of Congress. 1992. URL: https://americanarchive.org/catalog/cpb-aacip-55-76f1wdcf.

[3] Alexander Weiße and Holger Fehske. *Exact Diagonalization Techniques*, pages 529–544. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-74686-7_18.

[4] Alexander Wietek and Andreas M. Läuchli. Sublattice coding algorithm and distributed memory parallelization for large-scale exact diagonalizations of quantum many-body systems. *Phys. Rev. E*, 98:033309, Sep 2018. doi:10.1103/PhysRevE.98.033309.

[5] S. Brooks, A. Gelman, G. Jones, and X.L. Meng. *Handbook of Markov Chain Monte Carlo*. Chapman & Hall/CRC Handbooks of Modern Statistical Methods. CRC Press, 2011. ISBN 9781420079425. URL: https://books.google.de/books?id=qfRsAIKZ4rIC.

[6] N. Metropolis. The Beginning of the Monte Carlo Method. *Los Alamos Science*, 15:125–130, 1987. URL: https://library.lanl.gov/cgi-bin/getfile?15-12.pdf.

[7] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. doi:10.1109/N-SSC.2006.4785860.

[8] R. Blankenbecler, D. J. Scalapino, and R. L. Sugar. Monte Carlo calculations of coupled boson-fermion systems. *Phys. Rev. D*, 24:2278–2286, Oct 1981. doi:10.1103/PhysRevD.24.2278.

[9] S. R. White, D. J. Scalapino, R. L. Sugar, E. Y. Loh, J. E. Gubernatis, and R. T. Scalettar. Numerical study of the two-dimensional Hubbard model. *Phys. Rev. B*, 40:506–516, Jul 1989. doi:10.1103/PhysRevB.40.506.

[10] F.F. Assaad and H.G. Evertz. World-line and Determinantal Quantum Monte Carlo Methods for Spins, Phonons and Electrons. In H. Fehske, R. Schneider, and A. Weiße, editors, *Computational Many-Particle Physics*, volume 739 of Lect. Notes Phys., pages 277–356. Springer, Berlin Heidelberg, 2008. doi:10.1007/978-3-540-74686-7_10.

[11] 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. doi:10.21468/SciPostPhys.3.2.013.

[12] 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. *SciPost Phys. Codebases*, pages 1, Aug 2022. doi:10.21468/SciPostPhysCodeb.1.

[13] Jonas Schwab. Selection of my contributions to ALF. 2019-2022. https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/66, https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/75, https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/92, https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/105, https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/107, https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/113, https://git.physik.uni-wuerzburg.de/

ALF/ALF/-/merge_requests/117, https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/120, https://git.physik.uni-wuerzburg.de/ALF/ALF/-/merge_requests/141.

[14] Jonas Schwab, Lukas Janssen, Kai Sun, Zi Yang Meng, Igor F. Herbut, Matthias Vojta, and Fakher F. Assaad. Nematic Quantum Criticality in Dirac Systems. *Phys. Rev. Lett.*, 128:157203, Apr 2022. arXiv:2110.02668, doi:10.1103/PhysRevLett.128.157203.

[15] R Daou, J Chang, David LeBoeuf, Olivier Cyr-Choiniere, Francis Laliberté, Nicolas Doiron-Leyraud, BJ Ramshaw, Ruixing Liang, DA Bonn, WN Hardy, and others. Broken rotational symmetry in the pseudogap phase of a high-Tc superconductor. *Nature*, 463(7280):519–522, 2010. doi:10.1038/nature08716.

[16] RM Fernandes, AV Chubukov, and J Schmalian. What drives nematic order in iron-based superconductors? *Nature physics*, 10(2):97–104, 2014. doi:10.1038/nphys2877.

[17] Jonas Schwab, Francesco Parisen Toldin, and Fakher F. Assaad. Phase diagram of the SU(N) antiferromagnet of spin S on a square lattice. *Phys. Rev. B*, 108:115151, Sep 2023. arXiv:2304.07329, doi:10.1103/PhysRevB.108.115151.

[18] N. Read and Subir Sachdev. Some features of the phase diagram of the square lattice SU(N) antiferromagnet. *Nucl. Phys. B*, 316(3):609–640, April 1989. doi:10.1016/0550-3213(89)90061-8.

[19] Ian Affleck, Tom Kennedy, Elliott H. Lieb, and Hal Tasaki. Valence bond ground states in isotropic quantum antiferromagnets. *Commun. Math. Phys.*, 115(3):477–528, September 1988. doi:10.1007/BF01218021.

[20] N. Read and Subir Sachdev. Valence-bond and spin-Peierls ground states of low-dimensional quantum antiferromagnets. *Phys. Rev. Lett.*, 62(14):1694–1697, April 1989. doi:10.1103/PhysRevLett.62.1694.

[21] N. Read and Subir Sachdev. Spin-Peierls, valence-bond solid, and Néel ground states of low-dimensional quantum antiferromagnets. *Phys. Rev. B*, 42(7):4568–4589, September 1990. doi:10.1103/PhysRevB.42.4568.

[22] Executable Books Community. Jupyter book. February 2024. doi:10.5281/zenodo.2561065.

[23] S. S. Wilson. Bicycle Technology. *Scientific American*, pages 81–91, Mar 1973. URL: https://www.scientificamerican.com/article/bicycle-technology/.

[24] Jonas Schwab. Online version of this thesis. 2024. URL: https://purl.org/diss-jschwab.

[25] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2.

[26] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15. New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2833157.2833162.

[27] The Matplotlib Development Team. Matplotlib: Visualization with Python. August 2024. doi:10.5281/zenodo.592536.

[28] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[29] Hao Shi and Shiwei Zhang. Infinite variance in fermion quantum Monte Carlo calculations. *Phys. Rev. E*, 93:033303, Mar 2016. doi:10.1103/PhysRevE.93.033303.

[30] Maksim Ulybyshev and Fakher Assaad. Mitigating spikes in fermion Monte Carlo methods by reshuffling measurements. *Phys. Rev. E*, 106:025318, Aug 2022. doi:10.1103/PhysRevE.106.025318.

[31] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953. doi:10.1063/1.1699114.

[32] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 04 1970. doi:10.1093/biomet/57.1.97.

[33] Lars Onsager. Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition. *Phys. Rev.*, 65:117–149, Feb 1944. doi:10.1103/PhysRev.65.117.

[34] K. Binder. Finite size scaling analysis of Ising model block distribution functions. *Z. Phys. B Con. Mat.*, 43(2):119–140, 1981. doi:10.1007/BF01293604.

[35] U. Wolff. Collective Monte Carlo updating for spin systems. *Phys. Rev. Lett.*, 62:361–364, January 1989. doi:10.1103/PhysRevLett.62.361.

[36] Hale F Trotter. On the product of semi-groups of operators. *Proceedings of the American Mathematical Society*, 10(4):545–551, 1959. doi:10.1090/S0002-9939-1959-0108732-6.

[37] Subir Sachdev. *Quantum Phase Transitions*. Cambridge University Press, 2 edition, 2011. ISBN 9780511973765. doi:10.1017/CBO9780511973765.

[38] Subir Sachdev. Colloquium: Order and quantum phase transitions in the cuprate superconductors. *Rev. Mod. Phys.*, 75:913–932, Jul 2003. doi:10.1103/RevModPhys.75.913.

[39] Hilbert v. Löhneysen, Achim Rosch, Matthias Vojta, and Peter Wölfle. Fermi-liquid instabilities at magnetic quantum phase transitions. *Rev. Mod. Phys.*, 79(3):1015, July 2007. arXiv:cond-mat/0606317, doi:10.1103/RevModPhys.79.1015.

[40] Shinsei Ryu, Christopher Mudry, Chang-Yu Hou, and Claudio Chamon. Masses in graphenelike two-dimensional electronic systems: Topological defects in order parameters and their fractional exchange statistics. *Phys. Rev. B*, 80(20):205319, Nov 2009. doi:10.1103/PhysRevB.80.205319.

[41] David J. Gross and André Neveu. Dynamical symmetry breaking in asymptotically free field theories. *Phys. Rev. D*, 10:3235–3253, Nov 1974. doi:10.1103/PhysRevD.10.3235.

[42] Igor F. Herbut. Interactions and Phase Transitions on Graphene's Honeycomb Lattice. *Phys. Rev. Lett.*, 97(14):146401, October 2006. arXiv:cond-mat/0606195, doi:10.1103/PhysRevLett.97.146401.

[43] Igor F. Herbut, Vladimir Juričić, and Oskar Vafek. Relativistic Mott criticality in graphene. *Phys. Rev. B*, 80(7):075432, Aug 2009. arXiv:0904.1019, doi:10.1103/PhysRevB.80.075432.

[44] Lukas Janssen and Igor F. Herbut. Antiferromagnetic critical point on graphene's honeycomb lattice: A functional renormalization group approach. *Phys. Rev. B*, 89(20):205403, May 2014. arXiv:1402.6277, doi:10.1103/PhysRevB.89.205403.

[45] Nikolai Zerf, Luminita N. Mihaila, Peter Marquard, Igor F. Herbut, and Michael M. Scherer. Four-loop critical exponents for the Gross-Neveu-Yukawa models. *Phys. Rev. D*, 96(9):096010, Nov 2017. arXiv:1709.05057, doi:10.1103/PhysRevD.96.096010.

[46] Lukas Janssen, Igor F. Herbut, and Michael M. Scherer. Compatible orders and fermion-induced emergent symmetry in Dirac systems. *Phys. Rev. B*, 97:041117, Jan 2018. doi:10.1103/PhysRevB.97.041117(R).

[47] Shouryya Ray, Bernhard Ihrig, Daniel Kruti, John A. Gracey, Michael M. Scherer, and Lukas Janssen. Fractionalized quantum criticality in spin-orbital liquids from field theory beyond the leading order. *Phys. Rev. B*, 103(15):155160, Apr 2021. arXiv:2101.10335, doi:10.1103/PhysRevB.103.155160.

[48] Vadim Oganesyan, Steven A. Kivelson, and Eduardo Fradkin. Quantum theory of a nematic Fermi fluid. *Phys. Rev. B*, 64:195109, Oct 2001. doi:10.1103/PhysRevB.64.195109.

[49] Yoni Schattner, Samuel Lederer, Steven A. Kivelson, and Erez Berg. Ising Nematic Quantum Critical Point in a Metal: A Monte Carlo Study. *Phys. Rev. X*, 6:031028, Aug 2016. doi:10.1103/PhysRevX.6.031028.

[50] Matthias Vojta, Ying Zhang, and Subir Sachdev. Quantum Phase Transitions in d-Wave Superconductors. *Phys. Rev. Lett.*, 85:4940–4943, Dec 2000. arXiv:cond-mat/0007170, doi:10.1103/PhysRevLett.85.4940.

[51] Matthias Vojta, Ying Zhang, and Subir Sachdev. Renormalization group analysis of quantum critical points in *d*-wave superconductors. *Int. J. Mod. Phys. B*, 14(29n31):3719–3734, 2000. doi:10.1142/S0217979200004271.

[52] Yejin Huh and Subir Sachdev. Renormalization group theory of nematic ordering in *d*-wave superconductors. *Phys. Rev. B*, 78:064512, Aug 2008. doi:10.1103/PhysRevB.78.064512.

[53] Eun-Ah Kim, Michael J. Lawler, Paul Oreto, Subir Sachdev, Eduardo Fradkin, and Steven A. Kivelson. Theory of the nodal nematic quantum phase transition in superconductors. *Phys. Rev. B*, 77:184514, May 2008. doi:10.1103/PhysRevB.77.184514.

[54] Jing Wang. Velocity renormalization of nodal quasiparticles in *d*-wave superconductors. *Phys. Rev. B*, 87:054511, Feb 2013. doi:10.1103/PhysRevB.87.054511.

[55] Shouryya Ray and Lukas Janssen. Gross-Neveu-Heisenberg criticality from competing nematic and antiferromagnetic orders in bilayer graphene. *Phys. Rev. B*, 104:045101, Jul 2021. doi:10.1103/PhysRevB.104.045101.

[56] Xiao Yan Xu, Kai Sun, Yoni Schattner, Erez Berg, and Zi Yang Meng. Non-Fermi Liquid at 2+1 D Ferromagnetic Quantum Critical Point. *Phys. Rev. X*, 7:031058, Sep 2017. doi:10.1103/PhysRevX.7.031058.

[57] Yuan-Yao He, Xiao Yan Xu, Kai Sun, Fakher F. Assaad, Zi Yang Meng, and Zhong-Yi Lu. Dynamical generation of topological masses in Dirac fermions. *Phys. Rev. B*, 97:081110(R), Feb 2018. doi:10.1103/PhysRevB.97.081110.

[58] Douglas J. Scalapino, Steven R. White, and Shoucheng Zhang. Insulator, metal, or superconductor: The criteria. *Phys. Rev. B*, 47:7995–8007, Apr 1993. doi:10.1103/PhysRevB.47.7995.

[59] F. F. Assaad, W. Hanke, and D. J. Scalapino. Temperature derivative of the superfluid density and flux quantization as criteria for superconductivity in two-dimensional Hubbard models. *Phys. Rev. B*, 50:12835–12850, Nov 1994. doi:10.1103/PhysRevB.50.12835.

[60] Tobias Meng, Achim Rosch, and Markus Garst. Quantum criticality with multiple dynamics. *Phys. Rev. B*, 86:125107, Sep 2012. doi:10.1103/PhysRevB.86.125107.

[61] Lukas Janssen and Igor F. Herbut. Nematic quantum criticality in three-dimensional Fermi system with quadratic band touching. *Phys. Rev. B*, 92:045117, Jul 2015. doi:10.1103/PhysRevB.92.045117.

[62] Igor Herbut. *A Modern Approach to Critical Phenomena*. Cambridge University Press, 2007. doi:10.1017/CBO9780511755521.

[63] Yuzhi Liu, Wei Wang, Kai Sun, and Zi Yang Meng. Designer Monte Carlo simulation for the Gross-Neveu-Yukawa transition. *Phys. Rev. B*, 101:064308, Feb 2020. doi:10.1103/PhysRevB.101.064308.

[64] Bitan Roy, Vladimir Juričić, and Igor F. Herbut. Emergent Lorentz symmetry near fermionic quantum critical points in two and three dimensions. *J. High Energ. Phys.*, 2016(04):18, 2016. doi:10.1007/JHEP04(2016)018.

[65] Adam Nahum, J. T. Chalker, P. Serna, M. Ortuño, and A. M. Somoza. Deconfined Quantum Criticality, Scaling Violations, and Classical Loop Models. *Phys. Rev. X*, 5:041048, Dec 2015. doi:10.1103/PhysRevX.5.041048.

[66] Adam Nahum. Note on Wess-Zumino-Witten models and quasiuniversality in 2+1 dimensions. *Phys. Rev. B*, 102:201116, Nov 2020. doi:10.1103/PhysRevB.102.201116.

[67] Zi-Xiang Li, Yi-Fan Jiang, and Hong Yao. Majorana-Time-Reversal Symmetries: A Fundamental Principle for Sign-Problem-Free Quantum Monte Carlo Simulations. *Phys. Rev. Lett.*, 117:267002, Dec 2016. doi:10.1103/PhysRevLett.117.267002.

[68] Zi-Xiang Li, Yi-Fan Jiang, and Hong Yao. Fermion-sign-free Majarana-quantum-Monte-Carlo studies of quantum critical phenomena of Dirac fermions in two dimensions. *New Journal of Physics*, 17(8):085003, 2015. doi:10.1088/1367-2630/17/8/085003.

[69] Emilie Fulton Huffman and Shailesh Chandrasekharan. Solution to sign problems in half-filled spin-polarized electronic systems. *Phys. Rev. B*, 89:111101(R), Mar 2014. doi:10.1103/PhysRevB.89.111101.

[70] Congjun Wu and Shou-Cheng Zhang. Sufficient condition for absence of the sign problem in the fermionic quantum Monte Carlo algorithm. *Phys. Rev. B*, 71:155115, Apr 2005. doi:10.1103/PhysRevB.71.155115.

[71] N Goldenfeld. *Lectures on Phase Transitions and the Renormalization Group (1st ed.)*. CRC Press, (1992).

[72] Zi Hong Liu, Xiao Yan Xu, Yang Qi, Kai Sun, and Zi Yang Meng. Elective-momentum ultrasize quantum Monte Carlo method. *Phys. Rev. B*, 99:085114, Feb 2019. doi:10.1103/PhysRevB.99.085114.

[73] H T Diep. *Frustrated Spin Systems*. World Scientific, 3rd edition, 2020. doi:10.1142/11660.

[74] C. Castelnovo, R. Moessner, and S.L. Sondhi. Spin Ice, Fractionalization, and Topological Order. *Annual Review of Condensed Matter Physics*, 3(1):35–55, 2012. doi:10.1146/annurev-conmatphys-020911-125058.

[75] Leon Balents. Spin liquids in frustrated magnets. *Nature*, 464:199–208, 2010. doi:10.1038/nature08917.

[76] T. Senthil, Ashvin Vishwanath, Leon Balents, Subir Sachdev, and Matthew P. A. Fisher. Deconfined Quantum Critical Points. *Science*, 303(5663):1490–1494, 2004. doi:10.1126/science.1091806.

[77] H. Nishimori. *Statistical Physics of Spin Glasses and Information Processing: An Introduction*. International series of monographs on physics. Oxford University Press, 2001. ISBN 9780198509400. URL: https://global.oup.com/academic/product/statistical-physics-of-spin-glasses-and-information-processing-9780198509400.

[78] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. ISBN 9781107002173. doi:10.1017/CBO9780511976667.

[79] Anders W. Sandvik. Finite-size scaling of the ground-state parameters of the two-dimensional Heisenberg model. *Phys. Rev. B*, 56:11678–11690, Nov 1997. doi:10.1103/PhysRevB.56.11678.

[80] Matteo Calandra Buonaura and Sandro Sorella. Numerical study of the two-dimensional Heisenberg model using a Green function Monte Carlo technique with a fixed number of walkers. *Phys. Rev. B*, 57:11446–11456, May 1998. doi:10.1103/PhysRevB.57.11446.

[81] R. Coldea, S. M. Hayden, G. Aeppli, T. G. Perring, C. D. Frost, T. E. Mason, S.-W. Cheong, and Z. Fisk. Spin Waves and Electronic Interactions in La_2CuO_4. *Phys. Rev. Lett.*, 86:5377–5380, Jun 2001. doi:10.1103/PhysRevLett.86.5377.

[82] I. A. Zaliznyak, L.-P. Regnault, and D. Petitgrand. Neutron-scattering study of the dynamic spin correlations in CsNiCl3 above Néel ordering. *Phys. Rev. B*, 50:15824–15833, Dec 1994. doi:10.1103/PhysRevB.50.15824.

[83] Kliment I Kugel' and D I Khomskiĭ. The Jahn-Teller effect and magnetism: transition metal compounds. *Soviet Physics Uspekhi*, 25(4):231, 1982. doi:10.1070/PU1982v025n04ABEH004537.

[84] K. I. Kugel, D. I. Khomskii, A. O. Sboychakov, and S. V. Streltsov. Spin-orbital interaction for face-sharing octahedra: Realization of a highly symmetric SU(4) model. *Phys. Rev. B*, 91:155125, Apr 2015. doi:10.1103/PhysRevB.91.155125.

[85] S. Nakatsuji, K. Kuga, K. Kimura, R. Satake, N. Katayama, E. Nishibori, H. Sawa, R. Ishii, M. Hagiwara, F. Bridges, T. U. Ito, W. Higemoto, Y. Karaki, M. Halim, A. A. Nugroho, J. A. Rodriguez-Rivera, M. A. Green, and C. Broholm. Spin-Orbital Short-Range Order on a Honeycomb-Based Lattice. *Science*, 336(6081):559–563, 2012. doi:10.1126/science.1212154.

[86] Philippe Corboz, Miklós Lajkó, Andreas M. Läuchli, Karlo Penc, and Frédéric Mila. Spin-Orbital Quantum Liquid on the Honeycomb Lattice. *Phys. Rev. X*, 2:041013, Nov 2012. doi:10.1103/PhysRevX.2.041013.

[87] C. Wu, J. P. Hu, and S. C. Zhang. Exact SO(5) Symmetry in the Spin-3/2 Fermionic System. *Phys. Rev. Lett.*, 91:186402, 2003. doi:10.1103/PhysRevLett.91.186402.

[88] A. V. Gorshkov, M. Hermele, V. Gurarie, C. Xu, P. S. Julienne, J. Ye, P. Zoller, E. Demler, M. D. Lukin, and A. M. Rey. Two-orbital SU(N) magnetism with ultracold alkaline-earth atoms. *Nat. Phys.*, 6:289–295, 2010. doi:10.1038/nphys1535.

[89] F. D. M. Haldane. Nonlinear Field Theory of Large-Spin Heisenberg Antiferromagnets: Semiclassically Quantized Solitons of the One-Dimensional Easy-Axis Néel State. *Phys. Rev. Lett.*, 50:1153–1156, April 1983. doi:10.1103/PhysRevLett.50.1153.

[90] F. D. M. Haldane. O(3) nonlinear σ model and the topological distinction between integer- and half-integer-spin antiferromagnets in two dimensions. *Phys. Rev. Lett.*, 61(8):1029–1032, August 1988. doi:10.1103/PhysRevLett.61.1029.

[91] Anders W. Sandvik. Evidence for Deconfined Quantum Criticality in a Two-Dimensional Heisenberg Model with Four-Spin Interactions. *Phys. Rev. Lett.*, 98:227202, Jun 2007. doi:10.1103/PhysRevLett.98.227202.

[92] S Sorella, S Baroni, R Car, and M Parrinello. A Novel Technique for the Simulation of Interacting Fermion Systems. *Europhysics Letters (EPL)*, 8(7):663–668, apr 1989. doi:10.1209/0295-5075/8/7/014.

[93] F. F. Assaad. Phase diagram of the half-filled two-dimensional SU(N) Hubbard-Heisenberg model: A quantum Monte Carlo study. *Phys. Rev. B*, 71(7):075103, February 2005. arXiv:cond-mat/0406074, doi:10.1103/PhysRevB.71.075103.

[94] John Demetrius Vergados. *Group and Representation Theory*. World Scientific, February 2017. ISBN 9789813202443. doi:10.1142/10325.

[95] G Sugiyama and S.E Koonin. Auxiliary field Monte-Carlo for quantum many-body ground states. *Annals of Physics*, 168(1):1–26, 1986. doi:10.1016/0003-4916(86)90107-7.

[96] Z. Wang, F. F. Assaad, and F. Parisen Toldin. Finite-size effects in canonical and grand-canonical quantum Monte Carlo simulations for fermions. *Phys. Rev. E*, 96(4):042131, October 2017. arXiv:1706.01874, doi:10.1103/PhysRevE.96.042131.

[97] Tong Shen, Yuan Liu, Yang Yu, and Brenda M. Rubenstein. Finite temperature auxiliary field quantum Monte Carlo in the canonical ensemble. *J. Chem. Phys.*, 153(20):204108, November 2020. arXiv:2010.09813, doi:10.1063/5.0026606.

[98] Howard E. Haber. Useful relations among the generators in the defining and adjoint representations of SU(N). *SciPost Phys. Lect. Notes*, January 2021. arXiv:1912.13302, doi:10.21468/SciPostPhysLectNotes.21.

[99] K. Pilch and A. N. Schellekens. Formulas for the eigenvalues of the Laplacian on tensor harmonics on symmetric coset spaces. *J. Math. Phys.*, 25(12):3455–3459, December 1984. doi:10.1063/1.526101.

[100] Nisheeta Desai and Ribhu K. Kaul. Spin-S Designer Hamiltonians and the Square Lattice S=1 Haldane Nematic. *Phys. Rev. Lett.*, 123(10):107202, September 2019. arXiv:1904.09629, doi:10.1103/PhysRevLett.123.107202.

[101] S. Caracciolo and A. Pelissetto. Corrections to finite-size scaling in the lattice N-vector model for N=∞. *Phys. Rev. D*, 58(10):105007, November 1998. arXiv:hep-lat/9804001, doi:10.1103/PhysRevD.58.105007.

[102] F. Parisen Toldin, M. Hohenadler, F. F. Assaad, and I. F. Herbut. Fermionic quantum criticality in honeycomb and π-flux Hubbard models: Finite-size scaling of renormalization-group-invariant observables from quantum Monte Carlo. *Phys. Rev. B*, 91(16):165108, April 2015. arXiv:1411.2502, doi:10.1103/PhysRevB.91.165108.

[103] F. F. Assaad and I. F. Herbut. Pinning the Order: The Nature of Quantum Criticality in the Hubbard Model on Honeycomb Lattice. *Phys. Rev. X*, 3(3):031010, July 2013. arXiv:1304.6340, doi:10.1103/PhysRevX.3.031010.

[104] F. Parisen Toldin, F. F. Assaad, and S. Wessel. Critical behavior in the presence of an order-parameter pinning field. *Phys. Rev. B*, 95(1):014401, January 2017. arXiv:1607.04270, doi:10.1103/PhysRevB.95.014401.

[105] Arun Paramekanti and J B Marston. SU(N) quantum spin models: a variational wavefunction study. *Journal of Physics: Condensed Matter*, 19(12):125215, 2007. doi:10.1088/0953-8984/19/12/125215.

[106] Francisco H. Kim, Fakher F. Assaad, Karlo Penc, and Frédéric Mila. Dimensional crossover in the SU(4) Heisenberg model in the six-dimensional antisymmetric self-conjugate representation revealed by quantum Monte Carlo and linear flavor-wave theory. *Phys. Rev. B*, 100:085103, Aug 2019. doi:10.1103/PhysRevB.100.085103.

[107] Da Wang, Yi Li, Zi Cai, Zhichao Zhou, Yu Wang, and Congjun Wu. Competing Orders in the 2D Half-Filled SU(2N) Hubbard Model through the Pinning-Field Quantum Monte Carlo Simulations. *Phys. Rev. Lett.*, 112:156403, Apr 2014. doi:10.1103/PhysRevLett.112.156403.

[108] A. V. Onufriev and J. B. Marston. Enlarged symmetry and coherence in arrays of quantum dots. *Phys. Rev. B*, 59:12573–12578, May 1999. doi:10.1103/PhysRevB.59.12573.

[109] R. Assaraf, P. Azaria, E. Boulat, M. Caffarel, and P. Lecheminant. Dynamical Symmetry Enlargement versus Spin-Charge Decoupling in the One-Dimensional SU(4) Hubbard Model. *Phys. Rev. Lett.*, 93:016407, Jul 2004. doi:10.1103/PhysRevLett.93.016407.

[110] Nicholas Pomata and Tzu-Chieh Wei. Demonstrating the Affleck-Kennedy-Lieb-Tasaki Spectral Gap on 2D Degree-3 Lattices. *Phys. Rev. Lett.*, 124:177203, Apr 2020. doi:10.1103/PhysRevLett.124.177203.

[111] Didier Poilblanc, Norbert Schuch, and J. Ignacio Cirac. Field-induced superfluids and Bose liquids in projected entangled pair states. *Phys. Rev. B*, 88:144414, Oct 2013. doi:10.1103/PhysRevB.88.144414.

[112] Ribhu K. Kaul and Anders W. Sandvik. Lattice Model for the SU(*N*) Néel to Valence-Bond Solid Quantum Phase Transition at Large *N*. *Phys. Rev. Lett.*, 108:137201, Mar 2012. doi:10.1103/PhysRevLett.108.137201.

[113] 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.

[114] 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.

[115] 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. doi:10.1143/JPSJ.65.1604.

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

[117] Igor F. Herbut and Lukas Janssen. Topological Mott Insulator in Three-Dimensional Systems with Quadratic Band Touching. *Phys. Rev. Lett.*, 113(10):106401, Sep 2014. arXiv:1404.5721, doi:10.1103/PhysRevLett.113.106401.

[118] B. Efron and R.J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1994. ISBN 9780412042317. URL: https://books.google.de/books?id=gLlpIUxRntoC.

[119] I. Affleck and J. B. Marston. Large-*N* limit of the Heisenberg-Hubbard model: Implications for high-Tc superconductors. *Phys. Rev. B*, 37:3774–3777, March 1988. doi:10.1103/PhysRevB.37.3774.

[120] Zhenjiu Wang, Michael P. Zaletel, Roger S. K. Mong, and Fakher F. Assaad. Phases of the (2+1) Dimensional SO(5) Nonlinear Sigma Model with Topological Term. *Phys. Rev. Lett.*, 126:045701, Jan 2021. doi:10.1103/PhysRevLett.126.045701.

Affidavit

I hereby confirm that my thesis entitled *Phases and phase transitions in SU(N) spin and Dirac systems: Auxiliary field quantum Monte Carlo studies* is the result of my own work. I did not receive any help or support from commercial consultants. All sources and / or materials applied are listed and specified in the thesis.

Furthermore, I confirm that this thesis has not yet been submitted as part of another examination process neither in identical nor in similar form.

Würzburg, December 19, 2024

Place, Date Signature

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, die Dissertation *Phasen und Phasenübergänge in SU(N)-Spin- und Dirac-Systemen: Hilfsfeld Quanten Monte Carlo Studien* eigenständig, d.h. insbesondere selbständig und ohne Hilfe eines kommerziellen Promotionsberaters, angefertigt und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet zu haben.

Ich erkläre außerdem, dass die Dissertation weder in gleicher noch in ähnlicher Form bereits in einem anderen Prüfungsverfahren vorgelegen hat.

Würzburg, 19. Dezember 2024

Ort, Datum Unterschrift