The *ALF* (*A*lgorithms for *L*attice *F*ermions) project release 2.0

Documentation for the auxiliary-field quantum Monte Carlo code.

Martin Bercx, Florian Goth, Johannes S. Hofmann, Jefferson S. E. Portela,
Jonas Schwab, Fakher F. Assaad

October 16, 2020

Contents

1. Introduction **4**
1.1. Motivation . . . 4
1.2. Definition of the Hamiltonian . . . 5
1.3. Outline and What is new . . . 6

2. Auxiliary Field Quantum Monte Carlo: finite temperature **6**
2.1. Formulation of the method . . . 7
2.1.1. The partition function . . . 8
2.1.2. Observables . . . 8
2.1.3. Reweighting and the sign problem . . . 9
2.2. Updating schemes . . . 10
2.2.1. Sampling of e^{-S_0} . . . 10
2.2.2. Sequential single spin flips . . . 11
2.2.3. Global updates in space . . . 11
2.2.4. Global updates in time and space . . . 12
2.2.5. Parallel tempering . . . 12
2.2.6. Langevin dynamics . . . 13
2.3. The Trotter error and checkerboard decomposition . . . 16
2.3.1. Asymmetric Trotter decomposition . . . 16
2.3.2. Symmetric Trotter decomposition . . . 17
2.3.3. The `Symm` flag . . . 19
2.4. Stabilization - a peculiarity of the BSS algorithm . . . 19

3. Auxiliary Field Quantum Monte Carlo: projective algorithm **21**
3.1. Specification of the trial wave function . . . 21
3.2. Some technical aspects of the projective code. . . . 22
3.3. Comparison of finite and projective codes. . . . 22

4. Monte Carlo sampling **22**
4.1. The Jackknife resampling method . . . 24
4.2. An explicit example of error estimation . . . 25
4.3. Pseudo code description . . . 26

5. Data Structures and Input/Output **27**
5.1. The `Operator` type . . . 28

5.2. Handling of the fields: the `Fields` type . . . 28
5.3. The `Lattice` and `Unit_cell` types . . . 29
5.4. The observable types `Obser_Vec` and `Obser_Latt` . . . 30
5.4.1. Scalar observables . . . 32
5.4.2. Equal-time and time-displaced correlation functions . . . 32
5.5. The `WaveFunction` type . . . 33
5.6. Specification of the Hamiltonian: the `Hamiltonian` module . . . 33
5.7. File structure . . . 35
5.7.1. Input files . . . 35
5.7.2. Output files – observables . . . 37

6. Using the Code **38**
6.1. Compiling and running . . . 39
6.2. Error analysis . . . 40
6.3. Parameter optimization . . . 42

7. The plain vanilla Hubbard model on the square lattice **43**
7.1. Setting the Hamiltonian: `Ham_set` . . . 43
7.2. The lattice: `Call Ham_latt` . . . 43
7.3. The hopping: `Call Ham_hop` . . . 44
7.4. The interaction: `Call Ham_V` . . . 44
7.5. The trial wave function: `Call Ham_Trial` . . . 45
7.6. Observables . . . 45
7.6.1. Allocating space for the observables: `Call Alloc_obs(Ltau)` . . . 45
7.6.2. Measuring equal-time observables: `Obser(GR,Phase,Ntau)` . . . 46
7.6.3. Measuring time displaced observables: `ObserT(NT, GT0, G0T, G00, GTT, PHASE)` 47
7.7. Numerical precision . . . 48
7.8. Running the code and testing . . . 48

8. Predefined Structures **48**
8.1. Predefined lattices . . . 49
8.1.1. Square lattice, Fig. 5(a) . . . 49
8.1.2. Bilayer Square lattice, Fig. 5(b) . . . 50
8.1.3. N-leg Ladder lattice, Fig. 5(c) . . . 50
8.1.4. Honeycomb lattice, Fig. 5(d) . . . 51
8.1.5. Bilayer Honeycomb lattice, Fig. 5(e) . . . 51
8.1.6. π-Flux lattice (deprecated) . . . 51
8.2. Generic hopping matrices on bravais lattices . . . 52
8.2.1. Setting up the hopping matrix: the `Hopping_Matrix_type` . . . 52
8.2.2. An example: nearest neighbor hopping on the honeycomb lattice . . . 54
8.2.3. Predefined hoppings . . . 55
8.3. Predefined interaction vertices . . . 56
8.3.1. $SU(N)$ Hubbard interaction . . . 57
8.3.2. M_z-Hubbard interaction . . . 57
8.3.3. $SU(N)$ V-interaction . . . 57
8.3.4. Fermion-Ising coupling . . . 58
8.3.5. Long-Range Coulomb repulsion . . . 58
8.3.6. J_z-J_z interaction . . . 59
8.4. Predefined observables . . . 59
8.4.1. Equal-time $SU(N)$ spin-spin correlations . . . 59
8.4.2. Equal-time spin correlations . . . 60
8.4.3. Equal-time Green function . . . 60
8.4.4. Equal-time density-density correlations . . . 61
8.4.5. Time-displaced Green function . . . 61
8.4.6. Time-displaced $SU(N)$ spin-spin correlations . . . 61
8.4.7. Time-displaced spin correlations . . . 61
8.4.8. Time-displaced density-density correlations . . . 62
8.5. Predefined trial wave functions . . . 62
8.5.1. Square . . . 62

8.5.2. Honeycomb . . . 62
8.5.3. N-leg ladder . . . 62
8.5.4. Bilayer square . . . 63
8.5.5. Bilayer honeycomb . . . 63

9. Model Classes **63**
9.1. SU(N) Hubbard models `Hamiltonian_Hubbard_mod.F90` . . . 63
9.2. O(2N) t-V models `tV_mod.F90` . . . 64
9.3. SU(N) Kondo lattice models `Kondo_mod.F90` . . . 65
9.4. Models with long range Coulomb interactions `LRC_mod.F90` . . . 68
9.5. Z_2 lattice gauge theories coupled to fermion and Z_2 matter `Z2_mod.F90` . . . 69
9.5.1. Projective approach . . . 72
9.5.2. Observables . . . 72
9.5.3. A test case: Z_2 slave spin formulation of the SU(2) Hubbard model . . . 72

10. Maximum Entropy **73**
10.1. General setup . . . 73
10.2. Single-particle quantities . . . 75
10.3. Particle-hole quantities . . . 75
10.4. Particle-Particle quantities . . . 76
10.5. Zero-temperature, projective code . . . 76
10.6. Dynamics of the one-dimensional half-filled Hubbard model . . . 77

11. Conclusions and Future Directions **77**

Acknowledgments **77**

Appendix **78**
A. Performance, memory requirements and parallelization . . . 78

References **79**

License **82**

1. Introduction

1.1. Motivation

The aim of the ALF project is to provide a general formulation of the auxiliary-field QMC method with discrete fields that enables one to promptly play with different model Hamiltonians at minimal programming cost.

The auxiliary-field quantum Monte Carlo (QMC) approach is the algorithm of choice to simulate thermodynamic properties of a variety of correlated electron systems in the solid state and beyond [1, 2, 3, 4, 5, 6]. Apart from the physics of the canonical Hubbard model [7, 8], the topics one can investigate in detail include correlation effects in the bulk and on surfaces of topological insulators [9, 10], quantum phase transitions between Dirac fermions and insulators [11, 12, 13, 14, 15], deconfined quantum critical points [16, 17], topologically ordered phases [17], heavy fermion systems [18, 19], nematic [20] and magnetic [21] quantum phase transitions in metals, antiferromagnetism in metals [22], superconductivity in spin-orbit split bands [23], $SU(N)$ symmetric models [24, 25], long-ranged Coulomb interactions in graphene systems [26, 27], cold atomic gases [28], low energy nuclear physics [29], entanglement entropies and spectra [30, 31, 32, 33], among others. This ever-growing list of topics is based on algorithmic progress and on recent symmetry-related insights [34, 35, 36, 37] that lead to formulations free of the negative sign problem for a number of model systems with very rich phase diagrams.

Auxiliary-field methods can be formulated in a number of very different ways. The fields define the configuration space $\mathcal{C}$. They can stem from the Hubbard-Stratonovich (HS) [38] transformation required to decouple the many-body interacting term into a sum of non-interacting problems, or they can correspond to bosonic modes with predefined dynamics such as phonons or gauge fields. In all cases, the result is that the grand-canonical partition function takes the form

$$Z = \mathrm{Tr}\left(e^{-\beta \hat{\mathcal{H}}}\right) = \sum_{\mathcal{C}} e^{-S(\mathcal{C})}, \tag{1}$$

where β corresponds to the inverse temperature and S is the action of non-interacting fermions subject to a space-time fluctuating auxiliary field. The high-dimensional integration over the fields is carried out stochastically. In this formulation of many-body quantum systems, there is no reason for the action to be a real number. Thereby $e^{-S(\mathcal{C})}$ cannot be interpreted as a weight. To circumvent this problem one can adopt re-weighting schemes and sample $|e^{-S(\mathcal{C})}|$. This invariably leads to the so-called *negative sign problem*, with the associated exponential computational scaling in system size and inverse temperature [39]. The sign problem is formulation dependent and, as mentioned above, there has been tremendous progress at identifying an increasing number of models not affected by the negative sign problem which cover a rich domain of collective emergent phenomena. For continuous fields, the stochastic integrations can be carried out with Langevin dynamics or hybrid methods [40]. However, for many problems one can get away with discrete fields [41]. In this case, Monte Carlo importance sampling will often be put to use [42]. We note that due to the non-locality of the fermion determinant (see below), cluster updates, such as in the loop or stochastic series expansion algorithms for quantum spin systems [43, 44, 45], are hard to formulate for this class of problems. The search for efficient updating schemes that quickly wander through the configuration space defines the ongoing challenges.

Formulations differ not only in the choice of the fields, continuous or discrete, and sampling strategy, but also by the formulation of the action itself. For a given field configuration, integrating out fermionic degrees of freedom generically leads to a fermionic determinant of dimension βN where N is the volume of the system. Working with this determinant leads to the Hirsch-Fye approach [46] and the computational effort scales[1] as $\mathcal{O}\left(\beta N\right)^3$. The Hirsch-Fye algorithm is the method of choice for impurity problems, but has in general been outperformed by a class of so-called continuous-time quantum Monte Carlo approaches [47, 48, 49]. One key advantage of continuous-time methods is being action based, allowing one to better handle the retarded interactions obtained when integrating out fermion or boson baths. However, in high dimensions or at low temperatures, the cubic scaling originating from the fermionic determinant is expensive. To circumvent this, the hybrid Monte-Carlo approach [50, 5, 51] expresses the fermionic determinant in terms of a Gaussian integral thereby introducing a new variable in the Monte Carlo integration. The resulting algorithm is the method of choice for lattice gauge theories in 3+1 dimensions and has been used to provide *ab initio* estimates of light hadron masses starting from quantum chromodynamics [52].

[1] Here we implicitly assume the absence of negative sign problem

The approach we adopt lies between the above two *extremes*. We keep the fermionic determinant, but formulate the problem so as to work only with $N \times N$ matrices. This Blankenbecler, Scalapino, Sugar (BSS) algorithm scales linearly in imaginary time β, but remains cubic in the volume N. Furthermore, the algorithm can be formulated either in a projective manner [3, 4], adequate to obtain zero temperature properties in the canonical ensemble, or at finite temperatures, in the grand-canonical ensemble [2]. In this documentation we summarize the essential aspects of the auxiliary-field QMC approach, and refer the reader to Refs. [53, 6] for complete reviews.

1.2. Definition of the Hamiltonian

The first and most fundamental part of the project is to define a general Hamiltonian which can accommodate a large class of models. Our approach is to express the model as a sum of one-body terms, a sum of two-body terms each written as a perfect square of a one body term, as well as a one-body term coupled to an Ising field with dynamics to be specified by the user. Writing the interaction in terms of sums of perfect squares allows us to use generic forms of discrete approximations to the HS transformation [54, 55]. Symmetry considerations are imperative to increase the speed of the code. We therefore include a *color* index reflecting an underlying $SU(N)$ color symmetry as well as a *flavor* index reflecting the fact that after the HS transformation, the fermionic determinant is block diagonal in this index.

The class of solvable models includes Hamiltonians $\hat{\mathcal{H}}$ that have the following general form:

$$\hat{\mathcal{H}} = \hat{\mathcal{H}}_T + \hat{\mathcal{H}}_V + \hat{\mathcal{H}}_I + \hat{\mathcal{H}}_{0,I}\,, \text{ where} \tag{2}$$

$$\hat{\mathcal{H}}_T = \sum_{k=1}^{M_T}\sum_{\sigma=1}^{N_{\mathrm{col}}}\sum_{s=1}^{N_{\mathrm{fl}}}\sum_{x,y}^{N_{\mathrm{dim}}} \hat{c}^{\dagger}_{x\sigma s} T^{(ks)}_{xy} \hat{c}_{y\sigma s} \equiv \sum_{k=1}^{M_T} \hat{T}^{(k)}\,, \tag{3}$$

$$\hat{\mathcal{H}}_V = \sum_{k=1}^{M_V} U_k \left\{ \sum_{\sigma=1}^{N_{\mathrm{col}}}\sum_{s=1}^{N_{\mathrm{fl}}} \left[\left(\sum_{x,y}^{N_{\mathrm{dim}}} \hat{c}^{\dagger}_{x\sigma s} V^{(ks)}_{xy} \hat{c}_{y\sigma s} \right) + \alpha_{ks} \right] \right\}^2 \equiv \sum_{k=1}^{M_V} U_k \left(\hat{V}^{(k)} \right)^2\,, \tag{4}$$

$$\hat{\mathcal{H}}_I = \sum_{k=1}^{M_I} \hat{Z}_k \left(\sum_{\sigma=1}^{N_{\mathrm{col}}}\sum_{s=1}^{N_{\mathrm{fl}}}\sum_{x,y}^{N_{\mathrm{dim}}} \hat{c}^{\dagger}_{x\sigma s} I^{(ks)}_{xy} \hat{c}_{y\sigma s} \right) \equiv \sum_{k=1}^{M_I} \hat{Z}_k \hat{I}^{(k)}\,. \tag{5}$$

The indices and symbols used above have the following meaning:

- The number of fermion *flavors* is set by N_{fl}. After the HS transformation, the action will be block diagonal in the flavor index.
- The number of fermion *colors* is set[2] by N_{col}. The Hamiltonian is invariant under $SU(N_{\mathrm{col}})$ rotations.
- N_{dim} is the total number of spacial vertices: $N_{\mathrm{dim}} = N_{\text{unit-cell}} N_{\mathrm{orbital}}$, where $N_{\text{unit-cell}}$ is the number of unit cells of the underlying Bravais lattice and N_{orbital} is the number of (spacial) orbitals per unit cell.
- The indices x and y label lattice sites where $x, y = 1, \cdots, N_{\mathrm{dim}}$.
- Therefore, the matrices $\boldsymbol{T}^{(ks)}$, $\boldsymbol{V}^{(ks)}$ and $\boldsymbol{I}^{(ks)}$ are of dimension $N_{\mathrm{dim}} \times N_{\mathrm{dim}}$.
- The number of interaction terms is labeled by M_V and M_I. $M_T > 1$ would allow for a checkerboard decomposition.
- $\hat{c}^{\dagger}_{y\sigma s}$ is a second-quantized operator that creates an electron in a Wannier state centered around lattice site y, with color σ, and flavor index s. The operators satisfy the anti-commutation relations:

$$\left\{ \hat{c}^{\dagger}_{y\sigma s}, \hat{c}_{y'\sigma' s'} \right\} = \delta_{xx'}\delta_{ss'}\delta_{\sigma\sigma'}, \quad \text{and} \quad \left\{ \hat{c}_{y\sigma s}, \hat{c}_{y'\sigma' s'} \right\} = 0. \tag{6}$$

The bosonic part of the general Hamiltonian (2) is $\hat{\mathcal{H}}_{0,I} + \hat{\mathcal{H}}_I$ and has the following properties:

- $\hat{Z}_k$ couples to a general one-body term. It is a general operator, such as the Ising spin operator corresponding to the Pauli matrix $\hat{\sigma}_z$ or the position operator. Generically, we will work in a basis where this operator is diagonal: $\hat{Z}_k|\phi\rangle = \phi_k|\phi\rangle$. ϕ_k is a real or Ising variable.

[2] Note that in the code $N_{\mathrm{col}} \equiv$ `N_SUN`.

- The dynamics of the bosonic field is given by $\hat{\mathcal{H}}_{0,I}$. This term is not specified here; it has to be specified by the user and becomes relevant when the Monte Carlo update probability is computed in the code

Note that the matrices $\boldsymbol{T}^{(ks)}$, $\boldsymbol{V}^{(ks)}$ and $\boldsymbol{I}^{(ks)}$ explicitly depend on the flavor index s but not on the color index σ. The color index σ only appears in the second quantized operators such that the Hamiltonian is manifestly $SU(N_{\mathrm{col}})$ symmetric. We also require the matrices $\boldsymbol{T}^{(ks)}$, $\boldsymbol{V}^{(ks)}$ and $\boldsymbol{I}^{(ks)}$ to be Hermitian.

1.3. Outline and What is new

In order to use the program, a minimal understanding of the algorithm is necessary. Its code is written in Fortran, according to the 2003 standard, and natively uses MPI, for parallel runs on supercomputing systems. In this documentation we aim to present in enough detail both the algorithm and its implementation to allow the user to confidently use and modify the program.

In Sec. 2, we summarize the steps required to formulate the many-body, imaginary-time propagation in terms of a sum over HS and Ising fields of one-body, imaginary-time propagators. To simulate a model not already included in ALF, the user has to provide this one-body, imaginary-time propagator for a given configuration of HS and Ising fields. In this section we also touch on how to compute observables and on how we deal with the negative sign problem. The ALF-2.0 has a number of new updating schemes. The package comes with the possibility to implement global updates in space and time or only in space. We provide parallel-tempering and Langevin dynamics options. Another important addition in ALF 2.0 is the possibility to implement symmetric Trotter decompositions. At the end of the section we comment on the issue of stabilization for the finite temperature code.

In Sec. 3, we describe the projective version of the algorithm, constructed to produce ground state properties. This is a new feature of ALF 2.0, and one can very easily switch between projective and finite temperature codes.

One of the key challenges in Monte Carlo methods is to adequately evaluate the stochastic error. In Sec. 4 we provide an explicit example of how to correctly estimate the error.

Section 5 is devoted to the data structures that are needed to implement the model, as well as to the input and output file structure. The data structure includes an `Operator` type to optimally work with sparse Hermitian matrices, a `Lattice` type to define one- and two-dimensional Bravais lattices, a generic `Fields` type for the auxiliary fields, two `Observable` types to handle scalar observables (e.g., total energy) and equal-time or time-displaced two-point correlation functions (e.g., spin-spin correlations) and finally a `Wavefunction` type to define the trial wave function in the projective code. At the end of this section we comment on the file structure.

In Sec. 6 we provide details on running the code using the shell. As an alternative the user can download a separate project, pyALF that provides a convenient python interface as well as Jupyter notebooks.

In ALF-2.0 we have defined a set of predefined structures that allow easy reuse of lattices, observables, interactions and trial wave functions. Although convenient, this extra layer of abstraction might render ALF-2.0 harder to modify. To circumvent this we make available an implementation of a plain vanilla Hubbard model on the square lattice (see Sec. 7) that shows explicitly how to implement this basic model without making use of predefined structures. We believe that this is a good starting point to modify a Hamiltonian from scratch, as exemplified in the package's Tutorial.

Sec. 8 introduces the sets of predefined lattices, hopping matrices, interactions, observables and trial wave functions available. The goal here is to provide a library so as to facilitate implementation of new Hamiltonians.

The ALF 2.0 comes with as set of Hamiltonians, described in Sec. 9, which includes: (i) SU(N) Hubbard models, (ii) SU(N) t-V models, (iii) SU(N) Kondo lattice models, (iv) Models with long ranged coulomb interactions, and (v) Generic Z_2 lattice gauge theories coupled to Z_2 matter and fermions. These model classes are built on the predefined structures.

In Sec. 10 we describe how to use our implementation of the stochastic analytical continuation [56, 57].

Finally, in Sec. 11 we list a number of features being considered for future releases of the ALF package.

2. Auxiliary Field Quantum Monte Carlo: finite temperature

We start this section by deriving the detailed form of the partition function and outlining the computation of observables (Sec. 2.1.1 - 2.1.3). Next, we present a number of update strategies, namely local updates, global updates, and parallel tempering (Sec. 2.2). We then discuss the Trotter error, both for symmetric and asymmetric decompositions (Sec. 2.3) and, finally, we describe the measures we have implemented to make the code numerically stable (Sec. 2.4).

2.1. Formulation of the method

Our aim is to compute observables for the general Hamiltonian (2) in thermodynamic equilibrium as described by the grand-canonical ensemble. We show below how the grand-canonical partition function can be rewritten as

$$Z = \mathrm{Tr}\left(e^{-\beta\hat{\mathcal{H}}}\right) = \sum_{C} e^{-S(C)} + \mathcal{O}(\Delta\tau^2), \tag{7}$$

and define the space of configurations C. Note that the chemical potential term is already included in the definition of the one-body term $\hat{\mathcal{H}}_T$, see Eq. (3), of the general Hamiltonian. The essential ingredients of the auxiliary-field quantum Monte Carlo implementation in the ALF package are the following:

- We discretize the imaginary time propagation: $\beta = \Delta\tau L_{\mathrm{Trotter}}$. Generically this introduces a systematic Trotter error of $\mathcal{O}(\Delta\tau)^2$ [58]. We note that there has been considerable effort at getting rid of the Trotter systematic error and to formulate a genuine continuous-time BSS algorithm [59]. To date, efforts in this direction are based on a CT-AUX type formulation [60, 61] and face two issues. The first issue is that they are restricted to a class of models with Hubbard-type interactions

$$\left(\hat{n}_i - 1\right)^2 = \left(\hat{n}_i - 1\right)^4, \tag{8}$$

  such that the basic CT-AUX equation [62]

$$1 + \frac{U}{K}\left(\hat{n}_i - 1\right)^2 = \frac{1}{2}\sum_{s=\pm1} e^{\alpha s(\hat{n}_i - 1)} \quad \text{with} \quad \frac{U}{K} = \cosh(\alpha) - 1 \quad \text{and} \quad K \in \mathbb{R} \tag{9}$$

  holds. The second issue is that in the continuous-time approach it is hard to formulate a computationally efficient algorithm. Given this situation it turns out that the multi-grid method [63, 64, 65] is an efficient scheme to extrapolate to small imaginary-time steps so as to eliminate the Trotter systematic error if required.

- Having isolated the two-body term, we apply Gauß-Hermite quadrature [66] to the continuous HS transform and obtain the discrete HS transformation [54, 55]:

$$e^{\Delta\tau\lambda\hat{A}^2} = \frac{1}{4}\sum_{l=\pm1,\pm2} \gamma(l) e^{\sqrt{\Delta\tau\lambda}\,\eta(l)\hat{A}} + \mathcal{O}\left((\Delta\tau\lambda)^4\right), \tag{10}$$

  where the fields η and γ take the values:

$$\begin{aligned} \gamma(\pm1) &= 1 + \sqrt{6}/3, \qquad \eta(\pm1) = \pm\sqrt{2\left(3-\sqrt{6}\right)}, \\ \gamma(\pm2) &= 1 - \sqrt{6}/3, \qquad \eta(\pm2) = \pm\sqrt{2\left(3+\sqrt{6}\right)}. \end{aligned} \tag{11}$$

  Since the Trotter error is already of order $(\Delta\tau^2)$ per time slice, this transformation is next to exact. One can relate the expectation value of the field η_l to the operator $\hat{A}$ by noting that:

$$\begin{aligned} &\frac{1}{4}\sum_{l=\pm1,\pm2} \gamma_l e^{\sqrt{\Delta\tau\lambda}\eta_l\hat{A}}\left(\frac{\eta_l}{-2\sqrt{\Delta\tau\lambda}}\right) = e^{\Delta\tau\lambda\hat{A}^2}\hat{A} + \mathcal{O}\left((\Delta\tau\lambda)^3\right) \text{ and} \\ &\frac{1}{4}\sum_{l=\pm1,\pm2} \gamma_l e^{\sqrt{\Delta\tau\lambda}\eta_l\hat{A}}\left(\frac{\eta_l^2 - 2}{4\Delta\tau\lambda}\right) = e^{\Delta\tau\lambda\hat{A}^2}\hat{A}^2 + \mathcal{O}\left((\Delta\tau\lambda)^2\right). \end{aligned} \tag{12}$$

- $\hat{Z}_k$ can stand for a variety of operators, such as the Pauli matrix $\hat{\sigma}_z$, in which case the Ising spins take the values $s_k = \pm1$, or the position operator – such that $\hat{Z}_k|\phi\rangle = \phi_k|\phi\rangle$, with ϕ_k a real number.

- From the above it follows that the Monte Carlo configuration space C is given by the combined spaces of Ising spin configurations and of HS discrete field configurations:

$$C = \{\phi_{i,\tau}, l_{j,\tau} \text{ with } i = 1\cdots M_I,\ j = 1\cdots M_V,\ \tau = 1\cdots L_{\mathrm{Trotter}}\}. \tag{13}$$

  Here, the HS fields take the values $l_{j,\tau} = \pm2, \pm1$ and $\phi_{i,\tau}$ may, for instance, be a continuous real field or, if $\hat{Z}_k = \hat{\sigma}_z$, be restricted to ±1.

2.1.1. The partition function

With the above, the partition function of the model (2) can be written as follows.

$$
\begin{aligned}
Z &= \mathrm{Tr}\left(e^{-\beta\hat{\mathcal{H}}}\right) \\
&= \mathrm{Tr}\left[e^{-\Delta\tau\hat{\mathcal{H}}_{0,I}}\prod_{k=1}^{M_V}e^{-\Delta\tau U_k\left(\hat{V}^{(k)}\right)^2}\prod_{k=1}^{M_I}e^{-\Delta\tau\hat{\sigma}_k\hat{I}^{(k)}}\prod_{k=1}^{M_T}e^{-\Delta\tau\hat{T}^{(k)}}\right]^{L_{\mathrm{Trotter}}} + \mathcal{O}(\Delta\tau^2) \\
&= \sum_C\left(\prod_{k=1}^{M_V}\prod_{\tau=1}^{L_{\mathrm{Trotter}}}\gamma_{k,\tau}\right)e^{-S_{0,I}(\{s_{i,\tau}\})}\times \\
&\quad \mathrm{Tr}_{\mathrm{F}}\left\{\prod_{\tau=1}^{L_{\mathrm{Trotter}}}\left[\prod_{k=1}^{M_V}e^{\sqrt{-\Delta\tau U_k}\eta_{k,\tau}\hat{V}^{(k)}}\prod_{k=1}^{M_I}e^{-\Delta\tau s_{k,\tau}\hat{I}^{(k)}}\prod_{k=1}^{M_T}e^{-\Delta\tau\hat{T}^{(k)}}\right]\right\} + \mathcal{O}(\Delta\tau^2)\,.
\end{aligned} \tag{14}
$$

In the above, the trace Tr runs over the Ising spins as well as over the fermionic degrees of freedom, and Tr_{F} only over the fermionic Fock space. $S_{0,I}\left(\{s_{i,\tau}\}\right)$ is the action corresponding to the Ising Hamiltonian, and is only dependent on the Ising spins so that it can be pulled out of the fermionic trace. We have adopted the short hand notation $\eta_{k,\tau} = \eta(l_{k,\tau})$ and $\gamma_{k,\tau} = \gamma(l_{k,\tau})$. At this point, and since for a given configuration C we are dealing with a free propagation, we can integrate out the fermions to obtain a determinant:

$$
\begin{aligned}
&\mathrm{Tr}_{\mathrm{F}}\left\{\prod_{\tau=1}^{L_{\mathrm{Trotter}}}\left[\prod_{k=1}^{M_V}e^{\sqrt{-\Delta\tau U_k}\eta_{k,\tau}\hat{V}^{(k)}}\prod_{k=1}^{M_I}e^{-\Delta\tau s_{k,\tau}\hat{I}^{(k)}}\prod_{k=1}^{M_T}e^{-\Delta\tau\hat{T}^{(k)}}\right]\right\} = \\
&\prod_{s=1}^{N_{\mathrm{fl}}}\left[e^{\sum_{k=1}^{M_V}\sum_{\tau=1}^{L_{\mathrm{Trotter}}}\sqrt{-\Delta\tau U_k}\alpha_{k,s}\eta_{k,\tau}}\right]^{N_{\mathrm{col}}}\times \\
&\prod_{s=1}^{N_{\mathrm{fl}}}\left[\det\left(\mathbb{1}+\prod_{\tau=1}^{L_{\mathrm{Trotter}}}\prod_{k=1}^{M_V}e^{\sqrt{-\Delta\tau U_k}\eta_{k,\tau}\boldsymbol{V}^{(ks)}}\prod_{k=1}^{M_I}e^{-\Delta\tau s_{k,\tau}\boldsymbol{I}^{(ks)}}\prod_{k=1}^{M_T}e^{-\Delta\tau\boldsymbol{T}^{(ks)}}\right)\right]^{N_{\mathrm{col}}}\,,
\end{aligned} \tag{15}
$$

where the matrices $\boldsymbol{T}^{(ks)}$, $\boldsymbol{V}^{(ks)}$, and $\boldsymbol{I}^{(ks)}$ define the Hamiltonian [Eq. (2) - (5)]. All in all, the partition function is given by:

$$
\begin{aligned}
Z &= \sum_C e^{-S_{0,I}(\{s_{i,\tau}\})}\left(\prod_{k=1}^{M_V}\prod_{\tau=1}^{L_{\mathrm{Trotter}}}\gamma_{k,\tau}\right)e^{N_{\mathrm{col}}\sum_{s=1}^{N_{\mathrm{fl}}}\sum_{k=1}^{M_V}\sum_{\tau=1}^{L_{\mathrm{Trotter}}}\sqrt{-\Delta\tau U_k}\alpha_{k,s}\eta_{k,\tau}}\times \\
&\quad\prod_{s=1}^{N_{\mathrm{fl}}}\left[\det\left(\mathbb{1}+\prod_{\tau=1}^{L_{\mathrm{Trotter}}}\prod_{k=1}^{M_V}e^{\sqrt{-\Delta\tau U_k}\eta_{k,\tau}\boldsymbol{V}^{(ks)}}\prod_{k=1}^{M_I}e^{-\Delta\tau s_{k,\tau}\boldsymbol{I}^{(ks)}}\prod_{k=1}^{M_T}e^{-\Delta\tau\boldsymbol{T}^{(ks)}}\right)\right]^{N_{\mathrm{col}}} + \mathcal{O}(\Delta\tau^2) \\
&\equiv \sum_C e^{-S(C)} + \mathcal{O}(\Delta\tau^2)\,.
\end{aligned} \tag{16}
$$

In the above, one notices that the weight factorizes in the flavor index. The color index raises the determinant to the power N_{col}. This corresponds to an explicit $SU(N_{\mathrm{col}})$ symmetry for each configuration. This symmetry is manifest in the fact that the single particle Green functions are color independent, again for each given configuration C.

2.1.2. Observables

In the auxiliary-field QMC approach, the single-particle Green function plays a crucial role. It determines the Monte Carlo dynamics and is used to compute observables:

$$
\langle\hat{O}\rangle = \frac{\mathrm{Tr}\left[e^{-\beta\hat{H}}\hat{O}\right]}{\mathrm{Tr}\left[e^{-\beta\hat{H}}\right]} = \sum_C P(C)\langle\langle\hat{O}\rangle\rangle_{(C)},\ \text{with } P(C) = \frac{e^{-S(C)}}{\sum_C e^{-S(C)}}\,, \tag{17}
$$

and $\langle\langle\hat{O}\rangle\rangle_{(C)}$ denotes the observed value of $\hat{O}$ for a given configuration C. For a given configuration C one can use Wick's theorem to compute $O(C)$ from the knowledge of the single-particle Green function:

$$
G(x,\sigma,s,\tau|x',\sigma',s',\tau') = \langle\langle\mathcal{T}\hat{c}_{x\sigma s}(\tau)\hat{c}^{\dagger}_{x'\sigma's'}(\tau')\rangle\rangle_C \tag{18}
$$

where $\mathcal{T}$ corresponds to the imaginary-time ordering operator. The corresponding equal-time quantity reads,

$$G(x,\sigma,s,\tau|x',\sigma',s',\tau) = \langle\langle \hat{c}_{x\sigma s}(\tau)\hat{c}^{\dagger}_{x'\sigma' s'}(\tau)\rangle\rangle_C. \tag{19}$$

Since, for a given HS field, translation invariance in imaginary-time is broken, the Green function has an explicit τ and τ' dependence. On the other hand it is diagonal in the flavor index, and independent of the color index. The latter reflects the explicit $SU(N)$ color symmetry present at the level of individual HS configurations. As an example, one can show that the equal-time Green function at $\tau = 0$ reads [6]:

$$G(x,\sigma,s,0|x',\sigma,s,0) = \left(\mathbb{1} + \prod_{\tau=1}^{L_{\text{Trotter}}} \boldsymbol{B}^{(s)}_{\tau}\right)^{-1}_{x,x'} \tag{20}$$

with

$$\boldsymbol{B}^{(s)}_{\tau} = \prod_{k=1}^{M_V} e^{\sqrt{-\Delta\tau U_k}\eta_{k,\tau}\boldsymbol{V}^{(ks)}} \prod_{k=1}^{M_I} e^{-\Delta\tau s_{k,\tau}\boldsymbol{I}^{(ks)}} \prod_{k=1}^{M_T} e^{-\Delta\tau \boldsymbol{T}^{(ks)}}. \tag{21}$$

To compute equal-time, as well as time-displaced observables, one can make use of Wick's theorem. A convenient formulation of this theorem for QMC simulations reads:

$$\begin{aligned}
&\langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_1}(\tau_1) c_{\underline{x}'_1}(\tau'_1) \cdots c^{\dagger}_{\underline{x}_n}(\tau_n) c_{\underline{x}'_n}(\tau'_n)\rangle\rangle_C = \\
&\quad \det \begin{bmatrix}
\langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_1}(\tau_1) c_{\underline{x}'_1}(\tau'_1)\rangle\rangle_C & \langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_1}(\tau_1) c_{\underline{x}'_2}(\tau'_2)\rangle\rangle_C & \dots & \langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_1}(\tau_1) c_{\underline{x}'_n}(\tau'_n)\rangle\rangle_C \\
\langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_2}(\tau_2) c_{\underline{x}'_1}(\tau'_1)\rangle\rangle_C & \langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_2}(\tau_2) c_{\underline{x}'_2}(\tau'_2)\rangle\rangle_C & \dots & \langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_2}(\tau_2) c_{\underline{x}'_n}(\tau'_n)\rangle\rangle_C \\
\vdots & \vdots & \ddots & \vdots \\
\langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_n}(\tau_n) c_{\underline{x}'_1}(\tau'_1)\rangle\rangle_C & \langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_n}(\tau_n) c_{\underline{x}'_2}(\tau'_2)\rangle\rangle_C & \dots & \langle\langle \mathcal{T} c^{\dagger}_{\underline{x}_n}(\tau_n) c_{\underline{x}'_n}(\tau'_n)\rangle\rangle_C
\end{bmatrix}.
\end{aligned} \tag{22}$$

Here, we have defined the super-index $\underline{x} = \{x, \sigma, s\}$.

In Sec. 8.4 we describe the equal-time and time-displaced correlation functions that come predefined in ALF. Using the above formulation of Wick's theorem, arbitrary correlation functions can be computed. We note, however, that the program is limited to the calculation of observables that contain only two different imaginary times.

2.1.3. Reweighting and the sign problem

In general, the action $S(C)$ will be complex, thereby inhibiting a direct Monte Carlo sampling of $P(C)$. This leads to the infamous sign problem. The sign problem is formulation dependent and as noted above, much progress has been made at understanding the class of models that can be formulated without encountering this problem [34, 35, 36, 37]. When the average sign is not too small, we can nevertheless compute observables within a reweighting scheme. Here we adopt the following scheme. First note that the partition function is real such that:

$$Z = \sum_C e^{-S(C)} = \sum_C \overline{e^{-S(C)}} = \sum_C \text{Re}\left[e^{-S(C)}\right]. \tag{23}$$

Thereby[3] and with the definition

$$\text{sign }(C) = \frac{\text{Re}\left[e^{-S(C)}\right]}{\left|\text{Re}\left[e^{-S(C)}\right]\right|}, \tag{24}$$

[3]The attentive reader will have noticed that for arbitrary Trotter decompositions, the imaginary time propagator is not necessarily Hermitian. Thereby, the above equation is correct only up to corrections stemming from the controlled Trotter systematic error.

the computation of the observable [Eq. (17)] is re-expressed as follows:

$$
\begin{aligned}
\langle \hat{O} \rangle &= \frac{\sum_C e^{-S(C)} \langle\langle \hat{O} \rangle\rangle_{(C)}}{\sum_C e^{-S(C)}} \\
&= \frac{\sum_C \mathrm{Re}\left[e^{-S(C)}\right] \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]} \langle\langle \hat{O} \rangle\rangle_{(C)}}{\sum_C \mathrm{Re}\left[e^{-S(C)}\right]} \\
&= \frac{\left\{ \sum_C \left|\mathrm{Re}\left[e^{-S(C)}\right]\right| \ \mathrm{sign}\ (C) \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]} \langle\langle \hat{O} \rangle\rangle_{(C)} \right\} / \sum_C \left|\mathrm{Re}\left[e^{-S(C)}\right]\right|}{\left\{ \sum_C \left|\mathrm{Re}\left[e^{-S(C)}\right]\right| \ \mathrm{sign}\ (C) \right\} / \sum_C \left|\mathrm{Re}\left[e^{-S(C)}\right]\right|} \\
&= \frac{\left\langle \ \mathrm{sign}\ \frac{e^{-S}}{\mathrm{Re}\left[e^{-S}\right]} \langle\langle \hat{O} \rangle\rangle \right\rangle_{\overline{P}}}{\langle \mathrm{sign} \rangle_{\overline{P}}} \ .
\end{aligned} \tag{25}
$$

The average sign is

$$
\langle \mathrm{sign} \rangle_{\overline{P}} = \frac{\sum_C \left|\mathrm{Re}\left[e^{-S(C)}\right]\right| \ \mathrm{sign}\ (C)}{\sum_C \left|\mathrm{Re}\left[e^{-S(C)}\right]\right|} \ , \tag{26}
$$

and we have $\langle \mathrm{sign} \rangle_{\overline{P}} \in \mathbb{R}$ per definition. The Monte Carlo simulation samples the probability distribution

$$
\overline{P}(C) = \frac{\left|\mathrm{Re}\left[e^{-S(C)}\right]\right|}{\sum_C \left|\mathrm{Re}\left[e^{-S(C)}\right]\right|} \ . \tag{27}
$$

such that the nominator and denominator of Eq. (25) can be computed.

The negative sign problem is still an issue because the average sign is a ratio of two partition functions and one can argue that

$$
\langle \mathrm{sign} \rangle_{\overline{P}} \propto e^{-\Delta N \beta}, \tag{28}
$$

where Δ is intensive positive quantity and $N\beta$ denotes the Euclidean volume. In a Monte Carlo simulation the error scales as $1/\sqrt{T_{CPU}}$ where T_{CPU} corresponds to the computational time. Since the error on the average sign has to be much smaller than the average sign itself, one sees that:

$$
T_{CPU} \gg e^{2\Delta N \beta}. \tag{29}
$$

Two comments are in order. First, the presence of a sign problem invariably leads to an exponential increase of CPU time as a function of the Euclidean volume. And second, Δ is formulation dependent. For instance, at finite doping, the SU(2) invariant formulation of the Hubbard model presented in Sec. 9.1 has a much more severe sign problem than the formulation also presented in Sec. 9.1 where the HS field couples to the z-component of the magnetization.

2.2. Updating schemes

The program allows for different types of updating schemes, which are described below and summarized in Tab. 1. In all of them, given a configuration C, we propose a new one, C', with probability $T_0(C \to C')$ and accept it according to the Metropolis-Hastings acceptance-rejection probability,

$$
P(C \to C') = \min\left(1, \frac{T_0(C' \to C) W(C')}{T_0(C \to C') W(C)}\right), \tag{30}
$$

so as to guarantee the stationarity condition. Here, $W(C) = \left|\mathrm{Re}\left[e^{-S(C)}\right]\right|$.

2.2.1. Sampling of e^{-S_0}

This section refers to single spin-flip updates in the presence of a non-vanishing Ising action $S_0(C)$. This sampling scheme is used if the logical variable `Propose_S0` is set to `.true.`.

Consider an Ising spin at space-time i, τ in the configuration C. Flipping this spin will generate the configuration C' and we will propose the move according to

$$
T_0(C \to C') = \frac{e^{-S_0(C')}}{e^{-S_0(C')} + e^{-S_0(C)}} = 1 - \frac{1}{1 + e^{-S_0(C')}/e^{-S_0(C)}} \tag{31}
$$

Updating schemes	Type	Description
Propose_S0	Logical	If true, proposes sequential local moves according to the probability e^{-S_0}
Global_tau_moves	Logical	This option allows to carry out global moves on a single time slice. For a given time slice the user can define which part of the operator string is to be computed sequentially. This is specified by the variable N_sequential_start and Nt_sequential_end. A number of N_tau_Global user-defined global moves on the given time slice will then be carried out
Global_moves	Logical	If true, allows for global moves in space and time. A user-defined number N_Global of global moves in space and time will be carried out at the end of each sweep
Tempering	Compiling option	Requires MPI and runs the code in a parallel tempering mode, also see Sec. 2.2.5, 6.1

Table 1: Variables required to control the updating scheme. Per default the program carries out sequential, single spin-flip sweeps, and logical variables are set to false.

Note that the function S0 in the Hamitonian_example_mod.F90 module computes precisely the ratio $e^{-S_0(C')}/e^{-S_0(C)}$, therefore $T_0(C \to C')$ is obtained without any additional calculation. The proposed move is accepted with the probability:

$$P(C \to C') = \min\left(1, \frac{e^{-S_0(C)}W(C')}{e^{-S_0(C')}W(C)}\right). \tag{32}$$

Note that, as can be seen from Eq. (16), the bare action $S_0(C)$ determining the dynamics of the Ising spin in the absence of coupling to the fermions does not enter the Metropolis acceptance-rejection step.

2.2.2. Sequential single spin flips

The program adopts per default a sequential, single spin-flip strategy. It will visit sequentially each HS field in the space-time operator list and propose a spin flip. Consider the Ising spin $s_{i,\tau}$. We will flip it with probability 1, such that for this local move the proposal matrix is symmetric. If we are considering the HS field $l_{i,\tau}$ we will propose with probability 1/3 one of the other three possible fields. Again, for this local move, the proposal matrix is symmetric. Hence in both cases we will accept or reject the move according to

$$P(C \to C') = \min\left(1, \frac{W(C')}{W(C)}\right). \tag{33}$$

This default updating scheme can be overruled by, e.g., setting Global_tau_moves to .true. and not setting Nt_sequential_start and Nt_sequential_end (see Sec. 5.7.1). It is also worth noting that this type of sequential spin-flip updating does not satisfy detailed balance, but rather the more fundamental stationarity condition [42].

2.2.3. Global updates in space

This option allows one to carry out user-defined global moves on a single time slice. This option is enabled by setting the logical variable Global_tau_moves to .true.. To set the stage we recall that the propagation over a time step $\Delta\tau$ (see Eq. 21) can be written as:

$$e^{-V_{M_I+M_V}(s_{M_I+M_V,\tau})} \cdots e^{-V_1(s_{1,\tau})} \prod_{k=1}^{M_T} e^{-\Delta\tau \boldsymbol{T}^{(k)}} \tag{34}$$

where $e^{-V_n(s_n)}$ denotes one element of the operator list containing the HS fields. One can provide an interval of indices, [Nt_sequential_start, Nt_sequential_end], in which the operators will be updated sequentially. Setting Nt_sequential_start $= 1$ and Nt_sequential_end $= M_I + M_V$ reproduces the sequential single spin flip strategy of the above section.

The variable N_tau_Global sets the number of global moves carried out on each time slice ntau. Each global move is generated in the routine Global_move_tau, which is provided by the user in the Hamiltonian file. In order to define this move, one specifies the following variables:

- `Flip_length`: An integer stipulating the number of spins to be flipped.
- `Flip_list(1:Flip_length)`: Integer array containing the indices of the operators to be flipped.
- `Flip_value(1:Flip_length)`: `Flip_value(n)` is an integer containing the new value of the HS field for the operator `Flip_list(n)`.
- `T0_Proposal_ratio`: Real number containing the quotient

$$\frac{T_0(C' \to C)}{T_0(C \to C')} , \tag{35}$$

  where C' denotes the new configuration obtained by flipping the spins specified in the `Flip_list` array. Since we allow for a stochastic generation of the global move, it may very well be that no change is proposed. In this case, `T0_Proposal_ratio` takes the value 0 upon exit of the routine `Global_move_tau` and no update is carried out.
- `S0_ratio`: Real number containing the ratio $e^{-S_0(C')}/e^{-S_0(C)}$.

2.2.4. Global updates in time and space

The code allows for global updates as well. The user must then provide two additional functions in the module `Hamiltonian_Examples_mod.F90`: `Global_move` and `Delta_S0_global(Nsigma_old)`.

The subroutine `Global_move(T0_Proposal_ratio,nsigma_old,size_clust)` proposes a global move. Its single input is the variable `nsigma_old` of type `Field` (see Section 5.2) that contains the full configuration C stored in `nsigma_old%f(M_V + M_I, Ltrot)`. On output, the new configuration C', determined by the user, is stored in the two-dimensional array `nsigma`, which is a global variable declared in the module `Hamiltonian`. Like for the global move in space (Sec. 2.2.3), `T0_Proposal_ratio` contains the proposal ratio $T_0(C' \to C)/T_0(C \to C')$. Since we allow for a stochastic generation of the global move, it may very well be that no change is proposed. In this case, `T0_Proposal_ratio` takes the value 0 upon exit, and `nsigma`=`nsigma_old`. The real-valued `size_clust` gives the size of the proposed move (e.g., $\frac{\text{Number of flipped spins}}{\text{Total number of spins}}$). This is used to calculate the average sizes of proposed and accepted moves which will be printed in the `info` file. The variable `size_clust` is not necessary for the simulation, but may help the user to estimate the effectiveness of the global update.

In order to compute the acceptance-rejection ratio, the user must also provide a function `Delta_S0_global(nsigma_old)` that computes the ratio $e^{-S_0(C')}/e^{-S_0(C)}$. Again, the configuration C' is given by the field `nsigma`.

The variable `N_Global` determines the number of global updates performed per sweep. Note that global updates are expensive, since they require a complete recalculation of the weight.

2.2.5. Parallel tempering

Exchange Monte Carlo [67], or parallel tempering [68], is a possible route to overcome sampling issues in parts of the parameter space. Let h be a parameter which one can vary without altering the configuration space $\{C\}$ and let us assume that for some values of h one encounters sampling problems. For example, in the realm of spin glasses, h could correspond to the inverse temperature. Here at high temperatures the phase space is easily sampled, but at low temperatures simulations get stuck in local minima. For quantum systems, h could trigger a quantum phase transition where sampling issues are encountered, for example, in the ordered phase and not in the disordered one. As its name suggests, parallel tempering carries out in parallel simulations at consecutive values of h: $h_1, h_2, \cdots h_n$, with $h_1 < h_2 < \cdots < h_n$. One will sample the extended ensemble:

$$P([h_1, C_1], [h_2, C_2], \cdots, [h_n, C_n]) = \frac{W(h_1, C_1)W(h_2, C_2)\cdots W(h_n, C_n)}{\sum_{C_1, C_2, \cdots, C_n} W(h_1, C_1)W(h_2, C_2)\cdots W(h_n, C_n)} \tag{36}$$

where $W(h, C)$ corresponds to the weight for a given value of h and configuration C. Clearly, one can sample $P([h_1, C_1], [h_2, C_2], \cdots, [h_n, C_n])$ by carrying out n independent runs. However, parallel tempering includes the following exchange step:

$$[h_1, C_1], \cdots, [h_i, C_i], [h_{i+1}, C_{i+1}] \cdots, [h_n, C_n] \to [h_1, C_1], \cdots, [h_i, C_{i+1}], [h_{i+1}, C_i] \cdots, [h_n, C_n] \tag{37}$$

which, for a symmetric proposal matrix, will be accepted with probability:

$$\min\left(1, \frac{W(h_i, C_{i+1})W(h_{i+1}, C_i)}{W(h_i, C_i)W(h_{i+1}, C_{i+1})}\right). \tag{38}$$

Thereby, a configuration can meander in parameter space h and explore regions where ergodicity is not an issue. In the context of spin-glasses, a low temperature configuration, stuck in a local minima, can heat up, overcome the potential barrier and then cool down again.

A judicious choice of the values h_i is important to obtain a good acceptance rate for the exchange step. With $W(h, C) = e^{-S(h,C)}$, the distribution of the action S reads:

$$\mathcal{P}(h, S) = \sum_C P(h, C)\delta(S(h, C) - S). \tag{39}$$

A given exchange step can only be accepted if the distributions $\mathcal{P}(h, S)$ and $\mathcal{P}(h + \Delta h, S)$ overlap. For $\langle S\rangle_h < \langle S\rangle_{h+\Delta h}$ one can formulate this requirement as:

$$\langle S\rangle_h + \langle \Delta S\rangle_h \simeq \langle S\rangle_{h+\Delta h} - \langle \Delta S\rangle_{h+\Delta h}, \text{ with } \langle \Delta S\rangle_h = \sqrt{\langle (S - \langle S\rangle_h)^2\rangle_h}. \tag{40}$$

Assuming $\langle \Delta S\rangle_{h+\Delta h} \simeq \langle \Delta S\rangle_h$ and expanding in Δh one obtains:

$$\Delta h \simeq \frac{2\langle \Delta S\rangle_h}{\partial \langle S\rangle_h / \partial h}. \tag{41}$$

The above equation becomes transparent for classical systems with $S(h, C) = hH(C)$. In this case, the above equation reads:

$$\Delta h \simeq 2h\frac{\sqrt{C}}{C + h\langle H\rangle_h}, \text{ with } C = h^2\langle (H - \langle H\rangle_h)^2\rangle_h. \tag{42}$$

Several comments are in order:

i) Let us identify h with the inverse temperature such that C corresponds to the specific heat. This quantity is extensive, as well as the energy, such that $\Delta h \simeq 1/\sqrt{N}$ where N is the system size.

ii) In the proximity of a phase transition the specific heat can diverge, and h must be chosen with particular care.

iii) Since the action is formulation dependent, one expects the acceptance of the exchange move to equally depend upon the formulation.

The quantum Monte Carlo code in the ALF project carries out parallel-tempering runs when the script `configure.sh` is called with the argument `Tempering` before compilation, see Sec. 6.1.

2.2.6. Langevin dynamics

For models that include continuous real fields $\boldsymbol{s} \equiv \{s_{k,\tau}\}$ there is the option of using Langevin dynamics for the updating scheme, by setting the variable `Langevin` to `.true.`. This corresponds to a stochastic differential equation for the fields. They acquire a discrete Langevin time t_l with step width δt_l and satisfy the stochastic differential equation

$$\boldsymbol{s}(t_l + \delta t_l) = \boldsymbol{s}(t_l) - \frac{\partial S(C)}{\partial \boldsymbol{s}(t_l)}\delta t_l + \sqrt{2\delta t_l}\boldsymbol{\eta}(t_l). \tag{43}$$

Here, $\boldsymbol{\eta}(t_l)$ are independent Gaussian stochastic variables satisfying:

$$\langle \eta_{k,\tau}(t_l)\rangle_\eta = 0 \text{ and } \langle \eta_{k,\tau}(t_l)\eta_{k',\tau'}(t'_l)\rangle_\eta = \delta_{k,k'}\delta_{\tau,\tau'}\delta_{t_l,t'_l}. \tag{44}$$

We refer the reader to Ref. [69] for a more in depth introduction to stochastic differential equations. To see that the above indeed produces the desired probability distribution in the long Langevin time limit, we can transform the Langevin equation into the corresponding Fokker-Plank one. Let $P(\boldsymbol{s}, t_l)$ be the distribution of fields at Langevin time t_l. Then,

$$P(\boldsymbol{s}, t_l + \delta t_l) = \int D\boldsymbol{s}' P(\boldsymbol{s}', t_l)\left\langle \delta\left(\boldsymbol{s} - \left[\boldsymbol{s}' - \frac{\partial S(\boldsymbol{s}')}{\partial \boldsymbol{s}'}\delta t_l + \sqrt{2\delta t_l}\boldsymbol{\eta}(t_l)\right]\right)\right\rangle_\eta, \tag{45}$$

where δ corresponds to the $L_{\text{trot}} M_I$ dimensional Dirac δ-function. Taylor expanding up to order δt_l and averaging over the stochastic variable yields:

$$P(\boldsymbol{s}, t_l + \delta t_l) = \int D\boldsymbol{s'} P(\boldsymbol{s}', t_l) \times \left(\delta\left(\boldsymbol{s}' - \boldsymbol{s}\right) - \frac{\partial S(\boldsymbol{s'})}{\partial \boldsymbol{s'}} \frac{\partial}{\partial \boldsymbol{s'}} \delta\left(\boldsymbol{s}' - \boldsymbol{s}\right) \delta t_l + \frac{\partial}{\partial \boldsymbol{s'}} \frac{\partial}{\partial \boldsymbol{s'}} \delta\left(\boldsymbol{s}' - \boldsymbol{s}\right) \delta t_l \right) + \mathcal{O}\left(\delta t_l^2\right). \tag{46}$$

Partial integration and taking the limit of infinitesimal time steps gives the Fokker-Plank equation

$$\frac{\partial}{\partial t_l} P(\boldsymbol{s}, t_l) = \frac{\partial}{\partial \boldsymbol{s}} \left(P(\boldsymbol{s}, t_l) \frac{\partial S(\boldsymbol{s})}{\partial \boldsymbol{s}} + \frac{\partial P(\boldsymbol{s}, t_l)}{\partial \boldsymbol{s}} \right). \tag{47}$$

The stationary, $\frac{\partial}{\partial t_l} P(\boldsymbol{s}, t_l) = 0$, normalizable, solution to the above equation corresponds to the desired probability distribution:

$$P(\boldsymbol{s}) = \frac{e^{-S(\boldsymbol{s})}}{\int D\boldsymbol{s} e^{-S(\boldsymbol{s})}}. \tag{48}$$

In order to formulate the Langevin dynamics, we will need to estimate the forces:

$$\frac{\partial S(C)}{\partial s_{k,\tau}} = \frac{\partial S_{0,I}(C)}{\partial s_{k,\tau}} + \frac{\partial S^F(C)}{\partial s_{k,\tau}}, \tag{49}$$

with the fermionic part of the action

$$S^F(C) = -\ln \left\{ \prod_{s=1}^{N_{\text{fl}}} \left[\det \left(\mathbb{1} + \prod_{\tau=1}^{L_{\text{Trotter}}} \prod_{k=1}^{M_V} e^{\sqrt{-\Delta\tau U_k} \eta_{k,\tau} \boldsymbol{V}^{(ks)}} \prod_{k=1}^{M_I} e^{-\Delta\tau s_{k,\tau} \boldsymbol{I}^{(ks)}} \prod_{k=1}^{M_T} e^{-\Delta\tau \boldsymbol{T}^{(ks)}} \right) \right]^{N_{\text{col}}} \right\}. \tag{50}$$

The forces must be bounded for Langevin dynamics to work well. If this condition is violated the results produced by the code lose their reliability.

One possible source of divergence is the determinant in the fermionic action. Zeros lead to unbounded forces and, in order to circumvent this problem at least partially, we adopt a variable time step strategy in the code. The user provides an upper bound to the fermion force, `Max_Force` and, if the maximal force in a configuration, `Max_Force_Conf`, is larger than `Max_Force`, then the time step is rescaled as

$$\tilde{\delta t_l} = \frac{\texttt{Max\_Force} * \delta t_l}{\texttt{Max\_Force\_Conf}}. \tag{51}$$

With the adaptive time step, averages are computed as:

$$\langle \hat{O} \rangle = \frac{\sum_n (\tilde{\delta t_l})_n \langle\langle \hat{O} \rangle\rangle_{(C_n)}}{\sum_n (\tilde{\delta t_l})_n}. \tag{52}$$

In order to use Langevin dynamics the user also has to provide the Langevin time step `Delta_tau_Langevin` and the maximal force `Max_Force` in the `parameter` file. The routine `Langevin_update` in the module `Langevin_update_mod.F90` computes the fermion forces for a general model

$$\frac{\partial S^F(C)}{\partial s_{k,\tau}} = \Delta\tau N_{\text{col}} \sum_{s=1}^{N_{\text{fl}}} \text{Tr} \left[\boldsymbol{I}^{(ks)} \left(\mathbb{1} - \boldsymbol{G}^{(s)}(k,\tau) \right) \right]. \tag{53}$$

Here we introduce a Green function that depends on the time slice τ and the interaction term k to which the corresponding field $s_{k,\tau}$ belongs:

$$G^{(s)}_{x,y}(k,\tau) = \frac{\text{Tr}\left[\hat{U}^{<}_{(s)}(k,\tau) \hat{c}_{x,s} \hat{c}^{\dagger}_{y,s} \hat{U}^{>}_{(s)}(k,\tau) \right]}{\text{Tr}\left[\hat{U}^{<}_{(s)}(k,\tau) \hat{U}^{>}_{(s)}(k,\tau) \right]}, \tag{54}$$

where the following definitions are used

$$\hat{U}^{<}_{(s)}(k',\tau') = \prod_{\tau=\tau'+1}^{L_{\text{Trotter}}} \left(\hat{U}_{(s)}(\tau)\right) \prod_{k=1}^{M_V} e^{\sqrt{-\Delta\tau U_k}\eta_{k,\tau'}\hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{V}^{(ks)}\hat{\boldsymbol{c}}_s} \prod_{k=k'+1}^{M_I} e^{-\Delta\tau s_{k,\tau'}\hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{I}^{(ks)}\hat{\boldsymbol{c}}_s}, \tag{55}$$

$$\hat{U}^{>}_{(s)}(k',\tau') = \prod_{k=1}^{k'} e^{-\Delta\tau s_{k,\tau'}\hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{I}^{(ks)}\hat{\boldsymbol{c}}_s} \prod_{k=1}^{M_T} e^{-\Delta\tau \hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{T}^{(ks)}\hat{\boldsymbol{c}}_s} \prod_{\tau=1}^{\tau'-1} \left(\hat{U}_{(s)}(\tau)\right), \tag{56}$$

$$\hat{U}_{(s)}(\tau) = \prod_{k=1}^{M_V} e^{\sqrt{-\Delta\tau U_k}\eta_{k,\tau}\hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{V}^{(ks)}\hat{\boldsymbol{c}}_s} \prod_{k=1}^{M_I} e^{-\Delta\tau s_{k,\tau}\hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{I}^{(ks)}\hat{\boldsymbol{c}}_s} \prod_{k=1}^{M_T} e^{-\Delta\tau \hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{T}^{(ks)}\hat{\boldsymbol{c}}_s}. \tag{57}$$

The vector $\hat{\boldsymbol{c}}^{\dagger}_s$ contains all fermionic operators $\hat{c}^{\dagger}_{x,s}$ of flavor s. The fermion forces are passed to the routine `Ham_Langevin_update` in the Hamiltonian file, where the user has to define the update rule for the fields according to Eq. (43). During each sweep all fields are updated and the Langevin time is incremented by δt_l. All updates are accepted to ensure ergodicity. At the end of a run, the mean and maximal forces encountered during the run are printed out in the info file.

The great advantage of the Langevin updating scheme is the absence of update rejection, meaning that all fields are updated at each step. As mentioned above, the price we pay for using Langevin dynamics is ensuring that forces show no singularities, and two other potential issues should be highlighted:

- Langevin dynamics will be carried out at a finite Langevin time step, thereby introducing a further source of systematic error.
- The factor $\sqrt{2\delta t_l}$ multiplying the stochastic variable makes the noise dominant on short time scales. On these time scales Langevin dynamics essentially corresponds to a random walk. This has the advantage of allowing one to circumvent potential barriers, but may render the updating scheme less efficient than the hybrid molecular dynamics approach.

We have tested the code for a 6-site Hubbard chain at half-filling at $U/t = 4$, $\beta t = 4$. The Hubbard interaction can also be decoupled using a continuous HS transformation, where we introduce a real auxiliary field $s_{i,\tau}$ for every lattice site i and time slice τ. When the HS fields are coupled to the z-component of the magnetization (see Sec. 9.1), the partition function can be written as

$$Z = \int \left(\prod_{\tau=1}^{L_{\text{Trotter}}} \prod_{i=1}^{N_{\text{unit-cell}}} \frac{ds_{i,\tau}}{\sqrt{2\pi}} e^{-\frac{1}{2}s^2_{i,\tau}} \right) \times \prod_{s=\uparrow,\downarrow} \det \left(\mathbb{1} + \prod_{\tau=1}^{L_{\text{Trotter}}} \prod_{i=1}^{N_{\text{unit-cell}}} \left(e^{-\sqrt{\Delta\tau U} s_{i,\tau} \boldsymbol{V}^{(is)}} \right) e^{-\Delta\tau \boldsymbol{T}} \right) + \mathcal{O}(\Delta\tau^2). \tag{58}$$

The flavor dependent interaction matrices have only one non-vanishing entry $V^{(i,s=\uparrow)}_{x,y} = \delta_{x,y}\delta_{x,i}$ and $V^{(i,s=\downarrow)}_{x,y} = -\delta_{x,y}\delta_{x,i}$ respectively. The forces of the Hubbard model are given by:

$$\frac{\partial S(C)}{\partial s_{i,\tau}} = s_{i,\tau} - \sqrt{\Delta\tau U} \sum_{s=\uparrow,\downarrow} \text{Tr}\left[\boldsymbol{V}^{(is)} \left(\mathbb{1} - \boldsymbol{G}^{(s)}(i,\tau) \right) \right], \tag{59}$$

where the Green function is defined by eq. (54) with

$$\hat{U}^{<}_{(s)}(i',\tau') = \prod_{\tau=\tau'+1}^{L_{\text{Trotter}}} \left(\hat{U}_{(s)}(\tau)\right) \prod_{i=i'+1}^{N_{\text{unit-cell}}} e^{-\sqrt{\Delta\tau U} s_{i,\tau'} \hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{V}^{(is)} \hat{\boldsymbol{c}}_s}, \tag{60}$$

$$\hat{U}^{>}_{(s)}(i',\tau') = \prod_{i=1}^{i'} \left(e^{-\sqrt{\Delta\tau U} s_{i,\tau'} \hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{V}^{(is)} \hat{\boldsymbol{c}}_s} \right) e^{-\Delta\tau \hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{T} \hat{\boldsymbol{c}}_s} \prod_{\tau=1}^{\tau'-1} \left(\hat{U}_{(s)}(\tau)\right), \tag{61}$$

$$\hat{U}_{(s)}(\tau) = \prod_{i=1}^{N_{\text{unit-cell}}} \left(e^{-\sqrt{\Delta\tau U} s_{i,\tau} \hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{V}^{(is)} \hat{\boldsymbol{c}}_s} \right) e^{-\Delta\tau \hat{\boldsymbol{c}}^{\dagger}_s \boldsymbol{T} \hat{\boldsymbol{c}}_s}. \tag{62}$$

One can show that for periodic boundary conditions the forces are not bounded and to make sure that the program does not crash we have set `Max_Force = 1.5`.
The discrete variable code gives

$$\langle \hat{H} \rangle = -3.4684 \pm 0.0007, \tag{63}$$

while the Langevin code at $\delta t_l = 0.001$ yields

$$\langle \hat{H} \rangle = -3.457 \pm 0.010 \tag{64}$$

and at $\delta t_l = 0.01$

$$\langle \hat{H} \rangle = -3.495 \pm 0.007. \tag{65}$$

At $\delta t_l = 0.001$ the maximal force that occurred during the run was 112, whereas at $\delta t_l = 0.01$ it grew to 524. In both cases the average force was given by 0.45. For larger values of δt_l the maximal force grows and the fluctuations on the energy become larger. ($\langle \hat{H} \rangle = -3.718439 \pm 0.206469$ at $\delta t_l = 0.02$. For this parameter set the maximal force we encountered during the run was of 1658.)

Controlling Langevin dynamics when the action has logarithmic divergences is a challenge, and it is not clear that the results will be satisfying. For our specific problem we can solve this issue by considering open boundary conditions. Following an argument put forward in [49], we can show, using world lines, that the determinant is always positive. In this case the action does not have logarithmic divergences and the Langevin dynamics works beautifully well, see Fig. 1.

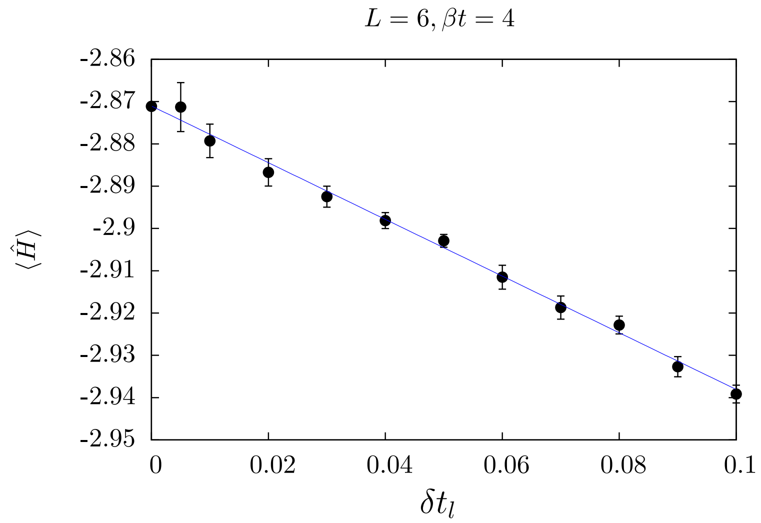

Figure 1: Total energy for the 6-site Hubbard chain at $U/t = 4$, $\beta t = 4$ and with open boundary conditions. Here one can show that the determinant is always positive such that no singularities occur in the action, and consequently the Langevin dynamics works very well. The data point at $\delta t_l = 0$ stems from running the discrete field code with coupling of the field to the z-component of the magnetization. The extrapolated value of the energy reads -2.8710 ± 0.0011 and the reference result from the discrete code is -2.8711 ± 0.0004. Throughout the runs the maximal force was always less than the threshold of 1.5.

2.3. The Trotter error and checkerboard decomposition

2.3.1. Asymmetric Trotter decomposition

In practice, many applications are carried out at finite imaginary time steps, and it is important to understand the consequences of the Trotter error. How does it scale with system size and what symmetries does it break? In particular, if one is investigating a critical point, then one should understand if the potential symmetry breaking associated with the Trotter decomposition generates relevant operators.

To at best describe the workings of the ALF code, we divide the Hamiltonian into hopping terms $\hat{\mathcal{H}}_T$ and interaction terms $\hat{\mathcal{H}}_V + \hat{\mathcal{H}}_I + \hat{\mathcal{H}}_{0,I}$. Let

$$\hat{\mathcal{H}}_T = \sum_{i=1}^{N_T} \sum_{k \in \mathcal{S}_i^T} \hat{T}^{(k)} \equiv \sum_{i=1}^{N_T} \hat{T}_i \tag{66}$$

Here the decomposition follows the rule that if k and k' belong to the same set $\mathcal{S}_i^T$ then $\left[\hat{T}^{(k)}, \hat{T}^{(k')}\right] = 0$. An important example is the checkerboard decomposition. For the square lattice we can decouple the nearest neighbor hopping into $N_T = 4$ groups, each group consisting of two site hopping processes. This type of checkerboard decomposition is activated for a set of predefined lattices by setting the flag `Checkerboard` to `.true.` We will carry out the same decomposition for the interaction:

$$\hat{\mathcal{H}}_V + \hat{\mathcal{H}}_I + \hat{\mathcal{H}}_{0,I} = \sum_{i=1}^{N_I} \hat{O}_i \tag{67}$$

where each $\hat{O}_i$ contains a set of commuting terms. For instance, for the Hubbard model, the above reduces to $U \sum_{\boldsymbol{i}} \hat{n}_{\boldsymbol{i},\uparrow}\hat{n}_{\boldsymbol{i},\downarrow}$ such the $N_T = 1$ and $\hat{O}_1 = U \sum_{\boldsymbol{i}} \hat{n}_{\boldsymbol{i},\uparrow}\hat{n}_{\boldsymbol{i},\downarrow}$.

The default Trotter decomposition in the ALF code is based on the equation:

$$e^{-\Delta\tau\left(\hat{A}+\hat{B}\right)} = e^{-\Delta\tau\hat{A}}e^{-\Delta\tau\hat{B}} + \frac{\Delta\tau^2}{2}\left[\hat{B}, \hat{A}\right] + \mathcal{O}\left(\Delta\tau^3\right) \tag{68}$$

Using iteratively the above the single time step is given by:

$$e^{-\Delta\tau\mathcal{H}} = \prod_{i=1}^{N_O} e^{-\Delta\tau\hat{O}_i} \prod_{j=1}^{N_T} e^{-\Delta\tau\hat{T}_j} + \underbrace{\frac{\Delta\tau^2}{2}\left(\sum_{i=1}^{N_O}\sum_{j=1}^{N_T}\left[\hat{T}_j, \hat{O}_i\right] + \sum_{j'}^{N_T-1}\left[\hat{T}_{j'}, \hat{T}_{j'}^{>}\right] + \sum_{i'=1}^{N_O-1}\left[\hat{O}_{i'}, \hat{O}_{i'}^{>}\right]\right)}_{\equiv\Delta\tau\hat{\lambda}_1} + \mathcal{O}\left(\Delta\tau^3\right). \tag{69}$$

In the above, we have introduced the short hand notation

$$\hat{T}_n^{>} = \sum_{j=n+1}^{N_T} \hat{T}_j, \text{ and } \hat{O}_n^{>} = \sum_{j=n+1}^{N_O} \hat{O}_j. \tag{70}$$

The full propagation then reads

$$\hat{U}_{\text{Approx}} = \left(\prod_{i=1}^{N_O} e^{-\Delta\tau\hat{O}_i} \prod_{j=1}^{N_T} e^{-\Delta\tau\hat{T}_j}\right) = e^{-\beta\left(\hat{H}+\hat{\lambda}_1\right)} + \mathcal{O}\left(\Delta\tau^2\right) = e^{-\beta\hat{H}} - \int_0^\beta d\tau e^{-(\beta-\tau)\hat{H}}\hat{\lambda}_1 e^{-\tau\hat{H}} + \mathcal{O}(\Delta\tau^2). \tag{71}$$

The last step follows from time-dependent perturbation theory. The following comments are in order:

- The error is anti-Hermitian since $\hat{\lambda}_1^\dagger = -\hat{\lambda}_1$. This has for consequence that if all the operators as well as the quantity being measured are all simultaneously real representable, then the prefactor of the linear in $\Delta\tau$ error vanishes since it ultimately corresponds to computing the trace of an anti-symmetric matrix. This *lucky* cancellation was put forward in Ref. [58]. Hence, under this assumption − which is certainly valid for the Hubbard model considered in Fig. 2 − the systematic error is of order $\Delta\tau^2$.

- The biggest drawback of the above decomposition is that the imaginary-time propagation is not Hermitian. This can lead to acausal features in imaginary-time correlation functions [70]. To be more precise, the eigenvalues of $H_{\text{Approx}} = -\frac{1}{\beta}\log U_{\text{Approx}}$ need not be real such that imaginary-time displaced correlation functions can have oscillatory behavior as a function of imaginary time. This is shown in Fig. 2 (a) that plots the absolute value of local time-displaced Green function for the Honeycomb lattice at $U/t = 2$. Sign changes of this quantity involve zeros that, on the considered log-scale, correspond to negative divergences. As we will see next, this issue can be solved by considering a symmetric Trotter decomposition.

2.3.2. Symmetric Trotter decomposition

To address the issue described above, the ALF library provides the possibility to use a symmetric Trotter decomposition,

$$e^{-\Delta\tau\left(\hat{A}+\hat{B}\right)} = e^{-\Delta\tau\hat{A}/2}e^{-\Delta\tau\hat{B}}e^{-\Delta\tau\hat{A}/2} + \frac{\Delta\tau^3}{12}\left[2\hat{A}+\hat{B}, \left[\hat{B}, \hat{A}\right]\right] + \mathcal{O}\left(\Delta\tau^5\right), \tag{72}$$

which is activated with the `Symm` flag. In order to apply the expression above to our time step, we first write

$$e^{-\Delta\tau\mathcal{H}} = e^{-\frac{\Delta\tau}{2}\sum_{j=1}^{N_T}\hat{T}_j} e^{-\Delta\tau\sum_{i=1}^{N_I}\hat{O}_i} e^{-\frac{\Delta\tau}{2}\sum_{j=1}^{N_T}\hat{T}_j} + \underbrace{\frac{\Delta\tau^3}{12}\left[2\hat{T}_0^{>} + \hat{O}_0^{>}, \left[\hat{O}_0^{>}, \hat{T}_0^{>}\right]\right]}_{\equiv\Delta\tau\hat{\lambda}_{TO}} + \mathcal{O}\left(\Delta\tau^5\right). \quad (73)$$

Then, using,

$$e^{-\Delta\tau\sum_i^{N_I}\hat{O}_i} = \left(\prod_{i=1}^{N_O-1} e^{-\frac{\Delta\tau}{2}\hat{O}_i}\right) e^{-\Delta\tau\hat{O}_{N_O}} \left(\prod_{i=N_O-1}^{1} e^{-\frac{\Delta\tau}{2}\hat{O}_i}\right) + \underbrace{\frac{\Delta\tau^3}{12}\sum_{i=1}^{N_0-1}\left[2\hat{O}_i + \hat{O}_i^{>}, \left[\hat{O}_i^{>}, \hat{O}_i\right]\right]}_{\equiv\Delta\tau\hat{\lambda}_O} + \mathcal{O}\left(\Delta\tau^5\right) \quad (74)$$

and

$$e^{-\frac{\Delta\tau}{2}\sum_j^{N_T}\hat{T}_j} = \left(\prod_{j=1}^{N_T-1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right) e^{-\frac{\Delta\tau}{2}\hat{T}_{N_T}} \left(\prod_{j=N_T-1}^{1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right) + \underbrace{\frac{\Delta\tau^3}{96}\sum_{j=1}^{N_T-1}\left[2\hat{T}_j + \hat{T}_j^{>}, \left[\hat{T}_j^{>}, \hat{T}_j\right]\right]}_{\equiv\Delta\tau\hat{\lambda}_T} + \mathcal{O}\left(\Delta\tau^5\right) \quad (75)$$

we can derive a closed equation for the free energy density:

$$\begin{aligned} f_{\text{Approx}} = -\frac{1}{\beta V}\log\text{Tr}\Bigg[& \left(\prod_{j=1}^{N_T-1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right) e^{-\frac{\Delta\tau}{2}\hat{T}_{N_T}} \left(\prod_{j=N_T-1}^{1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right) \\ & \left(\prod_{i=1}^{N_O-1} e^{-\frac{\Delta\tau}{2}\hat{O}_i}\right) e^{-\Delta\tau\hat{O}_{N_O}} \left(\prod_{i=N_O-1}^{1} e^{-\frac{\Delta\tau}{2}\hat{O}_i}\right) \\ & \left(\prod_{j=1}^{N_T-1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right) e^{-\frac{\Delta\tau}{2}\hat{T}_{N_T}} \left(\prod_{j=N_T-1}^{1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right)\Bigg]^{L_{\text{Trotter}}} \\ = f - \frac{1}{V}\langle\hat{\lambda}_{TO} + \hat{\lambda}_O + 2\hat{\lambda}_T\rangle + \mathcal{O}(\Delta\tau^5). & \end{aligned} \quad (76)$$

The following comments are in order:

- The approximate imaginary-time propagation from which the f_{Approx} is derived is Hermitian. Hence no spurious effects in imaginary-time correlation functions are to be expected. This is explicitly shown in Fig. 2(a).
- In Fig. 2(b) we have used the ALF-library with `Symm=.true.` with and without checkerboard decomposition. We still expect the systematic error to be of order $\Delta\tau^2$. However its prefactor is much smaller than that of the aforementioned anti-symmetric decomposition.
- We have taken the burden to evaluate explicitly the prefactor of the $\Delta\tau^2$ error on the free energy density. One can see that for Hamiltonians that are sums of local operators, $\langle\hat{\lambda}_{TO} + \hat{\lambda}_O + 2\hat{\lambda}_T\rangle$ will scale as the volume V of the system, such that the systematic error on the free energy density (and on correlations functions that can be computed by adding source terms) will be volume independent. For model Hamiltonians that are not sums of local terms, care must be taken. A conservative upper bound on the error is $\langle\hat{\lambda}_{TO} + \hat{\lambda}_O + 2\hat{\lambda}_T\rangle \propto \Delta\tau^2 V^3$, which means that, in order to maintain a constant systematic error for the free energy density, we have to keep $\Delta\tau V$ constant. Such a situation has been observed in Ref. [71].

Alternative symmetric second order methods as well as the issues with decompositions of higher order have been detailed in [66].

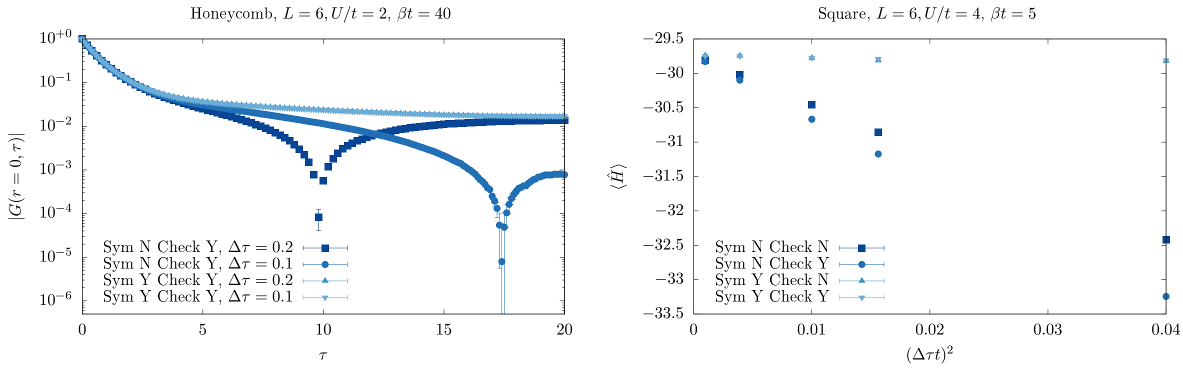

Figure 2: Analysis of Trotter systematic error. Left: We consider a 6×6 Hubbard model on the Honeycomb lattice, $U/t = 2$, half-band filling, inverse temperature $\beta t = 40$, and we have used an HS transformation that couples to the density. The figure plots the local-time displaced Green function. Right: Here we consider the 6×6 Hubbard model at $U/t = 4$, half-band filling, inverse temperature $\beta t = 5$, and we have used the HS transformation that couples to the z-component of spin. We provide data for the four combinations of the logical variables `Symm` and `Checkerboard`, where `Symm=.true.` (`.false.`) indicates a symmetric (asymmetric) Trotter decomposition has been used, and `Checkerboard=.true.` (`.false.`) that the checkerboard decomposition for the hopping matrix has (not) been used. The large deviations between different choices of `Symm` are here $\sim [T, [T, H]]$ as detailed in [66].

2.3.3. The `Symm` flag

If the `Symm` flag is set to true, then the program will automatically – for the set of predefined lattices and models – use the symmetric $\Delta\tau$ time step of the interaction and hopping terms.

To save CPU time when the `Symm` flag is on we carry out the following approximation:

$$\left[\left(\prod_{j=1}^{N_T-1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right) e^{-\frac{\Delta\tau}{2}\hat{T}_{N_T}} \left(\prod_{j=N_T-1}^{1} e^{-\frac{\Delta\tau}{4}\hat{T}_j}\right)\right]^2 \simeq \left(\prod_{j=1}^{N_T-1} e^{-\frac{\Delta\tau}{2}\hat{T}_j}\right) e^{-\Delta\tau\hat{T}_{N_T}} \left(\prod_{j=N_T-1}^{1} e^{-\frac{\Delta\tau}{2}\hat{T}_j}\right). \quad (77)$$

The above is consistent with the overall precision of the Trotter decomposition and more importantly conserves the Hermiticity of the propagation.

2.4. Stabilization - a peculiarity of the BSS algorithm

From the partition function in Eq. (16) it can be seen that, for the calculation of the Monte Carlo weight and of the observables, a long product of matrix exponentials has to be formed. In addition to that, we need to be able to extract the single-particle Green function for a given flavor index at, say, time slice $\tau = 0$. As mentioned above (cf. Eq. (20)), this quantity is given by:

$$\boldsymbol{G} = \left(\mathbb{1} + \prod_{\tau=1}^{L_{\text{Trotter}}} \boldsymbol{B}_\tau\right)^{-1}, \quad (78)$$

which can be recast as the more familiar linear algebra problem of finding a solution for the linear system

$$\left(\mathbb{1} + \prod_\tau \boldsymbol{B}_\tau\right) x = b. \quad (79)$$

The matrices $\boldsymbol{B}_\tau \in \mathbb{C}^{n\times n}$ depend on the lattice size as well as other physical parameters that can be chosen such that a matrix norm of $\boldsymbol{B}_\tau$ can be unbound in magnitude. From standard perturbation theory for linear systems, the computed solution $\tilde{x}$ would contain a relative error

$$\frac{|\tilde{x} - x|}{|x|} = \mathcal{O}\left(\epsilon\kappa_p\left(\mathbb{1} + \prod_\tau \boldsymbol{B}_\tau\right)\right), \quad (80)$$

where ϵ denotes the machine precision, which is 2^{-53} for IEEE double-precision numbers, and $\kappa_p(\boldsymbol{M})$ is the condition number of the matrix $\boldsymbol{M}$ with respect to the matrix p-norm. Due to $\prod_\tau \boldsymbol{B}_\tau$ containing exponentially large and small scales, as can be seen in Eq. (16), a straightforward inversion turns out to be completely ill-suited. That would lead the condition number, as a function of increasing inverse temperature, to grow exponentially, rendering the computed solution $\tilde{x}$ meaningless.

In order to circumvent this, more sophisticated methods have to be employed. As a first step, assuming that the multiplication of `NWrap` $\boldsymbol{B}$ matrices has an acceptable condition number and, for simplicity, that `NWrap` is a divisor of L_{Trotter}, we can write:

$$\boldsymbol{G} = \left(\mathbb{1} + \prod_{i=1}^{\frac{L_{\text{Trotter}}}{\texttt{NWrap}}} \underbrace{\prod_{\tau=1}^{\texttt{NWrap}} \boldsymbol{B}_{(i-1)\cdot\texttt{NWrap}+\tau}}_{\equiv \mathcal{B}_i} \right)^{-1} . \tag{81}$$

The default stabilization strategy in the auxiliary-field QMC implementation of the ALF project, is then to form a product of QR-decompositions, which was proven to be weakly backwards stable in [72]. The key idea is to efficiently separate the scales of a matrix from the orthogonal part of a matrix. This can be achieved using a QR decomposition of a matrix $\boldsymbol{A}$ in the form $\boldsymbol{A}_i = \boldsymbol{Q}_i \boldsymbol{R}_i$. The matrix $\boldsymbol{Q}_i$ is unitary and hence in the usual 2-norm it holds that $\kappa_2(\boldsymbol{Q}_i) = 1$. To get a handle on the condition number of $\boldsymbol{R}_i$ we will form the diagonal matrix

$$(\boldsymbol{D}_i)_{n,n} = |(\boldsymbol{R}_i)_{n,n}| \tag{82}$$

and set $\tilde{\boldsymbol{R}}_i = \boldsymbol{D}_i^{-1} \boldsymbol{R}_i$ This gives the decomposition

$$\boldsymbol{A}_i = \boldsymbol{Q}_i \boldsymbol{D}_i \tilde{\boldsymbol{R}}_i. \tag{83}$$

The matrix $\boldsymbol{D}_i$ now contains the row norms of the original $\boldsymbol{R}_i$ matrix and hence attempts to separate off the total scales of the problem from $\boldsymbol{R}_i$. This is similar in spirit to the so-called matrix equilibration which tries to improve the condition number of a matrix through suitably chosen column and row scalings. Due to a theorem by van der Sluis [73] we know that the choice in Eq. (82) is almost optimal among all diagonal matrices $\boldsymbol{D}$ from the space of diagonal matrices $\mathcal{D}$, in the sense that

$$\kappa_p((\boldsymbol{D}_i)^{-1} \boldsymbol{R}_i) \leq n^{1/p} \min_{\boldsymbol{D} \in \mathcal{D}} \kappa_p(\boldsymbol{D}^{-1} \boldsymbol{R}_i).$$

Now, given an initial decomposition of $\boldsymbol{A}_{j-1} = \prod_i \mathcal{B}_i = \boldsymbol{Q}_{j-1} \boldsymbol{D}_{j-1} \boldsymbol{T}_{j-1}$ an update $\mathcal{B}_j \boldsymbol{A}_{j-1}$ is formed in the following three steps:

1. Form $\boldsymbol{M}_j = (\mathcal{B}_j \boldsymbol{Q}_{j-1}) \boldsymbol{D}_{j-1}$. Note the parentheses.
2. Do a QR decomposition of $\boldsymbol{M}_j = \boldsymbol{Q}_j \boldsymbol{D}_j \boldsymbol{R}_j$. This gives the final $\boldsymbol{Q}_j$ and $\boldsymbol{D}_j$.
3. Form the updated $\boldsymbol{T}$ matrices $\boldsymbol{T}_j = \boldsymbol{R}_j \boldsymbol{T}_{j-1}$.

This is a stable but expensive method for calculating the matrix product. Here is where `NWrap` comes into play: it specifies the number of plain multiplications performed between the QR decompositions just described, so that `NWrap` $= 1$ corresponds to always performing QR decompositions whereas larger values define longer intervals where no QR decomposition will be performed. Whenever we perform a stabilization, we compare the old result (fast updates) with the new one (recalculated from the QR stabilized matrices). The difference is documented as the stability, both for the Green function and for the sign (of the determinant) The effectiveness of the stabilization *has* to be judged for every simulation from the output file `info` (Sec. 5.7.2). For most simulations there are two values to look out for:

- `Precision Green`
- `Precision Phase`

The Green function, as well as the average phase, are usually numbers with a magnitude of $\mathcal{O}(1)$. For that reason we recommend that `NWrap` is chosen such that the mean precision is of the order of 10^{-8} or better (for further recommendations see Sec. 6.3). We include typical values of `Precision Phase` and of the mean and the maximal values of `Precision Green` in the example simulations discussed in Sec. 7.7.

3. Auxiliary Field Quantum Monte Carlo: projective algorithm

The projective approach is the method of choice if one is interested in ground-state properties. The starting point is a pair of trial wave functions, $|\Psi_{T,L/R}\rangle$, that are not orthogonal to the ground state $|\Psi_0\rangle$:

$$\langle \Psi_{T,L/R}|\Psi_0\rangle \neq 0. \tag{84}$$

The ground-state expectation value of any observable $\hat{O}$ can then be computed by propagation along the imaginary time axis:

$$\frac{\langle \Psi_0|\hat{O}|\Psi_0\rangle}{\langle \Psi_0|\Psi_0\rangle} = \lim_{\theta\to\infty} \frac{\langle \Psi_{T,L}|e^{-\theta \hat{H}} e^{-(\beta-\tau)\hat{H}} \hat{O} e^{-\tau \hat{H}} e^{-\theta \hat{H}}|\Psi_{T,R}\rangle}{\langle \Psi_{T,L}|e^{-(2\theta+\beta)\hat{H}}|\Psi_{T,R}\rangle}, \tag{85}$$

where β defines the imaginary time range where observables (time displaced and equal time) are measured and τ varies from 0 to β in the calculation of time-displace observables. The simulations are carried out at large but finite values of θ so as to guarantee convergence to the ground state within the statistical uncertainly. The trial wave functions are determined up to a phase, and the program uses this gauge choice to guarantee that

$$\langle \Psi_{T,L}|\Psi_{T,R}\rangle > 0. \tag{86}$$

In order to use the projective version of the code, the model's namespace in the `parameter` file must set `projector=.true.` and specify the value of the projection parameter `Theta`, as well as the imaginary time interval `Beta` in which observables are measured.

Note that time-displaced correlation functions are computed for a τ ranging from 0 to β. The implicit assumption in this formulation is that the projection parameter `Theta` suffices to reach the ground state. Since the computational time scales linearly with `Theta` large projections parameters are computationally not expensive.

3.1. Specification of the trial wave function

For each flavor, one needs to specify a left and a right trial wave function. In the ALF, they are assumed to be the ground state of single-particle trial Hamiltonians $\hat{H}_{T,L/R}$ and hence correspond to a single Slater determinant each. More specifically, we consider a single-particle Hamiltonian with the same symmetries (color and flavor) as the original Hamiltonian:

$$\hat{H}_{T,L/R} = \sum_{\sigma=1}^{N_{\mathrm{col}}} \sum_{s=1}^{N_{\mathrm{fl}}} \sum_{x,y}^{N_{\mathrm{dim}}} \hat{c}^{\dagger}_{x\sigma s} h^{(s,L/R)}_{xy} \hat{c}_{y\sigma s}. \tag{87}$$

Ordering the eigenvalues of the Hamiltonian in ascending order yields the ground state

$$|\Psi_{T,L/R}\rangle = \prod_{\sigma=1}^{N_{\mathrm{col}}} \prod_{s=1}^{N_{\mathrm{fl}}} \prod_{n=1}^{N_{\mathrm{part},s}} \left(\sum_{x=1}^{N_{\mathrm{dim}}} \hat{c}^{\dagger}_{x\sigma s} U^{(s,L/R)}_{x,n} \right) |0\rangle, \tag{88}$$

where

$$U^{\dagger,(s,L/R)} h^{(s,L/R)} U^{(s,L/R)} = \mathrm{Diag}\left(\epsilon^{(s,L/R)}_1, \cdots, \epsilon^{(s,L/R)}_{N_{\mathrm{dim}}} \right). \tag{89}$$

The trial wave function is hence completely defined by the set of orthogonal vectors $U^{(s,L/R)}_{x,n}$ for n ranging from 1 to the number of particles in each flavor sector, $N_{\mathrm{part},s}$. This information is stored in the `WaveFunction` type defined in the module `WaveFunction_mod` (see Sec. 5.5). Note that, owing to the SU($\mathrm{N_{col}}$) symmetry, the color index is not necessary to define the trial wave function. The user will have to specify the trial wave function in the following way:

```
Do s = 1, N_fl
   Do x = 1,Ndim
      Do n = 1, N_part(s)
         WF_L(s)%P(x,n)   =   U^{(s,L)}_{x,n}
         WF_R(s)%P(x,n)   =   U^{(s,R)}_{x,n}
      Enddo
   Enddo
Enddo
```

In the above WF_L and WF_R are WaveFunction arrays of length N_{fl}. ALF comes with a set of predefined trial wave functions, see Sec. 8.5.

Generically, the unitary matrix will be generated by a diagonalization routine such that if the ground state for the given particle number is degenerate, the trial wave function has a degree of ambiguity and does not necessarily share the symmetries of the Hamiltonian $\hat{H}_{T,L/R}$. Since symmetries are the key for guaranteeing the absence of the negative sign problem, violating them in the choice of the trial wave function can very well lead to a sign problem. It is hence recommended to define the trial Hamiltonians $\hat{H}_{T,L/R}$ such that the ground state for the given particle number is non-degenerate. That can be checked using the value of WL_L/R(s)%Degen, which stores the energy difference between the last occupied and fist un-occupied single particle state. If this value is greater than zero, then the trial wave function is non-degenerate and hence has all the symmetry properties of the trial Hamiltonians, $\hat{H}_{T,L/R}$. When the projector variable is set to .true., this quantity is listed in the info file.

3.2. Some technical aspects of the projective code.

If one is interested solely in zero-temperature properties, the projective code offers many advantages. This comes from the related facts that the Green function matrix is a projector, and that scales can be omitted.

In the projective algorithm, it is known [6] that

$$G(x,\sigma,s,\tau|x',\sigma,s,\tau) = \left[1 - U^{>}_{(s)}(\tau)\left(U^{<}_{(s)}(\tau)U^{>}_{(s)}(\tau)\right)^{-1}U^{<}_{(s)}(\tau)\right]_{x,x'} \tag{90}$$

with

$$U^{>}_{(s)}(\tau) = \prod_{\tau'=1}^{\tau} \boldsymbol{B}^{(s)}_{\tau'} P^{(s),R} \quad \text{and} \quad U^{<}_{(s)}(\tau) = P^{(s),L,\dagger} \prod_{\tau'=L_{\text{Trotter}}}^{\tau+1} \boldsymbol{B}^{(s)}_{\tau'}, \tag{91}$$

where $\boldsymbol{B}^{(s)}_{\tau}$ is given by Eq. (21) and $P^{(s),L/R}$ correspond to the $N_{\text{dim}} \times N_{\text{part},s}$ submatrices of $U^{(s),L/R}$. To see that scales can be omitted, we carry out a singular value decomposition:

$$U^{>}_{(s)}(\tau) = \tilde{U}^{>}_{(s)}(\tau)d^{>}v^{>} \quad \text{and} \quad U^{<}_{(s)}(\tau) = v^{<}d^{<}\tilde{U}^{<}_{(s)}(\tau) \tag{92}$$

such that $\tilde{U}^{>}_{(s)}(\tau)$ corresponds to a set of column-wise orthogonal vectors. It can be readily seen that scales can be omitted, since

$$G(x,\sigma,s,\tau|x',\sigma,s,\tau) = \left[1 - \tilde{U}^{>}_{(s)}(\tau)\left(\tilde{U}^{<}_{(s)}(\tau)\tilde{U}^{>}_{(s)}(\tau)\right)^{-1}\tilde{U}^{<}_{(s)}(\tau)\right]_{x,x'}. \tag{93}$$

Hence, stabilization is never an issue for the projective code, and arbitrarily large projection parameters can be reached.

The form of the Green function matrix implies that it is a projector: $G^2 = G$. This property has been used in Ref. [74] to very efficiently compute imaginary-time-displaced correlation functions.

3.3. Comparison of finite and projective codes.

The finite temperature code operates in the grand canonical ensemble, whereas in the projective approach the particle number is fixed. On finite lattices, the comparison between both approaches can only be made at a temperature scale below which a finite-sized charge gap emerges. In Fig. 3 we consider a semi-metallic phase as realized by the Hubbard model on the Honeycomb lattice at $U/t = 2$. It is evident that, at a scale below which charge fluctuations are suppressed, both algorithms yield identical results.

4. Monte Carlo sampling

Error estimates in Monte Carlo simulations are based on the central limit theorem [76] and can be delicate. This theorem requires independent measurements and a finite variance. In this subsection we will give examples of the issues that a user will have to look out for while using a Monte Carlo code. Those effects are part of the common lore of the field and we can only cover them briefly in this text. For a deeper understanding of the inherent issues of Markov-chain Monte Carlo methods we refer the

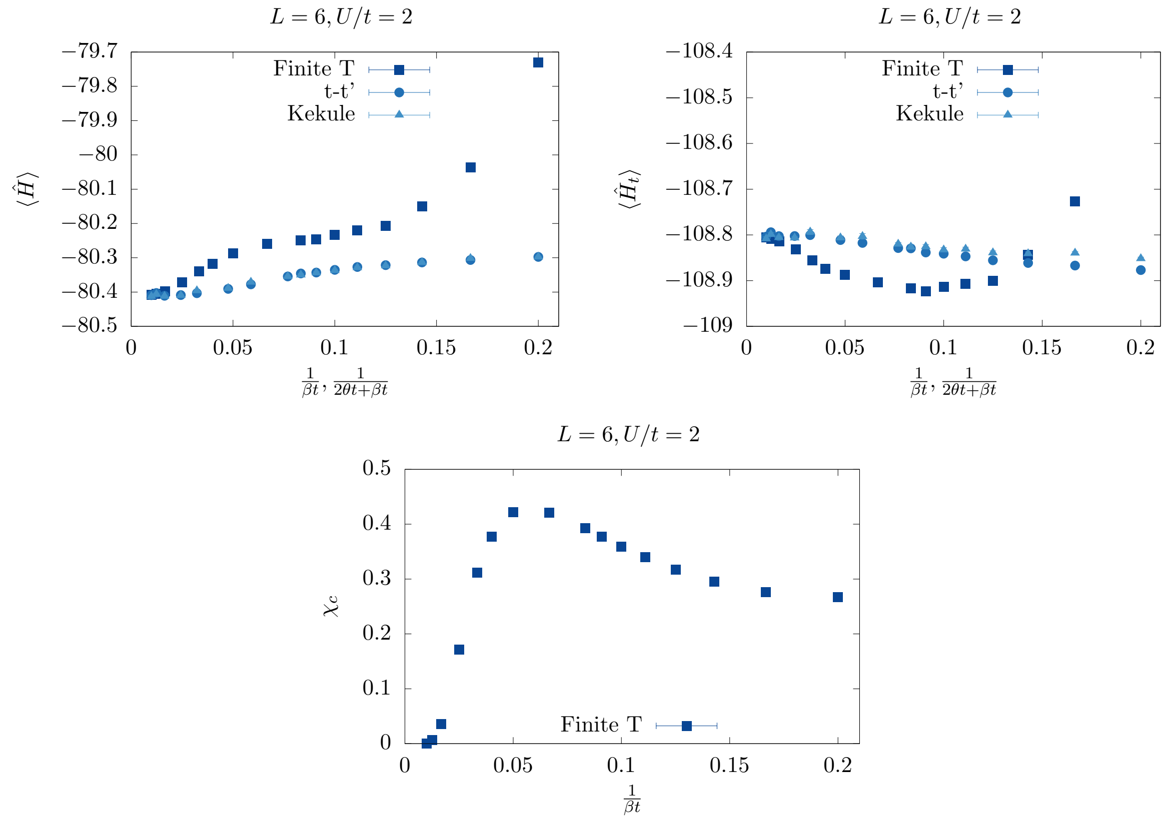

Figure 3: Comparison between the finite-temperature and projective codes for the Hubbard model on a 6×6 Honeycomb lattice at $U/t = 2$ and with periodic boundary conditions. For the projective code (blue and black symbols) $\beta t = 1$ is fixed, while θ is varied. In all cases we have $\Delta\tau t = 0.1$, no checkerboard decomposition, and a symmetric Trotter decomposition. For this lattice size and choice of boundary conditions, the non-interacting ground state is degenerate, since the Dirac points belong to the discrete set of crystal momenta. In order to generate the trial wave function we have lifted this degeneracy by either including a Kékulé mass term [25] that breaks translation symmetry (blue symbols), or by adding a next-next nearest neighbor hopping (black symbols) that breaks the symmetry nematically and shifts the Dirac points away from the zone boundary [75]. As apparent, both choices of trial wave functions yield the same answer, which compares very well with the finite temperature code at temperature scales below the finite-size charge gap.

reader to the pedagogical introduction in chapter 1.3.5 of Krauth [77], the overview article of Sokal [42], the more specialized literature by Geyer [78] and chapter 6.3 of Neal [79].

In general, one distinguishes local from global updates. As the name suggest, the local update corresponds to a small change of the configuration, e.g., a single spin flip of one of the $L_{\text{Trotter}}(M_I + M_V)$ field entries (see Sec. 2.2), whereas a global update changes a significant part of the configuration. The default update scheme of the ALF implementation are local updates, such that there is a minimum number of moves required for generating an independent configuration. The associated time scale is called the autocorrelation time, T_{auto}, and is generically dependent upon the choice of the observables.

Our unit of *sweeps* is defined such that each field is visited twice in a sequential propagation from $\tau = 0$ to $\tau = L_{\text{ Trotter}}$ and back. A single sweep will generically not suffice to produce an independent configuration. In fact, the autocorrelation time T_{auto} characterizes the required time scale to generate independent values of $\langle\langle \hat{O} \rangle\rangle_C$ for the observable O. This has several consequences for the Monte Carlo simulation:

- First of all, we start from a randomly chosen field configuration, such that one has to invest a time of *at least* one T_{auto}, but typically many more, in order to generate relevant, equilibrated configurations before reliable measurements are possible. This phase of the simulation is known as the warm-up or burn-in phase. In order to keep the code as flexible as possible (as different simulations might have different autocorrelation times), measurements are taken from the very beginning and, in the analysis phase, the parameter `n_skip` controls the number of initial bins that are ignored.

- Second, our implementation averages over bins with `NSWEEPS` measurements before storing the results on disk. The error analysis requires statistically independent bins in order to generate reliable confidence estimates. If the bins are instead too small (averaged over a period shorter then T_{auto}), then the error bars are typically underestimated. Most of the time, however, the autocorrelation time is unknown before the simulation is started and, sometimes, single runs long enough to generate appropriately sized bins are not feasible. For this reason, we provide a rebinning facility controlled by the parameter `N_rebin` that specifies the number of bins recombined into each new bin during the error analysis. One can test the suitability of a given bin size by verifying whether a increase in size changes the error estimate (For an explicit example, see Sec. 4.2 and the appendix of Ref. [53]).

  The `N_rebin` variable can be used to control a further issue. The distribution of the Monte Carlo estimates $\langle\langle \hat{O} \rangle\rangle_C$ is unknown, while a result in the form (mean $\pm$ error) assumes a Gaussian distribution. Every distribution with a finite variance turns into a Gaussian one once it is folded often enough (central limit theorem). Due to the internal averaging (folding) within one bin, many observables are already quite Gaussian. Otherwise one can increase `N_rebin` further, even if the bins are already independent [80].

- The third issue concerns time-displaced correlation functions. Even if the configurations are independent, the fields within the configuration are still correlated. Hence, the data for $S_{\alpha,\beta}(\boldsymbol{k}, \tau)$ (see Sec. 5.4; Eq. (116)) and $S_{\alpha,\beta}(\boldsymbol{k}, \tau + \Delta\tau)$ are also correlated. Setting the switch `N_Cov=1` triggers the calculation of the covariance matrix in addition to the usual error analysis. The covariance is defined by

$$COV_{\tau\tau'} = \frac{1}{N_{\text{Bin}}} \left\langle \left(S_{\alpha,\beta}(\boldsymbol{k},\tau) - \left\langle S_{\alpha,\beta}(\boldsymbol{k},\tau) \right\rangle \right) \left(S_{\alpha,\beta}(\boldsymbol{k},\tau') - \left\langle S_{\alpha,\beta}(\boldsymbol{k},\tau') \right\rangle \right) \right\rangle . \tag{94}$$

  An example where this information is necessary is the calculation of mass gaps extracted by fitting the tail of the time-displaced correlation function. Omitting the covariance matrix will underestimate the error.

4.1. The Jackknife resampling method

For each observable $\hat{A}, \hat{B}, \hat{C} \cdots$ the Monte Carlo program computes a data set of N_{Bin} (ideally) independent values where for each observable the measurements belong to the same statistical distribution. In the general case, we would like to evaluate a function of expectation values, $f(\langle \hat{A} \rangle, \langle \hat{B} \rangle, \langle \hat{C} \rangle \cdots)$ – see for example the expression (25) for the observable including reweighting – and are interested in the statistical estimates of its mean value and the standard error of the mean. A numerical method for the statistical analysis of a given function f which properly handles error propagation and correlations among the observables is the Jackknife method, which is, like the related Bootstrap method, a resampling scheme

[81]. Here we briefly review the *delete-1 Jackknife* scheme, which consists in generating N_{bin} new data sets of size $N_{\text{bin}} - 1$ by consecutively removing one data value from the original set. By $A_{(i)}$ we denote the arithmetic mean for the observable $\hat{A}$, without the i-th data value A_i, namely

$$A_{(i)} \equiv \frac{1}{N_{\text{Bin}} - 1} \sum_{k=1,\, k\neq i}^{N_{\text{Bin}}} A_k \; . \tag{95}$$

As the corresponding quantity for the function $f(\langle\hat{A}\rangle, \langle\hat{B}\rangle, \langle\hat{C}\rangle \cdots)$, we define

$$f_{(i)}(\langle\hat{A}\rangle, \langle\hat{B}\rangle, \langle\hat{C}\rangle \cdots) \equiv f(A_{(i)}, B_{(i)}, C_{(i)} \cdots) \; . \tag{96}$$

Following the convention in the literature, we will denote the final Jackknife estimate of the mean by $f_{(\cdot)}$ and its standard error by Δf. The Jackknife mean is given by

$$f_{(\cdot)}(\langle\hat{A}\rangle, \langle\hat{B}\rangle, \langle\hat{C}\rangle \cdots) = \frac{1}{N_{\text{Bin}}} \sum_{i=1}^{N_{\text{Bin}}} f_{(i)}(\langle\hat{A}\rangle, \langle\hat{B}\rangle, \langle\hat{C}\rangle \cdots) \; , \tag{97}$$

and the standard error, including bias correction, is given by

$$(\Delta f)^2 = \frac{N_{\text{Bin}} - 1}{N_{\text{Bin}}} \sum_{i=1}^{N_{\text{Bin}}} \left[f_{(i)}(\langle\hat{A}\rangle, \langle\hat{B}\rangle, \langle\hat{C}\rangle \cdots) - f_{(\cdot)}(\langle\hat{A}\rangle, \langle\hat{B}\rangle, \langle\hat{C}\rangle \cdots) \right]^2 \; . \tag{98}$$

In case of $f = \langle\hat{A}\rangle$, the results (97) and (98) reduce to the plain sample average and the standard, bias corrected, estimate of the error.

4.2. An explicit example of error estimation

In the following we use one of our examples, the Hubbard model on a square lattice in the M_z HS decoupling (see Sec. 9.1), to show explicitly how to estimate errors. We will show as well that the autocorrelation time is dependent upon the choice of the observable. In fact, different observables within the same run can have different autocorrelation times and, of course, this time scale depends on the parameter choice. Hence, the user has to check autocorrelations of individual observables for each simulation! Typical regions of the phase diagram that require special attention are critical points where length scales diverge.

In order to determine the autocorrelation time, we calculate the correlation function

$$S_{\hat{O}}(t_{\text{QMC}}) = \sum_{i=0}^{N_{\text{Bin}} - t_{\text{QMC}}} \frac{\left(O_i - \left\langle\hat{O}\right\rangle\right)\left(O_{i+t_{\text{QMC}}} - \left\langle\hat{O}\right\rangle\right)}{\left(O_i - \left\langle\hat{O}\right\rangle\right)\left(O_i - \left\langle\hat{O}\right\rangle\right)} \; , \tag{99}$$

where O_i refers to the Monte Carlo estimate of the observable $\hat{O}$ in the i^{th} bin. This function typically shows an exponential decay and the decay rate defines the autocorrelation time. Figure 4(a) shows the autocorrelation functions $S_{\hat{O}}(t_{\text{QMC}})$ for three spin-spin-correlation functions [Eq. (116)] at momentum $\boldsymbol{k} = (\pi, \pi)$ and at $\tau = 0$:

$\hat{O} = S_{\hat{S}^z}$ for the z spin direction, $\hat{O} = (S_{\hat{S}^x} + S_{\hat{S}^y})/2$ for the xy plane, and $\hat{O} = (S_{\hat{S}^x} + S_{\hat{S}^y} + S_{\hat{S}^z})/3$ for the total spin. The Hubbard model has a $SU(2)$ spin symmetry. However, we chose a HS field which couples to the z-component of the magnetization, M_z, such that each configuration breaks this symmetry. Of course, after Monte Carlo averaging one expects restoration of the symmetry. The model, on bipartite lattices, shows spontaneous spin-symmetry breaking at $T = 0$ and in the thermodynamic limit. At finite temperatures, and within the so-called renormalized classical regime, quantum antiferromagnets have a length scale that diverges exponentially with decreasing temperatures [82]. The parameter set chosen for Fig. 4 is non-trivial in the sense that it places the Hubbard model in this renormalized classical regime where the correlation length is substantial. Figure 4 clearly shows a very short autocorrelation time for the xy-plane whereas we detect a considerably longer autocorrelation time for the z-direction. This is a direct consequence of the *long* magnetic length scale and the chosen decoupling. The physical reason for the long autocorrelation time corresponds to the restoration of the $SU(2)$ spin symmetry. This insight can be used to define an improved, $SU(2)$ symmetric estimator for the spin-spin correlation function, namely $(S_{\hat{S}^x} + S_{\hat{S}^y} + S_{\hat{S}^z})/3$. Thereby, global spin rotations are no longer an issue and this improved

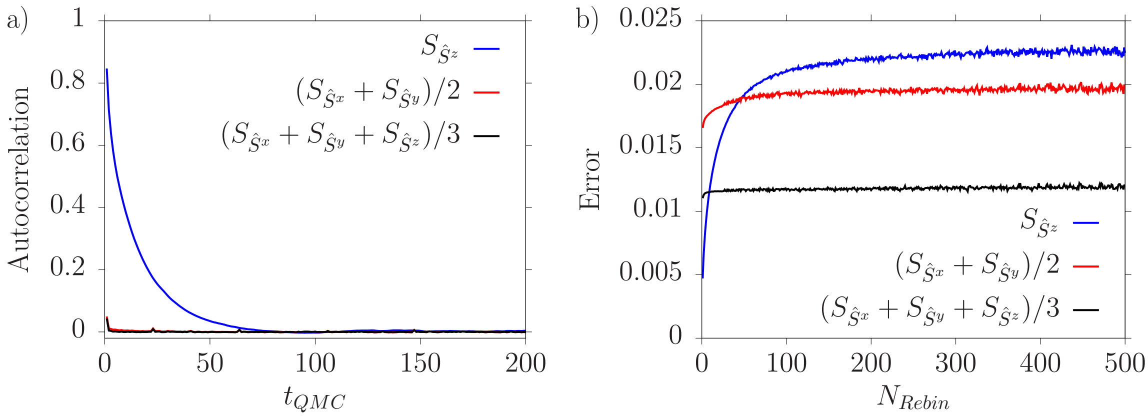

Figure 4: The autocorrelation function $S_{\hat{O}}(t_{\mathrm{QMC}})$ (a) and the scaling of the error with effective bin size (b) of three equal-time, spin-spin correlation functions $\hat{O}$ of the Hubbard model in the M_z decoupling (see Sec. 9.1). Simulations were done on a 6×6 square lattice, with $U/t = 4$ and $\beta t = 6$. The original bin contained only one sweep and we calculated around one million bins on a single core. The different autocorrelation times for the xy-plane compared to the z-direction can be detected from the decay rate of the autocorrelation function (a) and from the point where saturation of the error sets in (b), which defines the required effective bin size for independent measurements. The improved estimator $(S_{\hat{S}^x} + S_{\hat{S}^y} + S_{\hat{S}^z})/3$ appears to have the smallest autocorrelation time, as argued in the text.

estimator shows the shortest autocorrelation time, as can be clearly seen in Fig. 4(b). Other ways to tackle large autocorrelations are global updates and parallel tempering.

A simple method to obtain estimates of the mean and its standard error from the time series of Monte Carlo samples is provided by the aforementioned facility of rebinning. Also known in the literature as rebatching, it consists in aggregating a fixed number `N_rebin` of adjacent original bins into a new effective bin. In addition to measuring the decay rate of the autocorrelation function (Eq. (99)), a measure for the autocorrelation time can be also obtained by the rebinning method. For a comparison to other methods of estimating the autocorrelation time we refer the reader to the literature [83, 78, 79]. A reliable error analysis requires independent bins, otherwise the error is typically underestimated. This behavior is observed in Fig. 4 (b), where the effective bin size is systematically increased by rebinning. If the effective bin size is smaller than the autocorrelation time the error will be underestimated. When the effective bin size becomes larger than the autocorrelation time, converging behavior sets in and the error estimate becomes reliable.

For the analysis of the Monte Carlo data (see Sec. 6.2), the user can provide a finite value for `N_auto` to trigger the computation of autocorrelation functions $S_{\hat{O}}(t_{\mathrm{QMC}})$ in the range $t_{\mathrm{QMC}} = [0, \texttt{N\_auto}]$. Since these computations are quite time consuming and require many Monte Carlo bins, the default value is `N_auto=0`. For Fig. 4, we set `N_auto` $= 500$ and used a total of approximately one million bins.

4.3. Pseudo code description

Basic structure of the auxiliary-field QMC implementation (`Prog/main.F90`):

Set the Hamiltonian and the lattice:
Call ham_set
Read in an auxiliary-field configuration or generate it randomly:
Call field%in

This loop fills the storage needed for the first actual Monte Carlo sweep:
Do `ntau` from `ltrot` to `1`
    Compute propagation matrices and store them at the stabilization points:
    Call wrapul
Enddo

Loop over bins:
Do `nbc` from `1` to `nbin`
Loop over sweeps. Each sweep updates twice (upward and downward in time) the whole space-time lattice of auxiliary fields. The sweep defines the unit of Monte Carlo time:
Do `nsw` from `1` to `nsweep`
Upward sweep:
Do `ntau` from `1` to `ltrot`
Propagate the Green function from time `ntau−1` to `ntau`, and compute a new estimate (using sequential update scheme) of the Green function at `ntau`:
Call wrapgrup

Stabilization:
If `ntau` equals stabilization point in imaginary time **then**
Compute propagation matrix from previous stabilization point to `ntau`:
Call wrapur
Read from storage: propagation from `ltrot` to `ntau`
Write to storage: the just-computed propagation
Recalculate the Green function at time `ntau` in a stable way:
Call cgr
Check the precision between propagated and recalculated Green functions:
Call control_precisionG
Endif

Measure the equal-time observables:
If `ntau` is in the intervall `[LOBS_ST, LOBS_EN]` **then**
Call obser
Endif
Enddo

Downward sweep:
Do `ntau` from `ltrot` to `1`
Repeat the above steps (update, propagation, stabilization, equal-time measurements) for the downward direction in imaginary time
Enddo

Measure the time-displaced observables:
Call tau_m
Enddo (loop over sweeps)

Calculate measurement averages for current bin and write them to disk:
Call pr_obs
Write auxiliary-field configuration to disk:
Call field%out
Enddo (loop over bins)

5. Data Structures and Input/Output

To specify a general model, we define a set of data types. The `Operator` (see Sec. 5.1) type is used to specify the interaction as well as the hopping. The handling of the fields is taken care of by the `Fields` type. To define a Bravais lattice as well as a unit cell we introduce the `Lattice` and `Unit_cell` types. General scalar, equal time and time displaced correlation functions are handled by the `Observable` type. For the projective code, we provide a `WaveFunction` type to define the left and right trial wave functions. The Hamiltonian is then specified in the `Hamiltonian` module that allocates the aforementioned types.

5.1. The `Operator` type

The fundamental data structure in the code is the `Operator`. It is implemented as a Fortran derived data type designed to efficiently define the Hamiltonian (2).

Let the matrix $\boldsymbol{X}$ of dimension $N_{\text{dim}} \times N_{\text{dim}}$ stand for any of the typically sparse, Hermitian matrices $\boldsymbol{T}^{(ks)}$, $\boldsymbol{V}^{(ks)}$ and $\boldsymbol{I}^{(ks)}$ that define the Hamiltonian. Furthermore, let $\{z_1, \cdots, z_N\}$ denote a subset of N indices for which

$$X_{x,y} \begin{cases} \neq 0 & \text{if } x, y \in \{z_1, \cdots z_N\} \\ = 0 & \quad\text{otherwise.} \end{cases} \tag{100}$$

Usually, we have $N \ll N_{\text{dim}}$. We define the $N \times N_{\text{dim}}$ matrices $\boldsymbol{P}$ as

$$P_{i,x} = \delta_{z_i,x} \ , \tag{101}$$

where $i \in [1, \cdots, N]$ and $x \in [1, \cdots, N_{\text{dim}}]$. The matrix $\boldsymbol{P}$ selects the non-vanishing entries of $\boldsymbol{X}$, which are contained in the rank-N matrix $\boldsymbol{O}$ defined by:

$$\boldsymbol{X} = \boldsymbol{P}^T \boldsymbol{O} \boldsymbol{P} \ , \tag{102}$$

and

$$X_{x,y} = \sum_{i,j}^{N} P_{i,x} O_{i,j} P_{j,y} = \sum_{i,j}^{N} \delta_{z_i,x} O_{ij} \delta_{z_j,y} \ . \tag{103}$$

Since the $\boldsymbol{P}$ matrices have only one non-vanishing entry per column, they can conveniently be stored as a vector $\boldsymbol{P}$, with entries

$$P_i = z_i. \tag{104}$$

There are many useful identities which emerge from this structure. For example:

$$e^{\boldsymbol{X}} = e^{\boldsymbol{P}^T \boldsymbol{O} \boldsymbol{P}} = \sum_{n=0}^{\infty} \frac{\left(\boldsymbol{P}^T \boldsymbol{O} \boldsymbol{P}\right)^n}{n!} = \mathbb{1} + \boldsymbol{P}^T \left(e^{\boldsymbol{O}} - \mathbb{1}\right) \boldsymbol{P} \ , \tag{105}$$

since

$$\boldsymbol{P} \boldsymbol{P}^T = \mathbb{1}_{N \times N}. \tag{106}$$

In the code, we define a structure called `Operator` to capture the above. This type `Operator` bundles several components, listed in Table 2, that are needed to define and use an operator matrix in the program.

5.2. Handling of the fields: the `Fields` type

The partition function (see Sec. 2.1) consists of terms which, in general, can be written as $\gamma e^{g\phi \boldsymbol{X}}$, where $\boldsymbol{X}$ denotes an arbitrary operator, g is a constant, and γ and ϕ are fields. The ALF includes three different types of fields.

`t=1` This type is for an Ising field such that $\gamma = 1$ and $\phi = \pm 1$

`t=2` This type is for the generic HS transformation of Eq. 10 such that $\gamma \equiv \gamma(l)$ and $\phi = \eta(l)$ with $l = \pm 1, \pm 2$ (see Eq. 11.

`t=3` This type is for continuous fields such that $\gamma = 1$ and $\phi \in \mathbb{R}$.

For such auxiliary fields a dedicated type `Fields` is defined, whose components, listed in Table 5.2, include the variables `Field%f` and `Field%t`, which store the field values and types, respectively, and functions such as `Field%flip`, which flips the field values randomly.

Variable	Type	Description
Op_X%N	Integer	Effective dimension N
Op_X%O	Complex	Matrix $\boldsymbol{O}$ of dimension $N \times N$
Op_X%P	Integer	Matrix $\boldsymbol{P}$ encoded as a vector of dimension N
Op_X%g	Complex	Coupling strength g
Op_X%alpha	Complex	Constant α
Op_X%type	Integer	Sets the type of HS transformation (1: Ising; 2: discrete HS for perfect-square term; 3: continuous real field.)
Op_X%diag	Logical	True if $\boldsymbol{O}$ is diagonal
Op_X%U	Complex	Matrix containing the eigenvectors of $\boldsymbol{O}$
Op_X%E	Real	Eigenvalues of $\boldsymbol{O}$
Op_X%N_non_zero	Integer	Number of non-vanishing eigenvalues of $\boldsymbol{O}$
Op_X%M_exp	Complex	Stores M_exp$(:,:,s) = e^{g\phi(s,\texttt{type})\boldsymbol{O}(:,:)}$
Op_X%E_exp	Complex	Stores E_exp$(:,s) = e^{g\phi(s,\texttt{type})\boldsymbol{E}(:)}$

Table 2: Member variables of the Operator type. In the left column, the letter X is a placeholder for the letters T and V, indicating hopping and interaction operators, respectively. The highlighted variables must be specified by the user. M_exp and E_exp are allocated only if type $= 1, 2$.

5.3. The Lattice and Unit_cell types

ALF's lattice module can generate one- and two-dimensional Bravais lattices. Both the lattice and the unit cell are defined in the module Lattices_v3_mod.F90 and their components are detailed in Tables 4 and 5. As its name suggest the module Predefined_Latt_mod.F90 also provides predefined lattices as described in Sec. 8.1. The user who wishes to define his/her own lattice also has to specify unit vectors $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$, the size of the lattice, characterized by the vectors $\boldsymbol{L}_1$ and $\boldsymbol{L}_2$ as well as the unit cell characterized be the number of orbitals and their positions. The coordination number of the lattice is equally specified in the Unit_cell data type. The lattice is placed on a torus (periodic boundary conditions):

$$\hat{c}_{\boldsymbol{i}+\boldsymbol{L}_1} = \hat{c}_{\boldsymbol{i}+\boldsymbol{L}_2} = \hat{c}_{\boldsymbol{i}} \,. \tag{107}$$

The function call

```
Call Make_Lattice( L1, L2, a1, a2, Latt )
```

generates the lattice Latt of type Lattice. The reciprocal lattice vectors $\boldsymbol{g}_i$ are defined by:

$$\boldsymbol{a}_i \cdot \boldsymbol{g}_i = 2\pi\delta_{i,j}, \tag{108}$$

and the Brillouin zone BZ corresponds to the Wigner-Seitz cell of the lattice. With $\boldsymbol{k} = \sum_i \alpha_i \boldsymbol{g}_i$, the k-space quantization follows from:

$$\begin{bmatrix} \boldsymbol{L}_1 \cdot \boldsymbol{g}_1 & \boldsymbol{L}_1 \cdot \boldsymbol{g}_2 \\ \boldsymbol{L}_2 \cdot \boldsymbol{g}_1 & \boldsymbol{L}_2 \cdot \boldsymbol{g}_2 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = 2\pi \begin{bmatrix} n \\ m \end{bmatrix} \tag{109}$$

such that

$$\begin{aligned} \boldsymbol{k} = n\boldsymbol{b}_1 + m\boldsymbol{b}_2, \text{ with} \qquad \boldsymbol{b}_1 &= \frac{2\pi}{(\boldsymbol{L}_1\cdot\boldsymbol{g}_1)(\boldsymbol{L}_2\cdot\boldsymbol{g}_2) - (\boldsymbol{L}_1\cdot\boldsymbol{g}_2)(\boldsymbol{L}_2\cdot\boldsymbol{g}_1)} \left[(\boldsymbol{L}_2\cdot\boldsymbol{g}_2)\boldsymbol{g}_1 - (\boldsymbol{L}_2\cdot\boldsymbol{g}_1)\boldsymbol{g}_2\right], \\ \boldsymbol{b}_2 &= \frac{2\pi}{(\boldsymbol{L}_1\cdot\boldsymbol{g}_1)(\boldsymbol{L}_2\cdot\boldsymbol{g}_2) - (\boldsymbol{L}_1\cdot\boldsymbol{g}_2)(\boldsymbol{L}_2\cdot\boldsymbol{g}_1)} \left[(\boldsymbol{L}_1\cdot\boldsymbol{g}_1)\boldsymbol{g}_2 - (\boldsymbol{L}_1\cdot\boldsymbol{g}_2)\boldsymbol{g}_1\right] \end{aligned} \tag{110}$$

The Lattice module also handles the Fourier transformation. For example, the subroutine Fourier_R_to_K carries out the transformation:

$$S(\boldsymbol{k},:,:,:) = \frac{1}{N_{\text{unit-cell}}} \sum_{\boldsymbol{i},\boldsymbol{j}} e^{-i\boldsymbol{k}\cdot(\boldsymbol{i}-\boldsymbol{j})} S(\boldsymbol{i}-\boldsymbol{j},:,:,:) \tag{111}$$

and Fourier_K_to_R the inverse Fourier transform

$$S(\boldsymbol{r},:,:,:) = \frac{1}{N_{\text{unit-cell}}} \sum_{\boldsymbol{k}\in BZ} e^{i\boldsymbol{k}\cdot\boldsymbol{r}} S(\boldsymbol{k},:,:,:). \tag{112}$$

Component		Description
Variable	**Type**	
`Field%t(1:n_op)`	Integer	Sets the HS transformation type (1: Ising; 2: discrete HS for perfect-square term; 3: continuous real field). The index denotes the operator on a given time slice.
`Field%f(1:n_op,1:Ltrot)`	Real	Defines the auxiliary fields. The first index runs through the operator sequence and the second through the time slices. For `t=1` $f = \pm 1$, `t=2` $f = \pm 1, \pm 2$, and for `t=2` $f \in \mathbb{R}$
`del`	Real	Width Δx of box distribution for inititial `t=3` fields, with a default value of `1`.
`amplitude`	Real	Width of a random flip for fields of type `t=3`, defaults to `1`.
Method(arguments)		
`Fields_init(del)`		Initializes internal variables such as $\eta(l)$, $\gamma(l)$ (see Eq. 11) the variable `del` $= \Delta x$ (see above) is optional.
`Field%make(n_op,Ltrot)`		Reserves memory for the field.
`Field%clear()`		Clears field from memory.
`Field%set()`		Sets a random configuration.
`Field%flip(n,nt)`		Flips the field values randomly for field `n` on time slice `nt`. For `t=1` it flips the sign of the Ising spin. For `t=2` it randomly choose one of the three other values of l. For `t=3`, `f = f + amplitude*(ranf() -1/2)`
`Field%phi(n,nt)`		Returns ϕ for the `n`-th operator at the time slice `nt`.
`Field%gamma(n,nt)`		Returns γ for the `n`-th operator at the time slice `nt`.
`Field%i(n,nt)`		Returns `Field%f` rounded to nearest integer (for `t=1` or `2`).
`Field%in(Group_Comm,In_field)`		If the file `confin_np` exists it reads the field configuration from this file. Otherwise if `In_field` is present it sets the fields to `In_field`. If both `confin_np` and `In_field` are not provided it sets a random field by calling `Field%set()`. Here np is the rank number of the process.
`Field%out(Group_Comm)`		Writes out the field configuration.

Table 3: Components of a variable of type `Fields` named `Field`. `del` and `amplitude` are private variables of the fields module. `n_op` and `Ltrot` (integers) are the number of interacting operators per time slice and time slices, respectively, `Group_Comm` (integer) defines an MPI communicator, and the optional `In_field` stores the initial field configuration.

In the above, the unspecified dimensions of the structure factor can refer to imaginary-time and orbital indices.

The position of an orbital i is given by $\boldsymbol{R}_i + \boldsymbol{\delta}_i$. $\boldsymbol{R}_i$ is a point of the Bravais lattice that defines a unit cell, and $\boldsymbol{\delta}_i$ labels the orbital in the unit cell. This information is stored in the array `Unit_cell%Orb_pos` detailed in Table 5.

The total number of orbitals is then given by `Ndim=Lattice%N*Unit_cell%Norb`. To keep track of the orbital and unit cell structure, it is useful to define arrays `List(Ndim,2)` and `Inv_list(Latt%N, Unit_cell%Norb)`. For a superindex $x = (i, n)$ labeling the unit cell, i, and the orbital, n, of a site on the lattice, we have `List(x,1) = i`, `List(x,2)=n` and `Inv_list(i,n) = x` .

5.4. The observable types `Obser_Vec` and `Obser_Latt`

Our definition of the model includes observables [Eq. (25)]. We have defined two observable types: `Obser_vec` for an array of *scalar* observables such as the energy, and `Obser_Latt` for correlation functions that have the lattice symmetry. In the latter case, translation symmetry can be used to provide improved estimators and to reduce the size of the output. We also obtain improved estimators by taking measurements in the imaginary-time interval [`LOBS_ST`, `LOBS_EN`] (see the parameter file in Sec. 5.7.1) thereby exploiting the invariance under translation in imaginary-time. Note that the translation symmetries in space and in time are *broken* for a given configuration C but restored by the Monte Carlo

Variable	Type	Description
Latt%a1_p, Latt%a2_p	Real	Unit vectors $\boldsymbol{a}_1$, $\boldsymbol{a}_2$.
Latt%L1_p, Latt%L2_p	Real	Vectors $\boldsymbol{L}_1$, $\boldsymbol{L}_2$ that define the topology of the lattice. Tilted lattices are thereby possible to implement.
Latt%N	Integer	Number of lattice points, $N_{\text{unit-cell}}$.
Latt%list	Integer	Maps each lattice point $i = 1, \cdots, N_{\text{unit-cell}}$ to a real space vector denoting the position of the unit cell: $\boldsymbol{R}_i = \texttt{list(i,1)}\boldsymbol{a}_1 + \texttt{list(i,2)}\boldsymbol{a}_2 \equiv i_1\boldsymbol{a}_1 + i_2\boldsymbol{a}_2$.
Latt%invlist	Integer	Return lattice point from position: $\texttt{Invlist}(i_1, i_2) = i$.
Latt%nnlist	Integer	Nearest neighbor indices: $j = \texttt{nnlist}(i, n_1, n_2)$, $n_1, n_2 \in [-1, 1]$, $\boldsymbol{R}_j = \boldsymbol{R}_i + n_1\boldsymbol{a}_1 + n_2\boldsymbol{a}_2$.
Latt%imj	Integer	$\boldsymbol{R}_{\text{imj}(i,j)} = \boldsymbol{R}_i - \boldsymbol{R}_j$, with imj, $i, j \in 1, \cdots, N_{\text{unit-cell}}$.
Latt%BZ1_p, Latt%BZ2_p	Real	Reciprocal space vectors $\boldsymbol{g}_i$ (See Eq. 108).
Latt%b1_p, Latt%b1_p	Real	k-quantization (See Eq. 110).
Latt%listk	Integer	Maps each reciprocal lattice point $k = 1, \cdots, N_{\text{unit-cell}}$ to a reciprocal space vector $\boldsymbol{k}_k = \texttt{listk(k,1)}\boldsymbol{b}_1 + \texttt{listk(k,2)}\boldsymbol{b}_2 \equiv k_1\boldsymbol{b}_1 + k_2\boldsymbol{b}_2$.
Latt%invlistk	Integer	$\texttt{Invlistk}(k_1, k_2) = k$.
Latt%b1_perp_p, Latt%b2_perp_p	Real	Orthonormal vectors to $\boldsymbol{b}_i$. For internal use.

Table 4: Components of the `Lattice` type for two-dimensional lattices using as example the default lattice name `Latt`. The highlighted variables must be specified by the user. Other components of the Lattice are generated upon calling: `Call Make_Lattice( L1, L2, a1, a2, Latt )`.

Variable	Type	Description
Norb	Integer	Number of orbitals.
N_coord	Integer	Coordination number.
Orb_pos(1..Norb,2[3])	Real	Positions of the orbitals as measured from the lattice site.

Table 5: Components of an instance `Latt_unit` of the `Unit_cell` type. The highlighted variables have to be specified by the user. Note that for bilayer lattices the second index of the `Orb_pos` array ranges from 1 to 3.

sampling. In general, the user defines size and number of bins in the parameter file, each bin containing a given amount of sweeps. Within a sweep we run sequentially through the HS and Ising fields, from time slice 1 to time slice L_{Trotter} and back. The results of each bin are written to a file and analyzed at the end of the run.

To accomplish the reweighting of observables (see Sec. 2.1.3), for each configuration the measured value of an observable is multiplied by the factors `ZS` and `ZP`:

$$\texttt{ZS} = \operatorname{sign}(C)\,, \tag{113}$$

$$\texttt{ZP} = \frac{e^{-S(C)}}{\operatorname{Re}\left[e^{-S(C)}\right]}\,. \tag{114}$$

They are computed from the Monte Carlo phase of a configuration,

$$\texttt{phase} = \frac{e^{-S(C)}}{\left|e^{-S(C)}\right|}\,, \tag{115}$$

which is provided by the main program. Note that each observable structure also includes the average sign [Eq. (26)].

5.4.1. Scalar observables

Scalar observables are stored in the data type `Obser_vec`, described in Table 6. Consider a variable `Obs` of type `Obser_vec`. At the beginning of each bin, a call to `Obser_Vec_Init` in the module `observables_mod.F90` will set `Obs%N=0`, `Obs%Ave_sign=0` and `Obs%Obs_vec(:)=0`. Each time the main program calls the routine `Obser` in the `Hamiltonian` module, the counter `Obs%N` is incremented by one, the sign (see Eq. 24) is accumulated in the variable `Obs%Ave_sign`, and the desired observables (multiplied by the sign and $\frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]}$, see Sec. 2.1.2) are accumulated in the vector `Obs%Obs_vec`. At the end of the bin, a call to `Print_bin_Vec` in module `observables_mod.F90` will append the result of the bin in the file `File_Vec`_*scal*. Note that this subroutine will automatically append the suffix _*scal* to the the filename `File_Vec`. This suffix is important to allow automatic analysis of the data at the end of the run.

Variable	Type	Description	Contribution of configuration C
`Obs%N`	Int.	Number of measurements	$+1$
`Obs%Ave_sign`	Real	Cumulated average sign [Eq. (26)]	$\mathrm{sign}(C)$
`Obs%Obs_vec(:)`	Comp.	Cumul. vector of observables [Eq. (25)]	$\langle\langle \hat{O}(:)\rangle\rangle_C \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]}$ sign (C)
`Obs%File_Vec`	Char.	Name of output file	

Table 6: Components of a variable of type `Obser_vec` named `Obs`.

5.4.2. Equal-time and time-displaced correlation functions

The data type `Obser_latt` (see Table 7) is useful for dealing with both equal-time and imaginary-time-displaced correlation functions of the form:

$$S_{\alpha,\beta}(\boldsymbol{k},\tau) = \frac{1}{N_{\text{unit-cell}}} \sum_{\boldsymbol{i},\boldsymbol{j}} e^{-i\boldsymbol{k}\cdot(\boldsymbol{i}-\boldsymbol{j})} \left(\langle \hat{O}_{\boldsymbol{i},\alpha}(\tau)\hat{O}_{\boldsymbol{j},\beta}\rangle - \langle \hat{O}_{\boldsymbol{i},\alpha}\rangle\langle \hat{O}_{\boldsymbol{j},\beta}\rangle \right), \tag{116}$$

where α and β are orbital indices and $\boldsymbol{i}$ and $\boldsymbol{j}$ lattice positions. Here, translation symmetry of the Bravais lattice is explicitly taken into account. The correlation function splits in a correlated part $S^{(\mathrm{corr})}_{\alpha,\beta}(\boldsymbol{k},\tau)$

Variable	Type	Description	Contribution of configuration C
`Obs%N`	Int.	Number of measurements	$+1$
`Obs%Ave_sign`	Real	Cumulated sign [Eq. (26)]	$\mathrm{sign}(C)$
`Obs%Obs_latt`$(\boldsymbol{i}-\boldsymbol{j},$ $\tau,\alpha,\beta)$	Compl.	Cumul. correl. funct. [Eq. (25)]	$\langle\langle \hat{O}_{\boldsymbol{i},\alpha}(\tau)\hat{O}_{\boldsymbol{j},\beta}\rangle\rangle_C \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]}\mathrm{sign}(C)$
`Obs%Obs_latt0`(α)	Compl.	Cumul. expect. value [Eq. (25)]	$\langle\langle \hat{O}_{\boldsymbol{i},\alpha}\rangle\rangle_C \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]}$ sign (C)
`Obs%File_Latt`	Char.	Name of output file	
`Obs%Latt`	Lattice pointer	Bravais lattice [Tab. 4]	
`Obs%Latt_unit`	Unit_cell pointer	Unit cell [Tab. 5]	
`Obs%dtau`	Real	Imaginary time step.	
`Obs%Channel`	Char.	Channel for Maximum Entropy.	

Table 7: Components of a variable of type `Obser_latt` named `Obs`. Be aware: When creating the observable with the subroutine `Obser_Latt_make`, `Latt` and `Latt_unit` don't get copied but linked, meaning changing them after making the observable still affects the observable.

and a background part $S^{(\text{back})}_{\alpha,\beta}(\boldsymbol{k})$:

$$S^{(\text{corr})}_{\alpha,\beta}(\boldsymbol{k},\tau) = \frac{1}{N_{\text{unit-cell}}} \sum_{\boldsymbol{i},\boldsymbol{j}} e^{-i\boldsymbol{k}\cdot(\boldsymbol{i}-\boldsymbol{j})} \langle \hat{O}_{\boldsymbol{i},\alpha}(\tau) \hat{O}_{\boldsymbol{j},\beta} \rangle \ , \tag{117}$$

$$\begin{aligned} S^{(\text{back})}_{\alpha,\beta}(\boldsymbol{k}) &= \frac{1}{N_{\text{unit-cell}}} \sum_{\boldsymbol{i},\boldsymbol{j}} e^{-i\boldsymbol{k}\cdot(\boldsymbol{i}-\boldsymbol{j})} \langle \hat{O}_{\boldsymbol{i},\alpha} \rangle \langle \hat{O}_{\boldsymbol{j},\beta} \rangle \\ &= N_{\text{unit-cell}} \, \langle \hat{O}_{\alpha} \rangle \langle \hat{O}_{\beta} \rangle \, \delta(\boldsymbol{k}) \ , \end{aligned} \tag{118}$$

where translation invariance in space and time has been exploited to obtain the last line. The background part depends only on the expectation value $\langle \hat{O}_\alpha \rangle$, for which we use the following estimator

$$\langle \hat{O}_\alpha \rangle \equiv \frac{1}{N_{\text{unit-cell}}} \sum_{\boldsymbol{i}} \langle \hat{O}_{\boldsymbol{i},\alpha} \rangle \ . \tag{119}$$

Consider a variable `Obs` of type `Obser_latt`. At the beginning of each bin a call to `Obser_Latt_Init` in the module `observables_mod.F90` will initialize the elements of `Obs` to zero. Each time the main program calls the `Obser` or `ObserT` routines one accumulates $\langle\langle \hat{O}_{\boldsymbol{i},\alpha}(\tau) \hat{O}_{\boldsymbol{j},\beta} \rangle\rangle_C \ \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]} \mathrm{sign}(C)$ in `Obs%Obs_latt`$(\boldsymbol{i}-\boldsymbol{j},\tau,\alpha,\beta)$ and $\langle\langle \hat{O}_{\boldsymbol{i},\alpha} \rangle\rangle_C \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]}$ sign (C) in `Obs%Obs_latt0`(α). At the end of each bin, a call to `Print_bin_Latt` in the module `observables_mod.F90` will append the result of the bin in the specified file `Obs%File_Latt`. Note that the routine `Print_bin_Latt` carries out the Fourier transformation and prints the results in k-space. We have adopted the following naming conventions. For equal-time observables, defined by having the second dimension of the array `Obs%Obs_latt`$(\boldsymbol{i}-\boldsymbol{j},\tau,\alpha,\beta)$ set to unity, the routine `Print_bin_Latt` attaches the suffix *_eq* to `Obs%File_Latt`. For time-displaced correlation functions we use the suffix *_tau*.

5.5. The `WaveFunction` type

The projective algorithm (Sec. 3) requires a pair of trial wave functions, $|\Psi_{T,L/R}\rangle$, for which there is the dedicated `WaveFunction` type, defined in the module `WaveFunction_mod` as described in Table 8.

Variable	Type	Description
`WF%P(:,:)`	Complex	P is an `Ndim` $\times$ `N_part` matrix, where `N_part` is the number of particles
`WF%Degen`	Real	It stores the energy difference between the last occupied and fist unoccupied single particle state and can be used for checking for degeneracy

Table 8: Components of a variable of type `WaveFunction` named `WF`.

The module `WaveFunction_mod` also includes the routine `WF_overlap(WF_L,WF_R,Z_norm)` for normalizing the right trial wave function `WF_R` by the factor `Z_norm`, such that $\langle \Psi_{T,L} | \Psi_{T,R} \rangle = 1$.

5.6. Specification of the Hamiltonian: the `Hamiltonian` module

The modules `Hamiltonian` in the directory `$ALF/Prog/Hamiltonians` define specific Hamiltonians. This module must contain a set of subroutines that define the lattice, the hopping, the interaction, the observables, the trial wave function, and optionally updating schemes (see Sec. 2.2). In order to implement a user-defined model, only the module `Hamiltonian` has to be set up. Accordingly, this documentation focuses almost entirely on this module and the subprograms it includes. The remaining parts of the code may hence be treated as a black box. The mandatory elements of the Hamiltonian module are defined in Table 9). To simplify the implementation of a new Hamiltonian, ALF comes with a set of predefined structures (Sec. 8) which the user can combine together or use as templates.

In order to specify a Hamiltonian, we have to set the matrix representation of the imaginary-time propagators, $e^{-\Delta\tau \boldsymbol{T}^{(ks)}}$, $e^{\sqrt{-\Delta\tau U_k}\eta_{k\tau}\boldsymbol{V}^{(ks)}}$ and $e^{-\Delta\tau s_{k\tau}\boldsymbol{I}^{(ks)}}$, that appear in the partition function (16). For each pair of indices (k,s), these terms have the general form

$$\text{Matrix Exponential} = e^{g\,\phi(\texttt{type})\,\boldsymbol{X}} \ . \tag{120}$$

Subprogram	Description	Section
`Ham_Set`	Reads in model and lattice parameters from the file `parameters`. Sets the Hamiltonian calling the necessary subprograms: `Ham_Latt`, `Ham_Hop`, `Ham_V` and `Ham_Trial`	5.6, 9
`Ham_Latt`	Sets the `Lattice` and the `Unit_cell` as well as the the arrays `List` and `Inv_list` required for multiorbital problems	5.3, 7.2 8.1
`Ham_hop`	Sets the hopping term $\hat{\mathcal{H}}_T$ (i.e., operator `Op_T`) by calling `Op_make` and `Op_set`.	5.1, 7.3, 8.2
`Ham_V`	Sets the interaction term $\hat{\mathcal{H}}_V$ (i.e., operator `Op_V`) by calling `Op_make` and `Op_set`.	5.1, 7.4, 8.3
`Ham_Trial`	Sets the trial wave function for the projective code $\vert\Psi_{T,L/R}\rangle$ specified by the `Wavefunction` type.	5.5, 7.5, 8.5
`Alloc_obs`	Assigns memory storage to the observables.	5.4 , 7.6.1,
`Obser`	Computes the scalar and equal-time observables.	5.4, 7.6.2, 8.4
`ObserT`	Computes time-displaced correlation functions.	5.4, 7.6.3, 8.4
`S0`	Returns the ratio $e^{S_0(C')}/e^{-S_0(C)}$ for a single spin flip	2.2.1
`Global_move_tau`	Generates a global move on a given time slice τ. This routine is only called if `Global_tau_moves=True` and `N_Global_tau>0`.	2.2.3
`Overide_global_tau_sampling_parameters`	Allows setting `global_tau` parameters at run time.	2.2.3
`Hamiltonian_set_nsigma`	Sets the initial field configuration. This routine is to be modified if one wants to specify the initial configuration. By default the initial configuration is assumed to be random.	
`Global_move`	Handles global moves in time and space	2.2.4
`Delta_S0_global`	Computes $e^{S_0(C')}/e^{-S_0(C)}$ for a global move	2.2.4
`Init_obs`	Initializes the observables to zero.	
`Pr_obs`	Writes the observables to disk by calling `Print_bin` of the `observables_mod.F90` module.	

Table 9: Overview of the subprograms of the module `Hamiltonian`, contained in the Hamiltonian files used to define various Hamiltonians. The highlighted subroutines may have to be modified by the user.

In case of the perfect-square term, we additionally have to set the constant α, see the definition of the operators $\hat{V}^{(k)}$ in Eq. (4). The data structures which hold all the above information are variables of the type `Operator` (see Table 2). For each pair of indices (k, s), we store the following parameters in an `Operator` variable:

- $\boldsymbol{P}$ and $\boldsymbol{O}$ defining the matrix $\boldsymbol{X}$ [see Eq. (102)],
- the constants g, α,
- optionally: the type `type` of the discrete fields ϕ.

The latter parameter can take one of three values: Ising (1), discrete HS (2), and real (3), as detailed in Sec. 5.2. Note that we have dropped the color index σ, since the implementation uses the $SU(N_{\mathrm{col}})$ invariance of the Hamiltonian.

Accordingly, the following data structures fully describe the Hamiltonian (2):

- For the hopping Hamiltonian (3), we have to set the exponentiated hopping matrices $e^{-\Delta\tau \boldsymbol{T}^{(ks)}}$: In this case $\boldsymbol{X}^{(ks)} = \boldsymbol{T}^{(ks)}$, and a single variable `Op_T` describes the operator matrix

$$\left(\sum_{x,y}^{N_{\mathrm{dim}}} \hat{c}^{\dagger}_x T^{(ks)}_{xy} \hat{c}_y \right) , \tag{121}$$

where $k = [1, M_T]$ and $s = [1, N_{\mathrm{fl}}]$. In the notation of the general expression (120), we set $g = -\Delta\tau$ (and $\alpha = 0$). In case of the hopping matrix, the type variable `Op_T%type` is neglected by the code. All in all, the corresponding array of structure variables is `Op_T(M`$_T$`,N`$_{fl}$`)`.

- For the interaction Hamiltonian (4), which is of perfect-square type, we have to set the exponentiated matrices $e^{\sqrt{-\Delta\tau U_k}\eta_{k\tau}\boldsymbol{V}^{(ks)}}$:
  In this case, $\boldsymbol{X} = \boldsymbol{V}^{(ks)}$ and a single variable `Op_V` describes the operator matrix:

$$\left[\left(\sum_{x,y}^{N_{\mathrm{dim}}} \hat{c}_x^\dagger V_{x,y}^{(ks)} \hat{c}_y \right) + \alpha_{ks} \right] , \tag{122}$$

  where $k = [1, M_V]$ and $s = [1, N_{\mathrm{fl}}]$, $g = \sqrt{-\Delta\tau U_k}$ and $\alpha = \alpha_{ks}$. The discrete HS decomposition which is used for the perfect-square interaction, is selected by setting the type variable to `Op_V%type` $= 2$. All in all, the required structure variables `Op_V` are defined using the array `Op_V(M`$_V$`,N`$_{\mathrm{fl}}$`)`.

- For the Ising interaction Hamiltonian (5), we have to set the exponentiated matrices $e^{-\Delta\tau s_{k\tau}\boldsymbol{I}^{(ks)}}$:
  In this case, $\boldsymbol{X} = \boldsymbol{I}^{(k,s)}$ and a single variable `Op_V` then describes the operator matrix:

$$\left(\sum_{x,y}^{N_{\mathrm{dim}}} \hat{c}_x^\dagger I_{xy}^{(ks)} \hat{c}_y \right) , \tag{123}$$

  where $k = [1, M_I]$ and $s = [1, N_{\mathrm{fl}}]$ and $g = -\Delta\tau$ (and $\alpha = 0$). The Ising interaction is specified by setting the type variable `Op_V%type=1`. All in all, the required structure variables are contained in the array `Op_V(M`$_I$`,N`$_{\mathrm{fl}}$`)`.

- In case of a full interaction [perfect-square term (4) and Ising term (5)], we define the corresponding doubled array `Op_V(M`$_V$`+M`$_I$`,N`$_{\mathrm{fl}}$`)` and set the variables separately for both ranges of the array according to the above.

5.7. File structure

Directory	Description
`Prog/`	Main program and subroutines
`Libraries/`	Collection of mathematical routines
`Analysis/`	Routines for error analysis
`Scripts_and_Parameters_files/`	Helper scripts and the `Start/` directory, which contains the files required to start a run
`Documentation/`	This documentation
`testsuite/`	A suite for automatic testing various parts of the code

Table 10: Overview of the directories included in the ALF package.

The code package, summarized in Table 10, consists of the program directories `Prog/`, `Libraries/`, and `Analysis/`, as well as the directory `Scripts_and_Parameters_files/`, which contains supporting scripts and, in its subdirectory `Start`, the input files necessary for a run, described in the Sec. 5.7.1. The routines available in the directory `Analysis/` are described in Sec. 6.2, and the testsuite in Sec. 6.1.

Below we describe the structure of the input and output files of the QMC. Notice that the input/output files for the Analysis routines are described in Sec. 6.2.

5.7.1. Input files

The input files are listed in Table 11. The parameter file `Start/parameters` has the following form – using as an example the Hubbard model on a square lattice (see Sec. 9.1 for the general SU(N) Hubbard and Sec. 7 for a detailed walk-through on its plain vanilla version):

```
!===============================================================================
!  Input variables for a general ALF run
!-------------------------------------------------------------------------------

&VAR_lattice                 !! Parameters defining the specific lattice and base model
L1           = 6              ! Length in direction a_1
L2           = 6              ! Length in direction a_2
Lattice_type = "Square"       ! Sets a_1 = (1,0), a_2=(0,1), Norb=1, N_coord=2
Model        = "Hubbard"      ! Sets the Hubbard model, to be specified in &VAR_Hubbard
/

&VAR_Model_Generic           !! Common model parameters
Checkerboard = .T.            ! Whether checkerboard decomposition is used
Symm         = .T.            ! Whether symmetrization takes place
N_SUN        = 2              ! Number of colors
N_FL         = 1              ! Number of flavors
Phi_X        = 0.d0           ! Twist along the L_1 direction, in units of the flux quanta
Phi_Y        = 0.d0           ! Twist along the L_2 direction, in units of the flux quanta
Bulk         = .T.            ! Twist as a vector potential (.T.), or at the boundary (.F.)
N_Phi        = 0              ! Total number of flux quanta traversing the lattice
Dtau         = 0.1d0          ! Thereby Ltrot=Beta/dtau
Beta         = 5.d0           ! Inverse temperature
Projector    = .F.            ! Whether the projective algorithm is used
Theta        = 10.d0          ! Projection parameter
/

&VAR_QMC                     !! Variables for the QMC run
Nwrap                 = 10    ! Stabilization. Green functions will be computed from
                              ! scratch after each time interval Nwrap*Dtau
NSweep                = 20    ! Number of sweeps
NBin                  = 5     ! Number of bins
Ltau                  = 1     ! 1 to calculate time-displaced Green functions; 0 otherwise
LOBS_ST               = 0     ! Start measurements at time slice LOBS_ST
LOBS_EN               = 0     ! End measurements at time slice LOBS_EN
CPU_MAX               = 0.0   ! Code stops after CPU_MAX hours, if 0 or not
                              ! specified, the code stops after Nbin bins
Propose_S0            = .F.   ! Proposes single spin flip moves with probability exp(-S0)
Global_moves          = .F.   ! Allows for global moves in space and time
N_Global              = 1     ! Number of global moves per sweep
Global_tau_moves      = .F.   ! Allows for global moves on a single time slice.
N_Global_tau          = 1     ! Number of global moves that will be carried out on a
                              ! single time slice
Nt_sequential_start = 0       ! One can combine sequential and global moves on a time slice
Nt_sequential_end   = -1      ! The program then carries out sequential local moves in the
                              ! range [Nt_sequential_start, Nt_sequential_end] followed by
                              ! N_Global_tau global moves
/

&VAR_errors                  !! Variables for analysis programs
n_skip   = 1                  ! Number of bins that to be skipped.
N_rebin = 1                   ! Rebinning
N_Cov    = 0                  ! If set to 1 covariance computed for non-equal-time
                              ! correlation functions
N_auto    = 0                 ! If > 0  triggers  calculation of autocorrelation
N_Back    = 1                 ! If set to 1, substract background in correlation functions
/

&VAR_TEMP                    !! Variables for parallel tempering
N_exchange_steps      = 6     ! Number of exchange moves [see Eq. (37)]
N_Tempering_frequency = 10    ! The frequency in units of sweeps at which the
                              ! exchange moves are carried out
mpi_per_parameter_set = 2     ! Number of mpi-processes per parameter set
Tempering_calc_det    = .T.   ! Specifies whether the fermion weight has to be taken
```

```
                             ! into account while tempering. The default is .true.,
                             ! and it can be set to .F. if the parameters that
                             ! get varied only enter the Ising action S_0
/

&VAR_Max_Stoch              !! Variables for Stochastic Maximum entropy
Ngamma      = 400            ! Number of Dirac delta-functions for parametrization
Om_st       = -10.d0         ! Frequency range lower bound
Om_en       = 10.d0          ! Frequency range upper bound
NDis        = 2000           ! Number of boxes for histogram
Nbins       = 250            ! Number of bins for Monte Carlo
Nsweeps     = 70             ! Number of sweeps per bin
NWarm       = 20             ! The Nwarm first bins will be ommitted
N_alpha     = 14             ! Number of temperatures
alpha_st    = 1.d0           ! Smallest inverse temperature increment for inverse
R           = 1.2d0          ! temperature (see above)
Checkpoint = .F.             ! Whether to produce dump files, allowing the simulation
                             ! to be resumed later on
Tolerance  = 0.1d0           ! Data points for which the relative error exceeds the
                             ! tolerance threshold will be omitted.
/

&VAR_Hubbard                !! Variables for the specific model
Mz         = .T.             ! When true, sets the M_z-Hubbard model: Nf=2, N_sun=1, HS field
                             ! couples to the z-component of magnetization; otherwise, HS field
                             ! couples to the density

ham_T      = 1.d0            ! Hopping parameter
ham_chem   = 0.d0            ! Chemical potential
ham_U      = 4.d0            ! Hubbard interaction
ham_T2     = 1.d0            ! For bilayer systems
ham_U2     = 4.d0            ! For bilayer systems
ham_Tperp = 1.d0             ! For bilayer systems
/
```

File	Description
`parameters`	Sets the parameters for lattice, model, QMC process, and the error analysis.
`seeds`	List of integer numbers to initialize the random number generator and to start a simulation from scratch.

Table 11: Overview of the input files required for a simulation, which can be found in the subdirectory `Scripts_and_Parameters_files/Start/`.

The program allows for a number of different updating schemes. If no other variables are specified in the `VAR_QMC` name space, then the program will run in its default mode, namely the sequential single spin-flip mode. In particular, note that if `Nt_sequential_start` and `Nt_sequential_end` are not specified and that the variable `Global_tau_moves` is set to true, then the program will carry out only global moves, by setting `Nt_sequential_start=1` and `Nt_sequential_end=0`.

If the program is not compiled with the parallel tempering flag, then the `VAR_TEMP` name space can be omitted from the parameter file.

5.7.2. Output files – observables

The standard output files are listed in Table 12. The output of the measured data is organized in bins. One bin corresponds to the arithmetic average over a fixed number of individual measurements which depends on the chosen measurement interval `[LOBS_ST,LOBS_EN]` on the imaginary-time axis and on the number `NSweep` of Monte Carlo sweeps. If the user runs an MPI parallelized version of the code, the average also extends over the number of MPI threads. The formatting of a single bin's output depends on the observable type, `Obs_vec` or `Obs_Latt`:

File	Description
`info`	After completion of the simulation, this file documents the parameters of the model, as well as the QMC run and simulation metrics (precision, acceptance rate, wallclock time)
`X_scal`	Results of equal-time measurements of scalar observables The placeholder `X` stands for the observables `Kin`, `Pot`, `Part`, and `Ener`
`Y_eq`, `Y_tau`	Results of equal-time and time-displaced measurements of correlation functions. The placeholder `Y` stands for `Green`, `SpinZ`, `SpinXY`, and `Den`
`confout_<threadnumber>`	Output files (one per MPI instance) for the HS and Ising configuration

Table 12: Overview of the standard output files. See Sec. 5.4 for the definitions of observables and correlation functions.

- Observables of type `Obs_vec`: For each additional bin, a single new line is added to the output file. In case of an observable with `N_size` components, the formatting is

```
N_size + 1    <measured value, 1> ... <measured value, N_size>    <measured sign>
```

  The counter variable `N_size+1` refers to the number of measurements per line, including the phase measurement. This format is required by the error analysis routine (see Sec. 6.2). Scalar observables like kinetic energy, potential energy, total energy and particle number are treated as a vector of size `N_size=1`.

- Observables of type `Obs_Latt`: For each additional bin, a new data block is added to the output file. The block consists of the expectation values [Eq. (119)] contributing to the background part [Eq. (118)] of the correlation function, and the correlated part [Eq. (117)] of the correlation function. For imaginary-time displaced correlation functions, the formatting of the block is given by:

```
<measured sign>  <N_orbital>  <N_unit_cell>  <N_time_slices>  <dtau>  <Channel>
do alpha = 1, N_orbital
    ⟨Ô_α⟩
enddo
do i = 1, N_unit_cell
   <reciprocal lattice vector k(i)>
   do tau = 1, N_time_slices
      do alpha = 1, N_orbital
         do beta = 1, N_orbital
            ⟨S^(corr)_α,β(k(i),τ)⟩
         enddo
      enddo
   enddo
enddo
```

  The same block structure is used for equal-time correlation functions, except for the entries `<N_time_slices>`, `<dtau>` and `<Channel>`, which are then omitted. Using this structure for the bins as input, the full correlation function $S_{\alpha,\beta}(\boldsymbol{k},\tau)$ [Eq. (116)] is then calculated by calling the error analysis routine (see Sec. 6.2).

6. Using the Code

In this section we describe the steps for compiling and running the code from the shell, and describe how to search for optimal parameter values as well as how to perform the error analysis of the data.

A Python interface, **pyALF**, is also available and can be found, together with a number of Jupyter notebooks exploring the interface's capabilities, at `https://git.physik.uni-wuerzburg.de/ALF/pyALF`. This interface facilitates setting up simple runs and is ideal for setting benchmarks and getting acquainted with ALF. Some of pyALF's notebooks form the core of the introductory part of the ALF Tutorial, where pyALF's usage is described in more detail.

6.1. Compiling and running

The necessary environment variables and the directives for compiling the code are set by the script `configure.sh`:

```
source configure.sh [MACHINE] [MODE] [STAB]
```

If run with no arguments, it lists the available options and sets a generic, serial GNU compiler with minimal flags `-cpp -O3 -ffree-line-length-none -ffast-math`. The predefined machine configurations and parallelization modes available, as well as the options for stabilization schemes for the matrix multiplications (see Sec. 2.4) are shown Table 13. The stabilization scheme choice, in particular, is critical for performance and is discussed further in Sec. 6.3.

Argument	Selected feature
`MACHINE`	
`Intel`	Intel compiler for a generic machine[4].
`GNU`	GNU compiler for a generic machine (*default*).
`PGI`	PGI compiler for a generic machine.
`MAC`	GNU compiler for a generic MAC computer.
`SuperMUC-NG`	Intel compiler and loading the necessary modules for SuperMUC-NG[5].
`JUWELS`	Intel compiler and loading the necessary modules for JUWELS[6].
`Devel|Development`	GNU compiler, and flags appropriate for debugging.
`MODE`	
`noMPI|serial`	No parallelization.
`MPI`	MPI parallelization (*default* – if a machine is selected).
`Tempering`	Parallel tempering (Sec. 2.2.5) and the required MPI as well.
`STAB`	
`STAB1`	Simplest stabilization, with UDV (QR-, not SVD-based) decompositions.
`STAB2`	QR-based UDV decompositions with additional normalizations.
`STAB3`	Newest stabilization, additionally separates large and small scales (*default*).
`LOG`	Log storage for internal scales, increases accessible ranges.

Table 13: Available arguments for the script `configure.sh`, called before compilation of the package: predefined machines, parallelization modes, and stabilization schemes (see also Sec. 6.3).

In order to compile the libraries, the analysis routines and the QMC program at once, just execute the single command:

```
make
```

Related directories, object files and executables can be removed by executing the command `make clean`. The accompanying `Makefile` also provides rules for compiling and cleaning up the library, the analysis routines and the QMC program separately.

A suite of tests for individual parts of the code (subroutines, functions, operations, etc.) is available at the directory `testsuite`. The tests can be run by executing the following sequence of commands (the script `configure.sh` sets environment variables as described above.):

```
source configure.sh Devel serial
gfortran -v
make lib
make ana
```

[4] A known issue with the alternative Intel Fortran compiler `ifort` is the handling of automatic, temporary arrays which `ifort` allocates on the stack. For large system sizes and/or low temperatures this may lead to a runtime error. One solution is to demand allocation of arrays above a certain size on the heap instead of the stack. This is accomplished by the `ifort` compiler flag `-heap-arrays [n]` where `[n]` is the minimal size (in kilobytes, for example `n=1024`) of arrays that are allocated on the heap.

[5] Supercomputer at the Leibniz Supercomputing Centre.

[6] Supercomputer at the Jülich Supercomputing Centre.

```
make Examples
cd testsuite
cmake -E make_directory tests
cd tests
cmake -G "Unix Makefiles" -DCMAKE_Fortran_FLAGS_RELEASE=${F90OPTFLAGS} \
-DCMAKE_BUILD_TYPE=RELEASE ..
cmake --build . --target all --config Release
ctest -VV -O log.txt
```

which will output test results and total success rate.

Starting a simulation

In order to start a simulation from scratch, the following files have to be present: `parameters` and `seeds` (see Sec. 5.7.1). To run serially the simulation for a given model, for instance one of the plain vanilla Hubbard model included in `Hamiltonian_Hubbard_Plain_Vanilla_mod.F90`, described in Sec. 9.1, issue the command

```
./Prog/Hubbard_Plain_Vanilla.out
```

In order to run a different model, the corresponding executable should be used and, for running with parallelization, the appropriate MPI execution command should be called. For instance, a GNU-compiled Kondo model (Sec. 9.3) can be run in parallel by issuing

```
orterun -np <number of processes> ./Prog/Kondo_Honey.out
```

To restart the code using the configuration from a previous simulation as a starting point, first run the script `out_to_in.sh`, which copies outputted field configurations into input files, before calling the ALF executable.

6.2. Error analysis

The ALF package includes the analysis program `ana.out` for performing simple error analysis and correlation function calculations on the three observable types. To perform an error analysis based on the Jackknife resampling method [81] (Sec. 4.1) of the Monte Carlo bins for a list of observables run

```
/path/to/ALF/Analysis/ana.out <list of files>
```

or run

```
/path/to/ALF/Analysis/ana.out *
```

for all observables.

In case the parameter `N_auto` is set to a finite value the program will also trigger the computation of autocorrelation functions (Sec. 4.2).

The program `ana.out` is based on the included module `ana_mod`, which provides subroutines for reading an analyzing ALF Monte Carlo bins, that can be used to implement more specialized analysis. The three high-level analysis routines employed by `ana_mod` are listed in Table 14. The files taken as input, as well as the output files are listed in Table 15.

The error analysis is based on the central limit theorem, which requires bins to be statistically independent, and also the existence of a well-defined variance for the observable under consideration (see Sec. 4). The former will be the case if bins are longer than the autocorrelation time. The latter has to be checked by the user. In the parameter file described in Sec. 5.7.1, the user can specify how many initial bins should be omitted (variable `n_skip`). This number should be comparable to the autocorrelation time. The rebinning variable `N_rebin` will merge `N_rebin` bins into a single new bin. If the autocorrelation time is smaller than the effective bin size, the error should become independent of the bin size and thereby of the variable `N_rebin`. The analysis output files listed in Table 15 and are formatted in the following way:

- For the scalar quantities `X`, the output files `X_scalJ` have the following formatting:

```
Effective number of bins, and bins:      <N_bin - N_skip>/<N_rebin>       <N_bin>
OBS :    1       <mean(X)>       <error(X)>
OBS :    2       <mean(sign)>    <error(sign)>
```

Program	Description
`cov_vec(name)`	The bin file `name`, which should have suffix `_scal`, is read in, and the corresponding file with suffix `_scalJ` is produced. It contains the result of the Jackknife rebinning analysis (see Sec. 4).
`cov_eq(name)`	The bin file `name`, which should have suffix `_eq`, is read in, and the corresponding files with suffix `_eqJR` and `_eqJK` are produced. They correspond to correlation functions in real and Fourier space, respectively.
`cov_tau(name)`	The bin file `name`, which should have suffix `_tau`, is read in, and the directories `X_kx_ky` are produced for all `kx` and `ky` greater or equal to zero. Here `X` is a place holder from `Green`, `SpinXY`, etc., as specified in `Alloc_obs(Ltau)` (See section 7.6.1). Each directory contains a file `g_dat` containing the time-displaced correlation function traced over the orbitals. It also contains the covariance matrix if `N_cov` is set to unity in the parameter file (see Sec. 5.7.1). Also, a directory `X_R0` for the local time displaced correlation function is generated. For particle-hole, imaginary-time correlation functions (`Channel="PH"`) such as Spin and Charge, we use the fact that these correlation functions are symmetric around $\tau = \beta/2$ so that we can define an improved estimator by averaging over τ and $\beta - \tau$.

Table 14: Overview of analysis subroutines called within the program `ana.out`.

File	Description
Input	
`parameters`	Includes error analysis variables `N_skip`, `N_rebin`, and `N_Cov` (see Sec. 5.7.1)
`X_scal`, `Y_eq`, `Y_tau`	Monte Carlo bins (see Table 12)
Output	
`X_scalJ`	Jackknife mean and error of `X`, where `X` stands for `Kin, Pot, Part`, or `Ener`
`Y_eqJR` and `Y_eqJK`	Jackknife mean and error of `Y`, which stands for `Green, SpinZ, SpinXY`, or `Den`. The suffixes `R` and `K` refer to real and reciprocal space, respectively
`Y_R0/g_R0`	Time-resolved and spatially local Jackknife mean and error of `Y`, where `Y` stands for `Green, SpinZ, SpinXY`, and `Den`
`Y_kx_ky/g_kx_ky`	Time resolved and $\boldsymbol{k}$-dependent Jackknife mean and error of `Y`, where `Y` stands for `Green, SpinZ, SpinXY`, and `Den`

Table 15: Standard input and output files of the error analysis program `ana.out`.

- For the equal-time correlation functions `Y`, the formatting of the output files `Y_eqJR` and `Y_eqJK` follows the structure:

```
do i = 1, N_unit_cell
   <k_x(i)>   <k_y(i)>
   do alpha = 1, N_orbital
      do beta  = 1, N_orbital
         alpha   beta   Re<mean(Y)>   Re<error(Y)>   Im<mean(Y)>   Im<error(Y)>
      enddo
   enddo
enddo
```

  where `Re` and `Im` refer to the real and imaginary part, respectively.

- The imaginary-time displaced correlation functions `Y` are written to the output files `g_R0` inside folders `Y_R0`, when measured locally in space; and to the output files `g_kx_ky` inside folders `Y_kx_ky` when they are measured $\boldsymbol{k}$-resolved (where $\boldsymbol{k} = (\mathtt{kx}, \mathtt{ky})$). The first line of the file prints the number of time slices, the number of bins and the inverse temperature. Both output files have the following formatting:

```
do i = 0, Ltau
   tau(i)   <mean( Tr[Y] )>   <error( Tr[Y])>
enddo
```

where `Tr` corresponds to the trace over the orbital degrees of freedom. For particle-hole quantities at finite temperature, τ runs from 0 to $\beta/2$. In all other cases it runs from 0 to β.

6.3. Parameter optimization

The finite-temperature, auxiliary-field QMC algorithm is known to be numerically unstable, as discussed in Sec. 2.4. The numerical instabilities arise from the imaginary-time propagation, which invariably leads to exponentially small and exponentially large scales. As shown in Ref. [6], scales can be omitted in the ground state algorithm – thus rendering it very stable – but have to be taken into account in the finite-temperature code.

Numerical stabilization of the code is a delicate procedure that has been pioneered in Ref. [2] for the finite-temperature algorithm and in Refs. [3, 4] for the zero-temperature, projective algorithm. It is important to be aware of the fragility of the numerical stabilization and that there is no guarantee that it will work for a given model. It is therefore crucial to always check the file `info`, which, apart from runtime data, contains important information concerning the stability of the code, in particular `Precision Green`. If the numerical stabilization fails, one possible measure is to reduce the value of the parameter `Nwrap` in the parameter file, which will however also impact performance – see Table. 16 for further optimization tips for the Monte Carlo algorithm (Sec. 4). Typical values for the numerical precision ALF can achieve can be found in Sec. 9.1.

Element	Suggestion
`Precision Green`, `Precision Phase`	Should be found to be *small*, of order $< 10^{-8}$ (see Sec. 2.4)
`theta`	Should be *large* enough to guarantee convergence to ground state
`dtau`	Should be set *small* enough to limit Trotter errors
`Nwrap`	Should be set *small* enough to keep `Precision`s small
`Nsweep`	Should be set *large* enough for bins to be of the order of the auto-correlation time
`Nbin`	Should be set *large* enough to provide desired statistics
`nskip`	Should be set *large* enough to allow for equilibration ($\sim$ autocorrelation time)
`Nrebin`	Can be set to 1 when `Nsweep` is large enough; otherwise, and for testing, larger values can be used
Stabilization scheme	Use the default `STAB3` – newest and fastest, if it works for your model; alternatives are: `STAB1` – simplest, for reference only; `STAB2` – with additional normalizations; and `LOG` – for dealing with more extreme scales (see also Tab. 13)
Parallelism	For some models and systems, restricting parallelism in OpenBLAS can improve performance: try setting `OPENBLAS_NUM_THREADS=1` in the shell

Table 16: Rules of thumb for obtaining best results and performance from ALF. It is important to fine tune the parameters to the specific model under consideration and perform sanity checks throughout. Most suggestions can severely impact performance and numerical stability if overdone.

In particular, for the stabilization of the involved matrix multiplications we rely on routines from LAPACK. Notice that results are very likely to change depending on the specific implementation of the library used[7]. In order to deal with this possibility, we offer a simple baseline which can be used as a quick check as tho whether results depend on the library used for linear algebra routines. Namely, we have included QR-decomposition related routines of the LAPACK-3.6.1 reference implementation from `http://www.netlib.org/lapack/`, which you can use by running the script `configure.sh`, (described in Sec. 6), with the flag `STAB1` and recompiling ALF[8]. The stabilization flags available are described in Tables 13 and 16. The performance of the package is further discussed in Sec. A.

[7]The linked library should implement at least the LAPACK-3.4.0 interface.

[8]This flag may trigger compiling issues, in particular, the Intel ifort compiler version 10.1 fails for all optimization levels.

7. The plain vanilla Hubbard model on the square lattice

All the necessary data structures necessary to implement a given model have been introduced in the previous sections. Here we show how to implement the Hubbard model by specifying the lattice, the hopping, the interaction, the trial wave function (if required), and observables. Consider the *plain vanilla* Hubbard model written as:

$$\mathcal{H} = -t \sum_{\langle \boldsymbol{i},\boldsymbol{j} \rangle, \sigma=\uparrow,\downarrow} \left(\hat{c}^{\dagger}_{\boldsymbol{i},\sigma} \hat{c}_{\boldsymbol{j},\sigma} + H.c. \right) - \frac{U}{2} \sum_{\boldsymbol{i}} \left[\hat{c}^{\dagger}_{\boldsymbol{i},\uparrow} \hat{c}_{\boldsymbol{i},\uparrow} - \hat{c}^{\dagger}_{\boldsymbol{i},\downarrow} \hat{c}_{\boldsymbol{i},\downarrow} \right]^2 - \mu \sum_{\boldsymbol{i},\sigma} \hat{c}^{\dagger}_{\boldsymbol{i},\sigma} \hat{c}_{\boldsymbol{i},\sigma}. \tag{124}$$

Here $\langle \boldsymbol{i}, \boldsymbol{j} \rangle$ denotes nearest neighbors. We can make contact with the general form of the Hamiltonian (see Eq. 2) by setting: $N_{\mathrm{fl}} = 2$, $N_{\mathrm{col}} \equiv \texttt{N\_SUN} = 1$, $M_T = 1$,

$$T^{(ks)}_{xy} = \begin{cases} -t & \text{if } x, y \text{ are nearest neighbors} \\ -\mu & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \tag{125}$$

$M_V = N_{\text{unit-cell}}$, $U_k = \frac{U}{2}$, $V^{(k,s=1)}_{xy} = \delta_{x,y}\delta_{x,k}$, $V^{(k,s=2)}_{xy} = -\delta_{x,y}\delta_{x,k}$, $\alpha_{ks} = 0$ and $M_I = 0$. The coupling of the HS fields to the z-component of the magnetization breaks the $SU(2)$ spin symmetry. Nevertheless, the z-component of the spin remains a good quantum number such that the imaginary-time propagator – for a given HS field – is block diagonal in this quantum number. This corresponds to the flavor index which runs from one to two labeling spin up and spin down degrees of freedom. We note that in this formulation the hopping matrix can be flavor dependent such that a Zeeman magnetic field can be introduced. If the chemical potential is set to zero, this will not generate a negative sign problem [34, 84, 85]. The code that we describe below can be found in the module `Prog/Hamiltonians/Hamiltonian_plain_vanilla_hubbard_mod.F90`. Editing this file may be a good starting point to implement a new model Hamiltonian.

7.1. Setting the Hamiltonian: `Ham_set`

The main program will call the subroutine `Ham_set` in the module `Hamiltonian_plain_vanilla_hubbard_mod.F90`. The latter subroutine defines the public variables

```
Type (Operator),     dimension(:,:), allocatable :: Op_V    ! Interaction
Type (Operator),     dimension(:,:), allocatable :: Op_T    ! Hopping
Type (WaveFunction), dimension(:),   allocatable :: WF_L    ! Left trial wave function
Type (WaveFunction), dimension(:),   allocatable :: WF_R    ! Right trial wave function
Type (Fields)        :: nsigma                              ! Fields
Integer              :: Ndim                                ! Number of sites
Integer              :: N_FL                                ! number of flavors
Integer              :: N_SUN                               ! Number of colors
Integer              :: Ltrot                               ! Total number of trotter silces
Integer              :: Thtrot                              ! Number of trotter slices
                                                            ! reserved for projection
Logical              :: Projector                           ! Projector code
Integer              :: Group_Comm                          ! Group communicator for MPI
Logical              :: Symm                                ! Symmetric trotter
```

which specify the model. The routine `Ham_set` will first read the parameter file, then set the lattice, `Call Ham_latt`, set the hopping, `Call Ham_hop`, set the interaction, `call Ham_V` and if required the trial wave function `call Ham_trial`. The parameters are read in from the file `parameters`, see Sec. 5.7.1.

7.2. The lattice: `Call Ham_latt`

For the square lattice the routine reads:

```
a1_p(1) = 1.0  ; a1_p(2) = 0.d0
a2_p(1) = 0.0  ; a2_p(2) = 1.d0
L1_p    = dble(L1)*a1_p
L2_p    = dble(L2)*a2_p
Call Make_Lattice(L1_p, L2_p, a1_p, a2_p, Latt)

Latt_unit\%Norb = 1
```

```
Latt_unit\%N_coord = 2
allocate(Latt_unit\%Orb_pos_p(Latt_unit\%Norb,2))
Latt_unit\%Orb_pos_p(1, :) = [0.d0, 0.d0]
Ndim = Latt\%N*Latt_unit\%Norb
```

The routine also sets the number of single particle states per flavor and color: `Ndim=Latt%N*Latt_unit%Norb`.

7.3. The hopping: `Call Ham_hop`

The hopping matrix is implemented as follows. We allocate an array of dimension 1×1 of type operator called `Op_T` and set the dimension for the hopping matrix to $N = N_{\mathrm{dim}}$. One allocates and initializes this type by a single call to the subroutine `Op_make`:

```
call Op_make(Op_T(1,N_FL),Ndim)
```

Since the hopping does not break down into small blocks, we have $\boldsymbol{P} = \mathbb{1}$ and

```
Do nf = 1, N_FL
  Do i = 1,Ndim
     Op_T(1,nf)%P(i) = i
  Enddo
Enddo
```

We set the hopping matrix with

```
Do nf = 1, N_FL
   Do I = 1, Latt%N
      Ix = Latt%nnlist(I,1,0)
      Iy = Latt%nnlist(I,0,1)
      Op_T(1,nf)%O(I,  Ix) = cmplx(-Ham_T,    0.d0, kind(0.D0))
      Op_T(1,nf)%O(Ix, I ) = cmplx(-Ham_T,    0.d0, kind(0.D0))
      Op_T(1,nf)%O(I,  Iy) = cmplx(-Ham_T,    0.d0, kind(0.D0))
      Op_T(1,nf)%O(Iy, I ) = cmplx(-Ham_T,    0.d0, kind(0.D0))
      Op_T(1,nf)%O(I,  I ) = cmplx(-Ham_chem, 0.d0, kind(0.D0))
   Enddo
   Op_T(1,nf)%g     = -Dtau
   Op_T(1,nf)%alpha = cmplx(0.d0,0.d0, kind(0.D0))
   Call Op_set(Op_T(1,nf))
Enddo
```

Here, the integer function `j= Latt%nnlist(I,n,m)` is defined in the lattice module and returns the index of the lattice site $\boldsymbol{I} + n\boldsymbol{a}_1 + m\boldsymbol{a}_2$. Note that periodic boundary conditions are already taken into account. The hopping parameter `Ham_T` as well as the chemical potential `Ham_chem` are read from the parameter file. To completely define the hopping we further set: `Op_T(1,nf)%g = -Dtau` , `Op_T(1,nf)%alpha = cmplx(0.d0,0.d0, kind(0.D0))` and call the routine `Op_set(Op_T(1,nf))` so as to generate the unitary transformation and eigenvalues as specified in Table 2. Recall that for the hopping, the variable `Op_set(Op_T(1,nf))%type` is not required. Note that although a checkerboard decomposition is not used here, it can be implemented by considering a larger number of sparse hopping matrices.

7.4. The interaction: `Call Ham_V`

To implement the interaction, we allocate an array of `Operator` type. The array is called `Op_V` and has dimensions $N_{\mathrm{dim}} \times N_{\mathrm{fl}} = N_{\mathrm{dim}} \times 2$. We set the dimension for the interaction term to $N = 1$, and allocate and initialize this array of type `Operator` by repeatedly calling the subroutine `Op_make`:

```
Allocate(Op_V(Ndim,N_FL))
do nf = 1,N_FL
   do i  = 1, Ndim
      Call Op_make(Op_V(i,nf), 1)
   enddo
enddo
Do nf = 1,N_FL
   X = 1.d0
```

```
   if (nf == 2)  X = -1.d0
   Do i = 1,Ndim
      nc = nc + 1
      Op_V(i,nf)%P(1)   = I
      Op_V(i,nf)%O(1,1) = cmplx(1.d0, 0.d0, kind(0.D0))
      Op_V(i,nf)%g      = X*SQRT(CMPLX(DTAU*ham_U/2.d0, 0.D0, kind(0.D0)))
      Op_V(i,nf)%alpha  = cmplx(0.d0, 0.d0, kind(0.D0))
      Op_V(i,nf)%type   = 2
      Call Op_set( Op_V(i,nf) )
   Enddo
Enddo
```

In the above, one will see explicitly that there is a sign difference between the coupling of the HS field in the two flavor sectors.

7.5. The trial wave function: `Call Ham_Trial`

As argued in Sec. 3.1, it is useful to generate the trial wave function from a non-interacting trial Hamiltonian. Here we will use the same left and right flavor-independent trial wave functions that correspond to the ground state of:

$$\hat{H}_T = -t\sum_{\boldsymbol{i}}\left[\left(1+(-1)^{i_x+i_y}\delta\right)\hat{c}^{\dagger}_{\boldsymbol{i}}\hat{c}_{\boldsymbol{i}+\boldsymbol{a}_x} + (1-\delta)\,\hat{c}^{\dagger}_{\boldsymbol{i}}\hat{c}_{\boldsymbol{i}+\boldsymbol{a}_y} + H.c.\right] \equiv \sum_{\boldsymbol{i},\boldsymbol{j}}\hat{c}^{\dagger}_{\boldsymbol{i}}h_{\boldsymbol{i},\boldsymbol{j}}\hat{c}_{\boldsymbol{i}}. \tag{126}$$

For the half-filled case, the dimerization $\delta = 0^+$ opens up a gap at half-filling, thus generating the desired non-degenerate trial wave function that has the same symmetries (particle-hole for instance) as the trial Hamiltonian.

Diagonalization of $h_{\boldsymbol{i},\boldsymbol{j}}$, $U^{\dagger}hU = \text{Diag}\left(\epsilon_1,\cdots,\epsilon_{N_{\text{dim}}}\right)$ with $\epsilon_i < \epsilon_j$ for $i<j$, allows us to define the trial wave function. In particular, for the half-filled case, we set

```
Do s = 1, N_fl
   Do x = 1,Ndim
      Do n = 1, N_part
         WF_L(s)%P(x,n)  =  U_{x,n}
         WF_R(s)%P(x,n)  =  U_{x,n}
      Enddo
   Enddo
Enddo
```

with `N_part = Ndim/2`. The variable `Degen` belonging to the `WaveFunction` type is given by `Degen`$=\epsilon_{N_{\text{Part}}+1} - \epsilon_{N_{\text{Part}}}$. This quantity should be greater than zero for non-degenerate trial wave functions.

7.6. Observables

At this point, all the information for starting the simulation has been provided. The code will sequentially go through the operator list `Op_V` and update the fields. Between time slices `LOBS_ST` and `LOBS_EN` the main program will call the routine `Obser(GR,Phase,Ntau)` which is provided by the user and handles equal-time correlation functions. If `Ltau=1` the main program will call the routine `ObserT(NT, GT0,G0T,G00,GTT, PHASE)` which is again provided by the user and handles imaginary-time displaced correlation functions.

The users have to implement the observables they want to compute or use the predefined structures of Chap. 8. Here we will describe how to proceed.

7.6.1. Allocating space for the observables: `Call Alloc_obs(Ltau)`

For four scalar or vector observables, the user will have to declare the following:

```
Allocate ( Obs_scal(4) )
Do I = 1,Size(Obs_scal,1)
   select case (I)
   case (1)
      N = 2;  Filename ="Kin"
   case (2)
```

```
      N = 1;  Filename ="Pot"
   case (3)
      N = 1;  Filename ="Part"
   case (4)
      N = 1,  Filename ="Ener"
   case default
      Write(6,*) ' Error in Alloc_obs '
   end select
   Call Obser_Vec_make(Obs_scal(I), N, Filename)
enddo
```

Here, `Obs_scal(1)` contains a vector of two observables so as to account for the x- and y-components of the kinetic energy for example.

For equal-time correlation functions we allocate `Obs_eq` of type `Obser_Latt`. Here we include the calculation of spin-spin and density-density correlation functions alongside equal-time Green functions.

```
Allocate ( Obs_eq(5) )
Do I = 1,Size(Obs_eq,1)
   select case (I)
   case (1)
      Filename = "Green"
   case (2)
      Filename = "SpinZ"
   case (3)
      Filename = "SpinXY"
   case (4)
      Filename = "SpinT"
   case (5)
      Filename = "Den"
   case default
      Write(6,*) "Error in Alloc_obs"
   end select
Nt = 1
Channel = "--"
Call Obser_Latt_make(Obs_eq(I), Nt, Filename, Latt, Latt_unit, Channel, dtau)
```

Be aware that `Obser_Latt_make` does not copy the Bravais lattice `Latt` and unit cell `Latt_unit`, but links them through pointers to be more memory efficient. One can have different lattices attached to different observables by declaring additional instances of `Type(Lattice)` and `Type(Unit_cell)`. For equal-time correlation functions, we set `Nt = 1` and `Channel` does not have a meaning.

If `Ltau = 1`, then the code will allocate space for time displaced quantities. The same structure as for equal-time correlation functions will be used, albeit with `Nt = Ltrot + 1` and the channel should be set. Whith `Channel="PH"`, the analysis will assume the observable to be particle-hole symmetric. For more details on the meaning of this parameter, see Sec. 10.

At the beginning of each bin, the main program will set the bin observables to zero by calling the routine `Init_obs(Ltau)`. The user does not have to edit this routine.

7.6.2. Measuring equal-time observables: `Obser(GR,Phase,Ntau)`

The equal-time Green function,

$$\texttt{GR(x,y,}\sigma\texttt{)} = \langle \hat{c}_{x,\sigma} \hat{c}^{\dagger}_{y,\sigma} \rangle, \tag{127}$$

the phase factor `phase` [Eq. (115)], and time slice `Ntau` are provided by the main program.

Here, x and y label both unit cell as well as the orbital within the unit cell. For the Hubbard model described here, x corresponds to the unit cell. The Green function does not depend on the color index, and is diagonal in flavor. For the $SU(2)$ symmetric implementation there is only one flavor, $\sigma = 1$ and the Green function is independent on the spin index. This renders the calculation of the observables particularly easy.

An explicit calculation of the potential energy $\langle U \sum_{\boldsymbol{i}} \hat{n}_{\boldsymbol{i},\uparrow} \hat{n}_{\boldsymbol{i},\downarrow} \rangle$ reads

```
Obs_scal(2)%N        = Obs_scal(2)%N + 1
Obs_scal(2)%Ave_sign = Obs_scal(2)%Ave_sign + Real(ZS,kind(0.d0))
```

```
Do i = 1,Ndim
   Obs_scal(2)%Obs_vec(1) = Obs_scal(2)%Obs_vec(1) + (1-GR(i,i,1))*(1-GR(i,i,2))*Ham_U*ZS*ZP
Enddo
```

Here ZS $=$ sign(C) [see Eq. (24)], ZP $= \frac{e^{-S(C)}}{\mathrm{Re}\left[e^{-S(C)}\right]}$ [see Eq. (115)] and Ham_U corresponds to the Hubbard-U term.

Equal-time correlations are also computed in this routine. As an explicit example, we consider the equal-time density-density correlation:

$$\langle \hat{n}_{\boldsymbol{i}} \hat{n}_{\boldsymbol{j}} \rangle - \langle \hat{n}_{\boldsymbol{i}} \rangle \langle \hat{n}_{\boldsymbol{j}} \rangle, \tag{128}$$

with

$$\hat{n}_{\boldsymbol{i}} = \sum_{\sigma} \hat{c}^{\dagger}_{\boldsymbol{i},\sigma} \hat{c}_{\boldsymbol{i},\sigma}. \tag{129}$$

For the calculation of such quantities, it is convenient to define:

$$\texttt{GRC(x,y,s)} = \delta_{x,y} - \texttt{GR(y,x,s)} \tag{130}$$

such that GRC(x,y,s) corresponds to $\langle\langle \hat{c}^{\dagger}_{x,s} \hat{c}_{y,s} \rangle\rangle$. In the program code, the calculation of the equal-time density-density correlation function looks as follows:

```
Obs_eq(4)%N = Obs_eq(4)%N + 1            ! Even if it is redundant, each observable
                                         ! carries its own counter and sign.
Obs_eq(4)%Ave_sign = Obs_eq(4)%Ave_sign + Real(ZS,kind(0.d0))
Do I = 1,Ndim
   Do J = 1,Ndim
      imj = latt%imj(I,J)
      Obs_eq(4)%Obs_Latt(imj,1,1,1) =  Obs_eq(4)%Obs_Latt(imj,1,1,1) + &
                     &     ( (GRC(I,I,1)+GRC(I,I,2)) * (GRC(J,J,1)+GRC(J,J,2))       + &
                     &        GRC(I,J,1)*GR(I,J,1)    +  GRC(I,J,2)*GR(I,J,2)  ) * ZP * ZS
   Enddo
   Obs_eq(4)%Obs_Latt0(1) = Obs_eq(4)%Obs_Latt0(1) + (GRC(I,I,1)+GRC(I,I,2))*ZP*ZS
Enddo
```

At the end of each bin the main program will call the routine Pr_obs(LTAU). This routine will append the result of the bins in the specified file, with appropriate suffix.

7.6.3. Measuring time displaced observables: ObserT(NT, GT0, G0T, G00, GTT, PHASE)

This subroutine is called by the main program at the beginning of each sweep, provided that LTAU is set to unity. NT runs from 0 to Ltrot and denotes the imaginary time difference. For a given time displacement, the main program provides:

$$\begin{aligned}
\texttt{GT0(x,y,s)} &= \langle\langle \hat{c}_{x,s}(Nt\Delta\tau) \hat{c}^{\dagger}_{y,s}(0) \rangle\rangle = \langle\langle \mathcal{T} \hat{c}_{x,s}(Nt\Delta\tau) \hat{c}^{\dagger}_{y,s}(0) \rangle\rangle \\
\texttt{G0T(x,y,s)} &= -\langle\langle \hat{c}^{\dagger}_{y,s}(Nt\Delta\tau) \hat{c}_{x,s}(0) \rangle\rangle = \langle\langle \mathcal{T} \hat{c}_{x,s}(0) \hat{c}^{\dagger}_{y,s}(Nt\Delta\tau) \rangle\rangle \\
\texttt{G00(x,y,s)} &= \langle\langle \hat{c}_{x,s}(0) \hat{c}^{\dagger}_{y,s}(0) \rangle\rangle \\
\texttt{GTT(x,y,s)} &= \langle\langle \hat{c}_{x,s}(Nt\Delta\tau) \hat{c}^{\dagger}_{y,s}(Nt\Delta\tau) \rangle\rangle.
\end{aligned} \tag{131}$$

In the above we have omitted the color index since the Green functions are color independent. The time displaced spin-spin correlations $4\langle\langle \hat{S}^z_{\boldsymbol{i}}(\tau) \hat{S}^z_{\boldsymbol{j}}(0) \rangle\rangle$ are thereby given by:

$$\begin{aligned}
4\langle\langle \hat{S}^z_{\boldsymbol{i}}(\tau) \hat{S}^z_{\boldsymbol{j}}(0) \rangle\rangle = &\ (\texttt{GTT(I,I,1)} - \texttt{GTT(I,I,2)}) * (\texttt{G00(J,J,1)} - \texttt{G00(J,J,2)}) \\
&- \texttt{G0T(J,I,1)} * \texttt{GT0(I,J,1)} - \texttt{G0T(J,I,2)} * \texttt{GT0(I,J,2)}
\end{aligned} \tag{132}$$

The handling of time displaced correlation functions is identical to that of equal-time correlations.

7.7. Numerical precision

Information on the numerical stability is included in the following lines of the corresponding file `info`. For a *short* simulation on a 4×4 lattice at $U/t = 4$ and $\beta t = 10$ we obtain

```
Precision Green  Mean, Max :    5.0823874429126405E-011   5.8621144596315844E-006
Precision Phase, Max       :    0.0000000000000000
Precision tau    Mean, Max :    1.5929357848647394E-011   1.0985132530727526E-005
```

showing the mean and maximum difference between the *wrapped* and from scratched computed equal and time-displaced Green functions [6]. A stable code should produce results where the mean difference is smaller than the stochastic error. The above example shows a very stable simulation since the Green function is of order one.

7.8. Running the code and testing

To test the code, one can carry out high precision simulations. After compilation, the executable `Hubbard_Plain_Vanilla.out` is found in the directory `$ALF_DIR/Prog/` and can be run from any directory containing the files `parameters` and `seeds` (See Sec. 5.7).

Alternatively, it may be convenient to use `pyALF` to compile and run the code.

One dimensional case

The `pyALF` python script `Hubbard_Plain_Vanilla.py` runs the projective version of the code for the four site Hubbard model. At $\theta t = 10$, $\Delta\tau t = 0.05$ with the symmetric Trotter decomposition, we obtain after 40 bins of each 2000 sweeps the total energy:

$$\langle \hat{H} \rangle = -2.103750 \pm 0.004825$$

and the exact result is

$$\langle \hat{H} \rangle_{\texttt{Exact}} = -2.100396$$

Two dimensional case

For the two-dimensional case, with similar parameters, we obtain,

	QMC	Exact
Total energy	-13.618 $\pm$ 0.002	-13.6224
$\boldsymbol{Q} = (\pi, \pi)$ spin correlations	3.630 $\pm$ 0.006	3.64

The exact results stem from Ref. [86] and the slight discrepancies from the exact results can be assigned to the finite value of $\Delta\tau$. Note that all the simulations were carried out with the default value of the Hubbard interaction, $U/t = 4$.

8. Predefined Structures

ALF includes modules providing predefined structures which the user can combine together or use as templates for defining new structures, namely:

- lattices and unit cells – `Predefined_Latt_mod.F90`
- hopping Hamiltonians – `Predefined_Hop_mod.F90`
- interaction Hamiltonians – `Predefined_Int_mod.F90`
- observables – `Predefined_Obs_mod.F90`
- trial wave functions – `Predefined_Trial_mod.F90`

which are defined using the data structures defined in the Sec. 5, as described in this section.

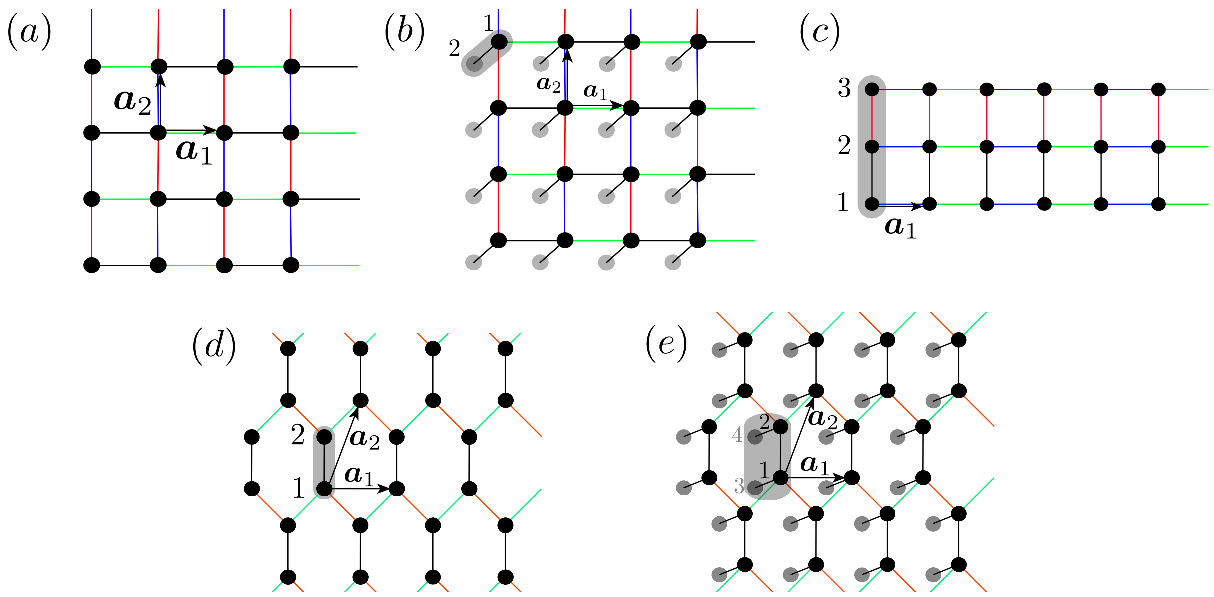

Figure 5: Predefined lattices in ALF: (a) square, (b) bilayer square, (c) 3-leg ladder (d) honeycomb and (e) bilayer honeycomb. Nontrivial unit cells are shown as gray regions, while gray sites belong to the second layer in bilayer systems. The links between the orbitals denote the hopping matrix elements and we have assumed, for the purpose of the plot, the absence of hopping in the second layer for bilayer systems. The color coding of the links denotes the checkerboard decomposition.

8.1. Predefined lattices

The types `Lattice` and `Unit_cell`, described in Section 5.3, allow us to define arbitrary one- and two-dimensional Bravais lattices. The subroutine `Predefined_Latt` provides some of the most common lattices, as described bellow.

The subroutine is called as:

```
Predefined_Latt(Lattice_type, L1, L2, Ndim, List, Invlist, Latt, Latt_Unit)
```

which returns a lattice of size `L1`×`L2` of the given `Lattice_type`, as detailed in Table 17. Notice that the orbital position `Latt_Unit%Orb_pos_p(1,:)` is set to zero unless otherwise specified.

In order to easily keep track of the orbital and unit cell, `List` and `Invlist` make use of a super-index, defined as shown below:

```
nc = 0                                  ! Super-index labeling unit cell and orbital
Do I = 1,Latt%N                         ! Unit-cell index
   Do no = 1,Norb                       ! Orbital index
      nc = nc + 1
      List(nc,1) = I                    ! Unit-cell of super index nc
      List(nc,2) = no                   ! Orbital of super index nc
      Invlist(I,no) = nc                ! Super-index for given unit cell and orbital
   Enddo
Enddo
```

With the above lists one can run through all the orbitals and at each time keep track of the unit-cell and orbital index. We note that when translation symmetry is completely absent one can work with a single unit cell, and the number of orbitals will then correspond to the number of lattice sites.

8.1.1. Square lattice, Fig. 5(a)

The choice `Lattice_type = "Square"` sets $\boldsymbol{a}_1 = (1,0)$ and $\boldsymbol{a}_2 = (0,1)$ and for an $L_1 \times L_2$ lattice $\boldsymbol{L}_1 = L_1\boldsymbol{a}_1$ and $\boldsymbol{L}_2 = L_2\boldsymbol{a}_2$:

Argument	Type	Role	Description
`Lattice_type`	String	Input	Lattice configuration, which can take the values: - `Square` - `Honeycomb` - `Pi_Flux` (deprecated) - `N_leg_ladder` - `Bilayer_square` - `Bilayer_honeycomb`
`L1, L2`	Integer	Input	Lattice sizes (set `L2=1` for 1D lattices)
`Ndim`	Integer	Output	Total number of orbitals
`List`	Integer	Output	For every site index `I` $\in [1,$`Ndim`$]$, stores the corresponding lattice position, `List(I,1)`, and the (local) orbital index, `List(I,2)`
`Invlist`	Integer	Output	For every `lattice_position` $\in [1,$`Latt%N`$]$ and `orbital` $\in [1,$`Norb`$]$ stores the corresponding site index `I(lattice_position,orbital)`
`Latt`	Lattice	Output	Sets the lattice
`Latt_Unit`	Unit_cell	Output	Sets the unit cell

Table 17: Arguments of the subroutine `Predefined_Latt`. Note that the `Pi_Flux` lattice is deprecated for the moment since it can be emulated with the Square lattice with half a flux quanta piercing each plaquette.

```
Latt_Unit%N_coord   = 2
Latt_Unit%Norb      = 1
Latt_Unit%Orb_pos_p(1,:) = 0.d0
a1_p(1) =  1.0  ; a1_p(2) =  0.d0
a2_p(1) =  0.0  ; a2_p(2) =  1.d0
L1_p    =  dble(L1)*a1_p
L2_p    =  dble(L2)*a2_p
Call Make_Lattice( L1_p, L2_p, a1_p,  a2_p, Latt )
```

Also, the number of orbitals per unit cell is given by `NORB=1` such that $N_{\text{dim}} \equiv N_{\text{unit-cell}} \cdot$ `NORB` $=$ `Latt%N` $\cdot$ `NORB`, since $N_{\text{unit-cell}} =$ `Latt%N`.

8.1.2. Bilayer Square lattice, Fig. 5(b)

The "`Bilayer_square`" configuration sets:

```
Latt_Unit%Norb     = 2
Latt_Unit%N_coord  = 2
do no = 1,2
   Latt_Unit%Orb_pos_p(no,1) = 0.d0
   Latt_Unit%Orb_pos_p(no,2) = 0.d0
   Latt_Unit%Orb_pos_p(no,3) = real(1-no,kind(0.d0))
enddo
Latt%a1_p(1) =  1.0  ; Latt%a1_p(2) =  0.d0
Latt%a2_p(1) =  0.0  ; Latt%a2_p(2) =  1.d0
Latt%L1_p    =  dble(L1)*a1_p
Latt%L2_p    =  dble(L2)*a2_p
Call Make_Lattice( L1_p, L2_p, a1_p,  a2_p, Latt )
```

8.1.3. N-leg Ladder lattice, Fig. 5(c)

The "`N_leg_ladder`" configuration sets:

```
Latt_Unit%Norb     = L2
Latt_Unit%N_coord  = 1
```

```
do no = 1,L2
   Latt_Unit%Orb_pos_p(no,1) = 0.d0
   Latt_Unit%Orb_pos_p(no,2) = real(no-1,kind(0.d0))
enddo
a1_p(1) =  1.0   ; a1_p(2) =  0.d0
a2_p(1) =  0.0   ; a2_p(2) =  1.d0
L1_p    =  dble(L1)*a1_p
L2_p    =          a2_p
Call Make_Lattice( L1_p, L2_p, a1_p,  a2_p, Latt )
```

8.1.4. Honeycomb lattice, Fig. 5(d)

In order to carry out simulations on the Honeycomb lattice, which is a triangular Bravais lattice with two orbitals per unit cell, we choose `Lattice_type="Honeycomb"` , which sets

```
a1_p(1) =  1.D0   ; a1_p(2) =  0.d0
a2_p(1) =  0.5D0  ; a2_p(2) =  sqrt(3.D0)/2.D0
L1_p    =  Dble(L1) * a1_p
L2_p    =  dble(L2) * a2_p
Call Make_Lattice( L1_p, L2_p, a1_p,  a2_p, Latt )
Latt_Unit%Norb    = 2
Latt_Unit%N_coord = 3
Latt_Unit%Orb_pos_p(1,:) = 0.d0
Latt_Unit%Orb_pos_p(2,:) = (a2_p(:) - 0.5D0*a1_p(:) ) * 2.D0/3.D0
```

The coordination number of this lattice is `N_coord=3` and the number of orbitals per unit cell, `NORB=2`. The total number of orbitals is therefore N_{dim}=`Latt%N*NORB`.

8.1.5. Bilayer Honeycomb lattice, Fig. 5(e)

The "`Bilayer_honeycomb`" configuration sets:

```
Latt_Unit%Norb     = 4
Latt_Unit%N_coord  = 3
Latt_unit%Orb_pos_p = 0.d0
do n = 1,2
   Latt_Unit%Orb_pos_p(1,n) = 0.d0
   Latt_Unit%Orb_pos_p(2,n) = (a2_p(n) - 0.5D0*a1_p(n) ) * 2.D0/3.D0
   Latt_Unit%Orb_pos_p(3,n) = 0.d0
   Latt_Unit%Orb_pos_p(4,n) = (a2_p(n) - 0.5D0*a1_p(n) ) * 2.D0/3.D0
enddo
Latt_Unit%Orb_pos_p(3,3) = -1.d0
Latt_Unit%Orb_pos_p(4,3) = -1.d0
a1_p(1) =  1.D0   ; a1_p(2) =  0.d0
a2_p(1) =  0.5D0  ; a2_p(2) =  sqrt(3.D0)/2.D0
L1_p    =  dble(L1)*a1_p
L2_p    =  dble(L2)*a2_p
Call Make_Lattice( L1_p, L2_p, a1_p,  a2_p, Latt )
```

8.1.6. π-Flux lattice (deprecated)

The Pi_Flux lattice has been deprecated, since it can be emulated with the Square lattice with half a flux quanta piercing each plaquette. Nonetheless, the configuration is still available, and sets:

```
Latt_Unit%Norb    = 2
Latt_Unit%N_coord = 4
a1_p(1) =  1.D0   ; a1_p(2) =   1.d0
a2_p(1) =  1.D0   ; a2_p(2) =  -1.d0
Latt_Unit%Orb_pos_p(1,:) = 0.d0
Latt_Unit%Orb_pos_p(2,:) = (a1_p(:) - a2_p(:))/2.d0
L1_p    =  dble(L1) * (a1_p - a2_p)/2.d0
L2_p    =  dble(L2) * (a1_p + a2_p)/2.d0
```

8.2. Generic hopping matrices on bravais lattices

8.2.1. Setting up the hopping matrix: the `Hopping_Matrix_type`

The module `Predefined_Hopping` provides a generic way to specify a hopping matrix on a multi-orbital Bravais lattice. The only assumption that we make is translation symmetry. We allow for twisted boundary conditions in the $\boldsymbol{L}_1$ and $\boldsymbol{L}_2$ lattice directions. The twist is given by `Phi_X` and `Phi_Y` respectively. If the flag `bulk=.true.`, then the twist is implemented with a vector potential. Otherwise, if `bulk=.false.`, the twist is imposed at the boundary. The routine also accounts for the inclusion of a total number of `N_Phi` flux quanta traversing the lattice. All phase factors mentioned above can be flavor dependent. The checkerboard decomposition can also be specified in this module. All information, including the checkerboard decomposition is specified in the `Hopping_Matrix_type` type (see below) from which the array of operator type `OP_T`, accounting for the single particle propagation in one time step, as well as the Kinetic energy can be derived.

Generic hopping matrices

The generic Hopping Hamiltonian reads:

$$\hat{H}_T = \sum_{(i,\delta),(j,\delta'),s,\sigma} T^{(s)}_{(i,\delta),(j,\delta')} \hat{c}^{\dagger}_{(i,\delta),s,\sigma} e^{\frac{2\pi i}{\Phi_0}\int_{i+\delta}^{j+\delta'} \boldsymbol{A}^{(s)}(\boldsymbol{l})d\boldsymbol{l}} \hat{c}_{(j,\delta'),s,\sigma} \tag{133}$$

with boundary conditions

$$\hat{c}^{\dagger}_{(i+L_i,\delta),s,\sigma} = e^{-2\pi i \frac{\Phi_i^{(s)}}{\Phi_0}} e^{\frac{2\pi i}{\Phi_0}\chi^{(s)}_{L_i}(i+\delta)} \hat{c}^{\dagger}_{(i,\delta),s,\sigma}. \tag{134}$$

Both the twist and vector potential can have a flavor dependency. For now onwards we will mostly omit the flavor index s.

Phase factors. The vector potential accounts for an orbital magnetic field that is implemented in the Landau gauge: $\boldsymbol{A}(\boldsymbol{x}) = -B(y, 0, 0)$ with $\boldsymbol{x} = (x, y, z)$. Φ_0 corresponds to the flux quanta and the scalar function χ is defined through as:

$$\boldsymbol{A}(\boldsymbol{x} + \boldsymbol{L}_i) = \boldsymbol{A}(\boldsymbol{x}) + \boldsymbol{\nabla}\chi_{L_i}(\boldsymbol{x}). \tag{135}$$

Provided that the bare hopping Hamiltonian, T, is invariant under lattice translations, $\hat{H}_T$ commutes with magnetic translations that satisfy the Algebra:

$$\hat{T}_{\boldsymbol{a}}\hat{T}_{\boldsymbol{b}} = e^{\frac{2\pi i}{\Phi_0}\boldsymbol{B}\cdot(\boldsymbol{a}\times\boldsymbol{b})}\hat{T}_{\boldsymbol{b}}\hat{T}_{\boldsymbol{a}}. \tag{136}$$

On the torus, the uniqueness of the wave functions requires that $\hat{T}_{\boldsymbol{L}_1}\hat{T}_{\boldsymbol{L}_2} = \hat{T}_{\boldsymbol{L}_2}\hat{T}_{\boldsymbol{L}_1}$ such that

$$\frac{\boldsymbol{B}\cdot(\boldsymbol{a}\times\boldsymbol{b})}{\Phi_0} = N_\Phi \tag{137}$$

with N_Φ an integer. The variable `N_Phi`, specified in the parameter file, denotes the number of flux quanta piercing the lattice. The variables `Phi_X` and `Phi_Y` also in the parameter file denote the twists – in units of the flux quanta – along the $\boldsymbol{L}_1$ and $\boldsymbol{L}_2$ directions. There are gauge equivalent ways to insert the twist in the boundary conditions. In the above we have inserted twist as a boundary condition such that for example setting `Phi_1=0.5` corresponds to anti-periodic boundary conditions along the L_1 axis. Alternatively we can consider the Hamiltonian:

$$\hat{H}_T = \sum_{(i,\delta),(j,\delta'),s,\sigma} T^{(s)}_{(i,\delta),(j,\delta')} \tilde{c}^{\dagger}_{(i,\delta),s,\sigma} e^{\frac{2\pi i}{\Phi_0}\int_{i+\delta}^{j+\delta'} (\boldsymbol{A}(\boldsymbol{l})+\boldsymbol{A}_\phi)d\boldsymbol{l}} \tilde{c}_{(j,\delta'),s,\sigma} \tag{138}$$

with boundary conditions

$$\tilde{c}^{\dagger}_{(i+L_i,\delta),s,\sigma} = e^{\frac{2\pi i}{\Phi_0}\chi_{L_i}(i+\delta)} \tilde{c}^{\dagger}_{(i,\delta),s,\sigma}. \tag{139}$$

Here

$$\boldsymbol{A}_\phi = \frac{\phi_1|\boldsymbol{a}_1|}{2\pi|\boldsymbol{L}_1|}\boldsymbol{b}_1 + \frac{\phi_2|\boldsymbol{a}_2|}{2\pi|\boldsymbol{L}_2|}\boldsymbol{b}_2 \tag{140}$$

and $\boldsymbol{b}_i$ correspond to the reciprocal lattice vectors satisfying $\boldsymbol{a}_i \cdot \boldsymbol{b}_j = 2\pi\delta_{i,j}$. The logical variable `bulk` chooses between these two gauge equivalent ways of inserting the twist angle. If `bulk=true` then we use periodic boundary conditions – in the absence of an orbital field – otherwise twisted boundaries are used. The above phase factors are computed in the module function:

```
complex function Generic_hopping(i,no_i, n_1, n_2, no_j, N_Phi, Phi_1,Phi_2, Bulk,
Latt, Latt_Unit)
```

that returns the phase factor involved in the hopping of a hole from lattice site $\boldsymbol{i} + \boldsymbol{\delta}_{\mathrm{no}_i}$ to $\boldsymbol{i} + n_1\boldsymbol{a}_1 + n_1\boldsymbol{a}_2 + \boldsymbol{\delta}_{\mathrm{no}_j}$. Here $\boldsymbol{\delta}_{\mathrm{no}_i}$ is the position of the no_i orbital in the unit cell $\boldsymbol{i}$. As we will see below, the information for the phases is encoded in the type `Hopping_matrix_type`.

The Hopping matrix elements. The hopping matrix is specified assuming only translation invariance. (The point group symmetry of the lattice can be broken). That is, we assume that for each flavor index:

$$T^{(s)}_{(\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{i}+n_1\boldsymbol{a}_1+n_2\boldsymbol{a}_2,\boldsymbol{\delta}')} = T^{(s)}_{(\boldsymbol{0},\boldsymbol{\delta}),(n_1\boldsymbol{a}_1+n_2\boldsymbol{a}_2,\boldsymbol{\delta}')}. \tag{141}$$

The right hand side of the above equation is given the type `Hopping_matrix_type`.

The checkerboard decomposition. Aside from the hopping phases and hopping matrix elements, the `Hopping_matrix_type` type, contains information concerning the checkerboard decomposition. In Eq. 66 we wrote the hopping Hamiltonian as:

$$\hat{\mathcal{H}}_T = \sum_{i=1}^{N_T} \sum_{k \in \mathcal{S}_i^T} \hat{T}^{(k)}, \tag{142}$$

with the rule that if k and k' belong to the same set $\mathcal{S}_i^T$ then $\left[\hat{T}^{(k)}, \hat{T}^{(k')}\right] = 0$. In the checkerboard decomposition, $\hat{T}^{(k)}$ corresponds to hopping on a bond. The checkerboard decomposition depends on the lattice type, as well as on the hopping matrix elements. The required information is stored in `Hopping_matrix_type`. In this data type, `N_FAM` corresponds to the number of sets (or families) (N_T in the above equation). `L_FAM(1:N_FAM)` corresponds to the number of bonds in the set, and finally, `LIST_FAM(1:N_FAM, 1:max(L_FAM(:)), 2)` contains information concerning the two legs of the bonds. In the checkerboard decomposition, care has to be taken for local terms: each site occurs multiple times in the list of bonds. Since we have postulated translation symmetry, a one-dimensional array, `Multiplicity`, of length given by the number of orbitals per unit cell suffices to encode the required information. Finally, to be able to generate the imaginary time step of length $\Delta\tau$ we have to know by which fraction of $\Delta\tau$ we have to propagate each set. This information is given in the array `Prop_Fam`.

As an example we can consider the three-leg ladder lattice of Figure 5(c). Here the number of sets (or families) `N_FAM` is equal to four corresponding to the red, green, black and blue bonds. As apparent, bonds in a given set do not have common legs such that hopping instances on the bonds of a given set commute. For this three leg ladder it is apparent that the second orbital in a unit cell appears in each set or family. It hence has a multiplicity of four. On the other hand, the top and bottom orbitals have a multiplicity of 3 since they appear in only three of the four sets.

Usage: the `Hopping_Matrix_type`

There are `N_bonds` hopping matrix elements emanating from a given unit cell. For each bond, the array `List` contains the full information to define the RHS of Eq. (141). The hopping amplitudes are stored in the array `T` and the local potentials in the array `T_loc` (See Table 18). The `Hopping_Matrix_type` type also contains the information for the checkerboard decomposition.

The data in the `Hopping_matrix_type` type suffices to uniquely define the unit step propagation for the kinetic energy, and for any combinations of the `Checkerboard` and `Symm` options (see Sec. 2.3). This is carried by the call:

```
Call Predefined_Hoppings_set_OPT(Hopping_Matrix, List, Invlist, Latt, Latt_unit, Dtau,
Checkerboard, Symm, OP_T)
```

in which the operator array `OP_T(*,N_FL)` is allocated and defined. In the simplest case, where no checkerboard is used, the first dimension is unity.

The data in the `Hopping_matrix_type` type equally suffices to compute the kinetic energy. This is carried out in the routine `Predefined_Hoppings_Compute_Kin`.

Variable	Type	Description
`N_bonds`	Integer	Number of hopping matrix elements within and emanating from a unit cell
`List(N_bonds,4)`	Integer	List(•,1) $= \delta$ List(•,2) $= \delta'$ List(•,3) $= n_1$ List(•,4) $= n_2$
`T(N_bonds)`	Complex	Hopping amplitude
`T_loc(Norb)`	Complex	On site potentials (e.g., chemical potential, Zeeman field)
`N_Phi`	Integer	Number of flux quanta piercing the lattice
`Phi_X`	Real	Twist in $\boldsymbol{a}_1$ direction
`Phi_Y`	Real	Twist in $\boldsymbol{a}_2$ direction
`Bulk`	Logical	Twist as vector potential (T) or boundary condition (F).
`N_Fam`	Integer	Number of sets, N_T in Eq. (66)
`L_Fam(N_FAM)`	Integer	Number of bonds per set $\mathcal{S}^T$
`List_Fam(N_FAM,L_FAM,2)`	Integer	`List Fam(•,•,1)` = Unit cell `List Fam(•,•,2)` = Bond number
`Multiplicity(Norb)`	Integer	Number of times a given orbital occurs in the list of bonds
`Prop_Fam`	Real	The fraction of $\Delta\tau$ with which the set will be propagated

Table 18: Member variables of the `Hopping_Matrix_type` type.

8.2.2. An example: nearest neighbor hopping on the honeycomb lattice

For the honeycomb lattice of Fig. 5(d) the number of bond within and emanating from a unit cell is `N_bonds = 3`. The list array of the `Hopping_matrix_type` reads:

```
list(1,1) = 1;  list(1,2) = 2;  list(1,3) = 0;   list(1,4) =  0  ! Intra unit-cell hopping
list(2,1) = 2;  list(2,2) = 1;  list(2,3) = 0:   list(2,4) =  1  ! Inter unit-cell hopping
list(3,1) = 1;  list(3,2) = 2;  list(3,3) = 1:   list(3,4) = -1  ! Inter unit-cell hopping
T(1) = -1.0; T(2) = -1.0; T(3) = -1.0                            ! Hopping
T_loc(1) =0.0; T_loc(2) =0.0                                     ! Chemical potential
```

In the last two lines, we have set the hopping matrix element for each bond to -1 and the chemical potential to zero. The fields, can then be specified with the variables `N_phi, Phi_x, Phi_y`. Setting the twists, `Phi_x, Phi_y` to zero and looping over `N_phi` from $1 \cdots L^2$ produces the single particle spectrum of Fig. 6(a).

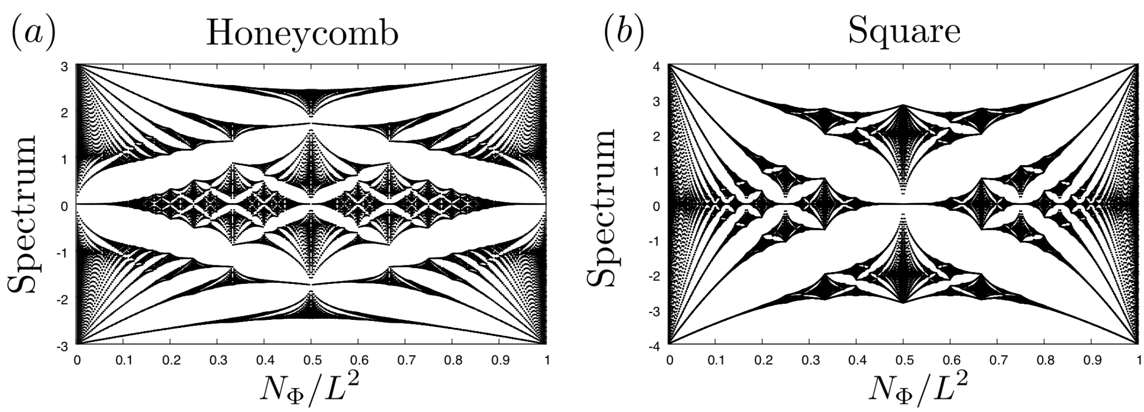

Figure 6: The single particle spectrum of the tight binding model on the honeycomb (a) and square (b) lattices as a function of the flux N_Φ. This corresponds to the well known Hofstadter butterflies.

For the Honeycomb lattice the checkerboard decomposition for the nearest neighbor hopping consists of three sets: `N_Fam = 3` each of length corresponding to the number of unit cells. In Fig. 5(d) the bond elements of these three sets are color coded. In the code, the element of the sets are specified as:

```
do I = 1,Latt%N
   Do nf = 1,N_FAM
      List_Fam(nf,I,1) = I  ! Unit cell
      List_Fam(nf,I,2) = nf ! The bond
   Enddo
enddo
Multiplicity  = 3
```

Since each site of the honeycomb lattice occurs in the three sets, the multiplicity is equal to 3.

8.2.3. Predefined hoppings

The module provides hopping and checkerboard decompositions, defining a `Hopping_Matrix` (an array of length `N_FL` of type `Hopping_Matrix_type`, see Sec. 8.2.1) for each of the following predefined lattices.

Square

The call:

```
Call Set_Default_hopping_parameters_square(Hopping_Matrix, T_vec, Chem_vec, Phi_X_vec,
        Phi_Y_vec, Bulk, N_Phi_vec, N_FL, List, Invlist, Latt, Latt_unit)
```

defines the `Hopping_Matrix` for the square lattice:

$$\hat{H}_T = \sum_{\boldsymbol{i},\sigma,s} \left(\left[\sum_{\boldsymbol{\delta}=\{\boldsymbol{a}_1,\boldsymbol{a}_2\}} -t^{(s)} \hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma} e^{\frac{2\pi i}{\Phi_0} \int_{\boldsymbol{i}}^{\boldsymbol{i}+\boldsymbol{\delta}} \boldsymbol{A}^{(s)}(\boldsymbol{l}) d\boldsymbol{l}} \hat{c}_{\boldsymbol{i}+\boldsymbol{\delta},s,\sigma} + H.c. \right] - \mu^{(s)} \hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma} \hat{c}_{\boldsymbol{i},s,\sigma} \right). \tag{143}$$

The vectors `T_vec` and `Chem_vec` have length `N_FL` and specify the hopping and the chemical potentials, while the vectors `Phi_X_vec`, `Phi_Y_vec` and `N_Phi_vec`, also of length `N_FL`, define the vector potential.

Honeycomb

The call:

```
Call Set_Default_hopping_parameters_honeycomb(Hopping_Matrix,T_vec, Chem_vec, Phi_X_vec,
        Phi_Y_vec, Bulk, N_Phi_vec, N_FL, List, Invlist, Latt, Latt_unit)
```

defines the `Hopping_Matrix` for the honeycomb lattice:

$$\begin{aligned} \hat{H}_T = & \sum_{\boldsymbol{i},\sigma,s} \left(\sum_{\boldsymbol{\delta}=\{\boldsymbol{\delta}_1,\boldsymbol{\delta}_2,\boldsymbol{\delta}_3\}} -t^{(s)} \hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma} e^{\frac{2\pi i}{\Phi_0} \int_{\boldsymbol{i}}^{\boldsymbol{i}+\boldsymbol{\delta}} \boldsymbol{A}^{(s)}(\boldsymbol{l}) d\boldsymbol{l}} \hat{c}_{\boldsymbol{i}+\boldsymbol{\delta},s,\sigma} + H.c. \right) \\ & \sum_{\boldsymbol{i},\sigma,s} -\mu^{(s)} \left(\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma} \hat{c}_{\boldsymbol{i},s,\sigma} + \hat{c}^{\dagger}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma} \hat{c}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma} \right), \end{aligned} \tag{144}$$

where the `T_vec` and `Chem_vec` have length `N_FL` and specify the hopping and the chemical potentials, while the vectors `Phi_X_vec`, `Phi_Y_vec` and `N_Phi_vec`, also of length `N_FL`, define the vector potential. Here $\boldsymbol{i}$ runs over sublattice A, and $\boldsymbol{i} + \boldsymbol{\delta}$ over the three nearest neighbors of site $\boldsymbol{i}$

Square bilayer

The call:

```
Call Set_Default_hopping_parameters_Bilayer_square(Hopping_Matrix, T1_vec, T2_vec, Tperp_vec,
        Chem_vec, Phi_X_vec, Phi_Y_vec, Bulk, N_Phi_vec, N_FL, List, Invlist, Latt, Latt_unit)
```

defines the `Hopping_Matrix` for the bilayer square lattice:

$$\begin{aligned}\hat{H}_T = &\sum_{\boldsymbol{i},\sigma,s,n}\left(\left[\sum_{\boldsymbol{\delta}=\{\boldsymbol{a}_1,\boldsymbol{a}_2\}} -t_n^{(s)}\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,n} e^{\frac{2\pi i}{\Phi_0}\int_{i}^{\boldsymbol{i}+\boldsymbol{\delta}}\boldsymbol{A}^{(s)}(\boldsymbol{l})d\boldsymbol{l}}\hat{c}_{\boldsymbol{i}+\boldsymbol{\delta},s,\sigma,n} + H.c.\right] - \mu^{(s)}\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,n}\hat{c}_{\boldsymbol{i},s,\sigma,n}\right) + \\ &\sum_{\boldsymbol{i},\sigma,s} -t_{\perp}^{(s)}\left(\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,1}\hat{c}_{\boldsymbol{i},s,\sigma,2} + H.c.\right),\end{aligned} \tag{145}$$

where the additional index n labels the layers.

Honeycomb bilayer

The call:

```
Call Set_Default_hopping_parameters_Bilayer_honeycomb(Hopping_Matrix, T1_vec, T2_vec, Tperp_vec,
       Chem_vec, Phi_X_vec, Phi_Y_vec, Bulk, N_Phi_vec, N_FL, List, Invlist, Latt, Latt_unit)
```

defines the `Hopping_Matrix` for the bilayer honeycomb lattice:

$$\begin{aligned}\hat{H}_T = &\sum_{\boldsymbol{i},\sigma,s,n}\left(\sum_{\boldsymbol{\delta}=\{\boldsymbol{\delta}_1,\boldsymbol{\delta}_2,\boldsymbol{\delta}_3\}} -t_n^{(s)}\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,n} e^{\frac{2\pi i}{\Phi_0}\int_{i}^{\boldsymbol{i}+\boldsymbol{\delta}}\boldsymbol{A}^{(s)}(\boldsymbol{l})d\boldsymbol{l}}\hat{c}_{\boldsymbol{i}+\boldsymbol{\delta},s,\sigma,n} + H.c.\right) + \\ &\sum_{\boldsymbol{i},\sigma,s} -t_{\perp}^{(s)}\left(\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,1}\hat{c}_{\boldsymbol{i},s,\sigma,2} + \hat{c}^{\dagger}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma,1}\hat{c}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma,2} + H.c.\right) + \\ &\sum_{\boldsymbol{i},\sigma,s,n} -\mu^{(s)}\left(\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,n}\hat{c}_{\boldsymbol{i},s,\sigma,n} + \hat{c}^{\dagger}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma,n}\hat{c}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma,n}\right)\end{aligned} \tag{146}$$

Here, the additional index n labels the layer. $\boldsymbol{i}$ runs over the unit cells and $\boldsymbol{\delta} = \{\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \boldsymbol{\delta}_3\}$ over the three nearest neighbors.

N-leg ladder

The call:

```
Call Set_Default_hopping_parameters_n_lag_ladder(Hopping_Matrix, T_vec, Tperp_vec, Chem_vec,
       Phi_X_vec, Phi_Y_vec, Bulk, N_Phi_vec, N_FL, List, Invlist, Latt, Latt_unit)
```

defines the `Hopping_Matrix` for the the N-leg ladder lattice:

$$\begin{aligned}\hat{H}_T = &\sum_{\boldsymbol{i},\sigma,s}\sum_{n=1}^{\texttt{Norb}}\left(-t^{(s)}\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,n} e^{\frac{2\pi i}{\Phi_0}\int_{i}^{\boldsymbol{i}+\boldsymbol{a}_1}\boldsymbol{A}^{(s)}(\boldsymbol{l})d\boldsymbol{l}}\hat{c}_{\boldsymbol{i}+\boldsymbol{a}_1,s,\sigma,n} + H.c. - \mu^{(s)}\hat{c}^{\dagger}_{\boldsymbol{i},s,\sigma,n}\hat{c}_{\boldsymbol{i},s,\sigma,n}\right) + \\ &\sum_{\boldsymbol{i},\sigma,s}\sum_{n=1}^{\texttt{Norb}-1} -t_{\perp}^{(s)}\left(\hat{c}^{\dagger}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma,n} e^{\frac{2\pi i}{\Phi_0}\int_{(n-1)\boldsymbol{a}_2}^{(n)\boldsymbol{a}_2}\boldsymbol{A}^{(s)}(\boldsymbol{l})d\boldsymbol{l}}\hat{c}_{\boldsymbol{i}+\boldsymbol{\delta}_1,s,\sigma,n+1} + H.c.\right).\end{aligned} \tag{147}$$

Here, the additional index n defines the orbital. Note that this lattice has open boundary conditions in the $\boldsymbol{a}_2$ direction.

8.3. Predefined interaction vertices

In its most general form, an interaction Hamiltonian expressed in terms of sums of perfect squares can be written, as presented in Section 1, as a sum of M_V vertices:

$$\begin{aligned}\hat{\mathcal{H}}_V &= \sum_{k=1}^{M_V} U_k \left\{\sum_{\sigma=1}^{N_{\mathrm{col}}}\sum_{s=1}^{N_{\mathrm{fl}}}\left[\left(\sum_{x,y}^{N_{\mathrm{dim}}}\hat{c}^{\dagger}_{x\sigma s}V_{xy}^{(ks)}\hat{c}_{y\sigma s}\right) + \alpha_{ks}\right]\right\}^2 \equiv \sum_{k=1}^{M_V} U_k\left(\hat{V}^{(k)}\right)^2 \\ &\equiv \sum_{k=1}^{M_V}\hat{\mathcal{H}}_V^{(k)},\end{aligned} \tag{4}$$

which are encoded in one or more variables of type `Operator`, described in Sec. 5.1. We often use arrays of `Operator` type, which should be initialized by repeatedly calling the subroutine `Op_make`.

The module `Predefined_Int_mod.F90` implements some of the most common of such interaction vertices $\hat{\mathcal{H}}_V^{(k)}$, as detailed in the remaining of this section, where we drop the superscript (k) when unambiguous.

8.3.1. $SU(N)$ Hubbard interaction

The $SU(N)$ Hubbard interaction on a given site i is given by

$$\hat{\mathcal{H}}_{V,i} = +\frac{U}{N_{\text{col}}}\left[\sum_{\sigma=1}^{N_{\text{col}}}\left(\hat{c}^{\dagger}_{i\sigma}\hat{c}_{i\sigma} - 1/2\right)\right]^2 . \tag{148}$$

Assuming that no other term in the Hamiltonian breaks the $SU(N)$ color symmetry, then this interaction term conveniently corresponds to a single operator, obtained by calling, for each of the N_{dim} sites i:

```
Call Predefined_Int_U_SUN( OP, I, N_SUN, DTAU, U )
```

which defines:

```
Op%P(1)   = I
Op%O(1,1) = cmplx(1.d0,  0.d0, kind(0.D0))
Op%alpha  = cmplx(-0.5d0,0.d0, kind(0.D0))
Op%g      = SQRT(CMPLX(-DTAU*U/(DBLE(N_SUN)), 0.D0, kind(0.D0)))
Op%type   = 2
```

To relate to Eq. (4) we have, $V^{(is)}_{xy} = \delta_{x,y}\delta_{x,i}$, $\alpha_{is} = -\frac{1}{2}$ and $U_k = \frac{U}{N_{\text{col}}}$. Here the flavor index, s, plays no role.

8.3.2. M_z-Hubbard interaction

```
Call Predefined_Int_U_MZ( OP_up, Op_do, I, DTAU, U )
```

The M_z-Hubbard interaction is given by

$$\hat{\mathcal{H}}_V = -\frac{U}{2}\sum_i \left[\hat{c}^{\dagger}_{i\uparrow}\hat{c}_{i\uparrow} - \hat{c}^{\dagger}_{i\downarrow}\hat{c}_{i\downarrow}\right]^2 , \tag{149}$$

which corresponds to the general form of Eq. (4) by setting: $N_{\text{fl}} = 2$, $N_{\text{col}} \equiv$ `N_SUN` $= 1$, $M_V = N_{\text{unit-cell}}$, $U_k = \frac{U}{2}$, $V^{(i,s=1)}_{xy} = \delta_{x,y}\delta_{x,i}$, $V^{(i,s=2)}_{xy} = -\delta_{x,y}\delta_{x,i}$, and $\alpha_{is} = 0$; and which is defined in the subroutine `Predefined_Int_U_MZ` by two operators:

```
Op_up%P(1)   = I
Op_up%O(1,1) = cmplx(1.d0, 0.d0, kind(0.D0))
Op_up%alpha  = cmplx(0.d0, 0.d0, kind(0.D0))
Op_up%g      = SQRT(CMPLX(DTAU*U/2.d0, 0.D0, kind(0.D0)))
Op_up%type   = 2

Op_do%P(1)   = I
Op_do%O(1,1) = cmplx(1.d0, 0.d0, kind(0.D0))
Op_do%alpha  = cmplx(0.d0, 0.d0, kind(0.D0))
Op_do%g      = -SQRT(CMPLX(DTAU*U/2.d0, 0.D0, kind(0.D0)))
Op_do%type   = 2
```

8.3.3. $SU(N)$ V-interaction

```
Call Predefined_Int_V_SUN( OP, I, J, N_SUN, DTAU, V )
```

The interaction term of the generalized t-V model, given by

$$\hat{\mathcal{H}}_{V,i,j} = -\frac{V}{N_{\mathrm{col}}}\left[\sum_{\sigma=1}^{N_{\mathrm{col}}}\left(\hat{c}^{\dagger}_{i\sigma}\hat{c}_{j\sigma} + \hat{c}^{\dagger}_{j\sigma}\hat{c}_{i\sigma}\right)\right]^2, \tag{150}$$

is coded in the subroutine `Predefined_Int_V_SUN` by a single symmetric operator:

```
Op%P(1)    = I
Op%P(2)    = J
Op%O(1,2) = cmplx(1.d0 ,0.d0, kind(0.D0))
Op%O(2,1) = cmplx(1.d0 ,0.d0, kind(0.D0))
Op%g       = SQRT(CMPLX(DTAU*V/real(N_SUN,kind(0.d0)), 0.D0, kind(0.D0)))
Op%alpha   = cmplx(0.d0, 0.d0, kind(0.D0))
Op%type    = 2
```

8.3.4. Fermion-Ising coupling

```
Call Predefined_Int_Ising_SUN( OP, I, J, DTAU, XI )
```

The interaction between the Ising and a fermion degree of freedom, given by

$$\hat{\mathcal{H}}_{V,i,j} = \hat{Z}_{i,j}\xi\sum_{\sigma=1}^{N_{\mathrm{col}}}\left(\hat{c}^{\dagger}_{i\sigma}\hat{c}_{j\sigma} + \hat{c}^{\dagger}_{j\sigma}\hat{c}_{i\sigma}\right), \tag{151}$$

where ξ determines the coupling strength, is implemented in the subroutine `Predefined_Int_Ising_SUN`:

```
Op%P(1)    = I
Op%P(2)    = J
Op%O(1,2) = cmplx(1.d0 ,0.d0, kind(0.D0))
Op%O(2,1) = cmplx(1.d0 ,0.d0, kind(0.D0))
Op%g       = cmplx(-DTAU*XI,0.D0,kind(0.D0))
Op%alpha   = cmplx(0d0,0.d0, kind(0.D0))
Op%type    = 1
```

8.3.5. Long-Range Coulomb repulsion

```
Call Predefined_Int_LRC( OP, I, DTAU )
```

The Long-Range Coulomb (LRC) interaction can be written as

$$\hat{\mathcal{H}}_V = \frac{1}{N}\sum_{\boldsymbol{i},\boldsymbol{j}}\left(\hat{n}_{\boldsymbol{i}} - \frac{N}{2}\right)V_{\boldsymbol{i},\boldsymbol{j}}\left(\hat{n}_{\boldsymbol{j}} - \frac{N}{2}\right), \tag{152}$$

where

$$\hat{n}_{\boldsymbol{i}} = \sum_{\sigma=1}^{N}\hat{c}^{\dagger}_{\boldsymbol{i},\sigma}\hat{c}_{\boldsymbol{i},\sigma} \tag{153}$$

and $\boldsymbol{i}$ corresponds to a super-index labelling the unit cell and orbital.

The code uses the following HS decomposition:

$$e^{-\Delta\tau\hat{H}_{V,k}} = \int\prod_{\boldsymbol{i}}d\phi_{\boldsymbol{i}}e^{-\frac{N\Delta\tau}{4}\phi_{\boldsymbol{i}}V^{-1}_{\boldsymbol{i},\boldsymbol{j}}\phi_{\boldsymbol{j}} - \sum_{\boldsymbol{i}}i\Delta\tau\phi_{\boldsymbol{i}}\left(\hat{n}_{\boldsymbol{i}} - \frac{N}{2}\right)}. \tag{154}$$

The above holds only provided that the matrix V is positive definite and the implementation follows Ref. [26].

The LRC interaction is implemented in the subroutine `Predefined_Int_LRC`:

```
Op%P(1)    = I
Op%O(1,1) = cmplx(1.d0   ,0.d0, kind(0.D0))
Op%alpha   = cmplx(-0.5d0,0.d0, kind(0.D0))
Op%g       = cmplx(0.d0   ,DTAU, kind(0.D0))
Op%type    = 3
```

8.3.6. J_z-J_z interaction

```
Call Predefined_Int_Jz( OP_up, Op_do, I, J, DTAU, Jz )
```

Another predefined vertex is:

$$\hat{\mathcal{H}}_{V,i,j} = -\frac{|J_z|}{2}\left(S_i^z - \mathrm{sgn}\,|J_z|S_j^z\right)^2 = J_z S_i^z S_j^z - \frac{|J_z|}{2}(S_i^z)^2 - \frac{|J_z|}{2}(S_j^z)^2 \tag{155}$$

which, if particle fluctuations are frozen on the i and j sites, then $(S_i^z)^2 = 1/4$ and the interactions corresponds to a J_z-J_z ferro or antiferro coupling.

The implementation of the interaction in `Predefined_Int_Jz` defines two operators:

```
Op_up%P(1)    = I
Op_up%P(2)    = J
Op_up%O(1,1) = cmplx(1.d0,              0.d0, kind(0.D0))
Op_up%O(2,2) = cmplx(-Jz/Abs(Jz),       0.d0, kind(0.D0))
Op_up%alpha  = cmplx(0.d0,              0.d0, kind(0.D0))
Op_up%g      = SQRT(CMPLX(DTAU*Jz/8.d0, 0.d0, kind(0.D0)))
Op_up%type   = 2

Op_do%P(1)    = I
Op_do%P(2)    = J
Op_do%O(1,1) = cmplx(1.d0,               0.d0, kind(0.d0))
Op_do%O(2,2) = cmplx(-Jz/Abs(Jz),        0.d0, kind(0.d0))
Op_do%alpha  = cmplx(0.d0,               0.d0, kind(0.d0))
Op_do%g      = -SQRT(CMPLX(DTAU*Jz/8.d0, 0.d0, kind(0.d0)))
Op_do%type   = 2
```

8.4. Predefined observables

The types `Obser_Vec` and `Obser_Latt` described in Section 5.4 handles arrays of scalar observables and correlation functions with lattice symmetry respectively. The module `Predefined_Obs` provides a set of standard equal-time and time-displaced observables, as described bellow.

The predefined measurements methods take as input Green functions `GR`, `GT0`, `G0T`, `G00`, and `GTT`, defined in Sec. 7.6.2 and 7.6.3, as well as `N_SUN`, time slice `Ntau`, lattice information, and so on – see Table 19.

8.4.1. Equal-time $SU(N)$ spin-spin correlations

A measurement of $SU(N)$ spin-spin correlations can be obtained by:

```
Call Predefined_Obs_eq_SpinSUN_measure(Latt, Latt_unit, List,  GR, GRC, N_SUN, ZS, ZP,  Obs)
```

If `N_FL = 1` then this routine returns

$$\texttt{Obs}(\boldsymbol{i}-\boldsymbol{j}, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) = \frac{2N}{N^2-1}\sum_{a=1}^{N^2-1}\langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} T^a \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}}\ \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} T^a \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}}\rangle\rangle_C, \tag{156}$$

where T^a are the generators of $SU(N)$ satisfying the normalization conditions $\mathrm{Tr}[T^a T^b] = \delta_{a,b}/2$, $\mathrm{Tr}[T^a] = 0$, $\hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} = \left(\hat{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}},1}, \cdots, \hat{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}},N}\right)$ is an N-flavored spinor, $\boldsymbol{j}$ corresponds to the unit-cell index and $n_{\boldsymbol{j}}$ labels the orbital.

Using Wicks theorem, valid for a given configuration of fields, we obtain

$$\begin{aligned}\texttt{Obs} = \quad & \frac{2N}{N^2-1}\sum_{a=1}^{N^2-1}\sum_{\alpha,\beta,\gamma,\delta=1}^{N} T^a_{\alpha,\beta}T^a_{\gamma,\delta}\times \\ & \left(\langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\alpha}\hat{c}_{\boldsymbol{i},n_{\boldsymbol{i}},\beta}\rangle\rangle_C \langle\langle \hat{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}},\gamma}\hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}},\delta}\rangle\rangle_C + \langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\alpha}\hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}},\delta}\rangle\rangle_C \langle\langle \hat{c}_{\boldsymbol{i},n_{\boldsymbol{i}},\beta}\hat{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}},\gamma}\rangle\rangle_C\right)\end{aligned} \tag{157}$$

For this SU(N) symmetric code, the Green function is diagonal in the spin index and spin independent:

$$\langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\alpha}\hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}},\beta}\rangle\rangle_C = \delta_{\alpha,\beta}\langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}}\hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}}}\rangle\rangle_C. \tag{158}$$

Argument	Type	Role	Description
`Latt`	Lattice	Input	Lattice as a variable of type `Lattice`, see Sec. 5.3
`Latt_Unit`	Unit_cell	Input	Unit cell as a variable of type `Unit_cell`, see Sec. 5.3
`List(Ndim,2)`	Integer	Input	For every site index `I`, stores the corresponding lattice position, `List(I,1)`, and the (local) orbital index, `List(I,2)`
`NT`	Integer	Input	Imaginary time τ
`GR(Ndim,Ndim,N_FL)`	Complex	Input	Equal-time Green function `GR(i,j,s)` $= \langle c_{i,s} c^{\dagger}_{j,s}\rangle$
`GRC(Ndim,Ndim,N_FL)`	Complex	Input	`GRC(i,j,s)` $= \langle c^{\dagger}_{i,s} c_{j,s}\rangle = \delta_{i,j} -$ `GR(j,i,s)`
`GT0(Ndim,Ndim,N_FL)`	Complex	Input	Time-displaced Green function $\langle\langle \mathcal{T} \hat{c}_{i,s}(\tau) \hat{c}^{\dagger}_{j,s}(0)\rangle\rangle$
`G0T(Ndim,Ndim,N_FL)`	Complex	Input	Time-displaced Green function $\langle\langle \mathcal{T} \hat{c}_{i,s}(0) \hat{c}^{\dagger}_{j,s}(\tau)\rangle\rangle$
`G00(Ndim,Ndim,N_FL)`	Complex	Input	Time-displaced Green function $\langle\langle \mathcal{T} \hat{c}_{i,s}(0) \hat{c}^{\dagger}_{j,s}(0)\rangle\rangle$
`GTT(Ndim,Ndim,N_FL)`	Complex	Input	Time-displaced Green function $\langle\langle \mathcal{T} \hat{c}_{i,s}(\tau) \hat{c}^{\dagger}_{j,s}(\tau)\rangle\rangle$
`N_SUN`	Integer	Input	Number of fermion colors N_{col}
`ZS`	Complex	Input	`ZS` $= \mathrm{sign}(C)$, see Sec. 5.4
`ZP`	Complex	Input	`ZP` $= e^{-S(C)} / \operatorname{Re}\left[e^{-S(C)}\right]$, see Sec. 5.4
`Obs`	Obser_Latt	Output	One or more measurement result

Table 19: Arguments taken by the subroutines in the module `Predefined_Obs`. Note that a given method makes use of only a subset of this list, as specified in their calls described bellow. Note that the superindex $i = (\boldsymbol{i}, n_{\boldsymbol{i}})$ where $\boldsymbol{i}$ denotes the unit cell and $n_{\boldsymbol{i}}$ the orbital.

Hence,

$$\begin{aligned} \texttt{Obs} = & \quad \frac{2N}{N^2-1} \sum_{a=1}^{N^2-1} \left([\mathrm{Tr} T^a]^2 \langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} \hat{c}_{\boldsymbol{i},n_{\boldsymbol{i}}} \rangle\rangle_C \langle\langle \hat{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} \hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C + \mathrm{Tr}\left[T^a T^a\right] \langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} \hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \langle\langle \hat{c}_{\boldsymbol{i},n_{\boldsymbol{i}}} \hat{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \right) \\ = & \quad N \langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} \hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \langle\langle \hat{c}_{\boldsymbol{i},n_{\boldsymbol{i}}} \hat{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \end{aligned} \tag{159}$$

8.4.2. Equal-time spin correlations

A measurement of the equal-time spin correlations can be obtained by:

```
Call Predefined_Obs_eq_SpinMz_measure(Latt, Latt_unit, List,  GR, GRC, N_SUN, ZS, ZP,
                                      ObsZ, ObsXY, ObsXYZ)
```

If `N_FL=2` and `N_SUN=1`, then the routine returns:

$$\begin{aligned} \texttt{ObsZ}\,(\boldsymbol{i}-\boldsymbol{j}, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) &= 4\langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} S^z \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}} \; \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^z \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C - 4\langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} S^z \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}} \rangle\rangle_C \langle\langle \, \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^z \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \\ \texttt{ObsXY}\,(\boldsymbol{i}-\boldsymbol{j}, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) &= 2\left(\langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} S^x \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}} \; \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^x \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C + \langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} S^y \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}} \; \boldsymbol{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^y \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \right) \\ \texttt{ObsXYZ} &= \frac{2 \cdot \texttt{ObsXY} + \texttt{ObsZ}}{3}. \end{aligned} \tag{160}$$

Here $\hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} = \left(\hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\uparrow}, \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\downarrow} \right)$ is a two component spinor and $\boldsymbol{S} = \frac{1}{2}\boldsymbol{\sigma}$ with

$$\boldsymbol{\sigma} = \left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \right) \tag{161}$$

the Pauli spin matrices. Add this to the code since you are not assuming SU(2) spin symmetry.

8.4.3. Equal-time Green function

A measurement of the equal-time Green function can be obtained by:

```
Call Predefined_Obs_eq_Green_measure(Latt, Latt_unit, List,  GR, GRC, N_SUN, ZS, ZP,  Obs)
```

Which returns:

$$\texttt{Obs}(\boldsymbol{i}-\boldsymbol{j}, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) = \sum_{\sigma=1}^{N_{\text{col}}} \sum_{s=1}^{N_{\text{fl}}} \langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\sigma,s} \hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}},\sigma,s} \rangle. \tag{162}$$

8.4.4. Equal-time density-density correlations

A measurement of equal-time density-density correlations can be obtained by:

```
Call Predefined_Obs_eq_Den_measure(Latt, Latt_unit, List,  GR, GRC, N_SUN, ZS, ZP,  Obs)
```

Which returns:

$$\texttt{Obs}(\boldsymbol{i}-\boldsymbol{j}, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) = \langle\langle \hat{N}_{\boldsymbol{i},n_{\boldsymbol{i}}} \hat{N}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle - \langle \hat{N}_{\boldsymbol{i},n_{\boldsymbol{i}}} \rangle \langle \hat{N}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C, \tag{163}$$

where

$$\hat{N}_{\boldsymbol{i},n_{\boldsymbol{i}}} = \sum_{\sigma=1}^{N_{\text{col}}} \sum_{s=1}^{N_{\text{fl}}} \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\sigma,s} \hat{c}_{\boldsymbol{i},n_{\boldsymbol{i}},\sigma,s}. \tag{164}$$

8.4.5. Time-displaced Green function

A measurement of the time-displaced Green function can be obtained by:

```
Call Predefined_Obs_tau_Green_measure(Latt, Latt_unit, List,  NT, GT0, G0T, G00, GTT,
                                      N_SUN, ZS, ZP,  Obs)
```

Which returns:

$$\texttt{Obs}(\boldsymbol{i}-\boldsymbol{j}, \tau, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) = \sum_{\sigma=1}^{N_{\text{col}}} \sum_{s=1}^{N_{\text{fl}}} \langle\langle \hat{c}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}},\sigma,s}(\tau) \hat{c}_{\boldsymbol{j},n_{\boldsymbol{j}},\sigma,s} \rangle\rangle_C \tag{165}$$

8.4.6. Time-displaced $SU(N)$ spin-spin correlations

A measurement of time-displaced spin-spin correlations for $SU(N)$ models ($N_{\text{fl}} = 1$) can be obtained by:

```
Call Predefined_Obs_tau_SpinSUN_measure(Latt, Latt_unit, List,  NT, GT0, G0T, G00, GTT,
                                        N_SUN, ZS, ZP,  Obs)
```

$$\texttt{Obs}(\boldsymbol{i}-\boldsymbol{j}, \tau, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) = \frac{2N}{N^2-1} \sum_{a=1}^{N^2-1} \langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau) T^a \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau)\ \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} T^a \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \tag{166}$$

where T^a are the generators of $SU(N)$ (see Sec. 8.4.1 for more details).

8.4.7. Time-displaced spin correlations

A measurement of time-displaced spin-spin correlations for Mz models ($N_{\text{fl}} = 2, N_{\text{col}} = 1$) is returned by:

```
Call Predefined_Obs_tau_SpinMz_measure(Latt, Latt_unit, List,  NT, GT0, G0T, G00, GTT,
                                       N_SUN, ZS, ZP,  ObsZ, ObsXY, ObsXYZ)
```

Which calculates the following observables:

$$\begin{aligned}
\texttt{ObsZ}(\boldsymbol{i}-\boldsymbol{j}, \tau, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) &= 4\langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau) S^z \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau)\ \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^z \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C - 4\langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}} S^z \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}} \rangle\rangle_C \langle\langle\ \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^z \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \\
\texttt{ObsXY}(\boldsymbol{i}-\boldsymbol{j}, \tau, n_{\boldsymbol{i}}, n_{\boldsymbol{j}}) &= 2\left(\langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau) S^x \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau)\ \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^x \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C + \langle\langle \hat{\boldsymbol{c}}^{\dagger}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau) S^y \hat{\boldsymbol{c}}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau)\ \boldsymbol{c}^{\dagger}_{\boldsymbol{j},n_{\boldsymbol{j}}} S^y \hat{\boldsymbol{c}}_{\boldsymbol{j},n_{\boldsymbol{j}}} \rangle\rangle_C \right) \\
\texttt{ObsXYZ} &= \frac{2 \cdot \texttt{ObsXY} + \texttt{ObsZ}}{3}.
\end{aligned} \tag{167}$$

8.4.8. Time-displaced density-density correlations

A measurement of time-displaced density-density correlations for general $SU(N)$ models is given by:

```
Call Predefined_Obs_tau_Den_measure(Latt, Latt_unit, List,  NT, GT0, G0T, G00, GTT,
                                    N_SUN, ZS, ZP,  Obs)
```

Which returns:

$$\texttt{Obs}(\boldsymbol{i}-\boldsymbol{j},\tau,n_{\boldsymbol{i}},n_{\boldsymbol{j}}) = \langle\langle \hat{N}_{\boldsymbol{i},n_{\boldsymbol{i}}}(\tau)\hat{N}_{\boldsymbol{j},n_{\boldsymbol{j}}}\rangle - \langle \hat{N}_{\boldsymbol{i},n_{\boldsymbol{i}}}\rangle\langle \hat{N}_{\boldsymbol{j},n_{\boldsymbol{j}}}\rangle\rangle_C. \tag{168}$$

The density operator is defined in Eq. 164.

8.5. Predefined trial wave functions

When using the projective algorithm (see Sec. 3), trial wave functions must be specified. These are stored in variables of the `WaveFunction` type (Sec. 5.5). The ALF package provides a set of predefined trial wave functions $|\Psi_{T,L/R}\rangle$=`WF_L/R`, returned by the call:

```
Call Predefined_TrialWaveFunction(Lattice_type, Ndim, List, Invlist, Latt, Latt_unit, N_part,
                                  N_FL, WF_L, WF_R)
```

Twisted boundary conditions (`Phi_X_vec=0.01`) are implemented for some lattices so as to generate a non-degenerate trial wave function. Here the marker `"_vec"` indicates the variable may assume different values depending on the flavor (e.g., spin up and down). Currently predefined trial wave functions are flavor independent.

The predefined trial wave functions correspond to the solution of the non-interacting tight binding Hamiltonian on each of the predefined lattices. These solutions are the ground states of the predefined hopping matrices (Sec. **??**) with default parameters, for each lattice, as follows.

8.5.1. Square

Parameter values for the predefined trial wave function on the square lattice:

```
Checkerboard  = .false.
Symm          = .false.
Bulk          = .false.
N_Phi_vec     = 0
Phi_X_vec     = 0.01d0
Phi_Y_vec     = 0.d0
Ham_T_vec     = 1.d0
Ham_Chem_vec  = 0.d0
Dtau          = 1.d0
```

8.5.2. Honeycomb

The twisted boundary condition for the square lattice lifts the degeneracy present at half-band filling, but breaks time reversal symmetry as well as the C_4 lattice symmetry. If time reversal symmetry is required to avoid the negative sign problem (that would be the case for the attractive Hubbard model at finite doping), then this choice of the trial wave function will introduce a negative sign. One should then use the trial wave function presented in Sec. 7.5. For the Honeycomb case, the trial wave function we choose is the ground state of the tight binding model with small next-next-next nearest hopping matrix element t' [75].. This breaks the C_3 symmetry and shifts the Dirac cone away from the zone boundary. Time reversal symmetry is however not broken. Alternatively, one could include a small Kekule mass term. As shown in Sec. 3.3 both choices of trial wave functions produce good results.

8.5.3. N-leg ladder

Parameter values for the predefined trial wave function on the N-leg ladder lattice:

```
Checkerboard  = .false.
Symm          = .false.
Bulk          = .false.
N_Phi_vec     = 0
Phi_X_vec     = 0.01d0
Phi_Y_vec     = 0.d0
Ham_T_vec     = 1.d0
Ham_Tperp_vec = 1.d0
Ham_Chem_vec  = 0.d0
Dtau          = 1.d0
```

8.5.4. Bilayer square

Parameter values for the predefined trial wave function on the bilayer square lattice:

```
Checkerboard  = .false.
Symm          = .false.
Bulk          = .false.
N_Phi_vec     = 0
Phi_X_vec     = 0.d0
Phi_Y_vec     = 0.d0
Ham_T_vec     = 1.d0
Ham_T2_vec    = 0.d0
Ham_Tperp_vec = 1.d0
Ham_Chem_vec  = 0.d0
Dtau          = 1.d0
```

8.5.5. Bilayer honeycomb

Parameter values for the predefined trial wave function on the bilayer honeycomb lattice:

```

Checkerboard  = .false.
Symm          = .false.
Bulk          = .false.
N_Phi_vec     = 0
Phi_X_vec     = 0.d0
Phi_Y_vec     = 0.d0
Ham_T_vec     = 1.d0
Ham_T2_vec    = 0.d0
Ham_Tperp_vec = 1.d0
Ham_Chem_vec  = 0.d0
Dtau          = 1.d0
```

9. Model Classes

The ALF library comes with five model classes: (i) SU(N) Hubbard models, (ii) O(2N) t-V models, (iii) Kondo models, (iv) Models with long ranged coulomb, and (v) Generic Z_2 lattice gauge theories coupled to Z_2 matter and fermions. Below we detail the functioning of these classes.

9.1. SU(N) Hubbard models `Hamiltonian_Hubbard_mod.F90`

The parameter space for this model class reads:

```
&VAR_Hubbard              !! Variables for the Hubbard class
Mz        = .T.            ! If true , sets the M_z-Hubbard model: Nf=2, N_SUN has to be even
    .
                           ! HS field couples to the z-component of magnetization
ham_T     = 1.d0           ! Hopping parameter
```

```
ham_chem  = 0.d0          ! Chemical potential
ham_U     = 4.d0          ! Hubbard interaction
ham_T2    = 1.d0          ! For bilayer systems
ham_U2    = 4.d0          ! For bilayer systems
ham_Tperp = 1.d0          ! For bilayer systems
/
```

In the above `ham_T` and `ham_T2` correspond to the hopping in the first and second layers respectively and `ham_Tperp` is to the interlayer hopping. The Hubbard U term has an orbital index, `ham_U` for the first and `ham_U2` for the second layers. Finally `ham_chem` corresponds to the chemical potential. If the flag `Mz` is set to `.False.`, then the code will simulate the following SU(N) symmetry Hubbard model.

$$\hat{H} = \sum_{(\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')} \sum_{\sigma=1}^{N} T_{(\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')} \hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} e^{\frac{2\pi i}{\Phi_0}\int_{\boldsymbol{i}+\boldsymbol{\delta}}^{\boldsymbol{j}+\boldsymbol{\delta}'} \boldsymbol{A}(\boldsymbol{l}) d\boldsymbol{l}} \hat{c}_{(\boldsymbol{j},\boldsymbol{\delta}'),\sigma} + \sum_{\boldsymbol{i}} \sum_{\delta} \frac{U_\delta}{N} \left(\sum_{\sigma=1}^{N} \left[\hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} \hat{c}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} - 1/2 \right] \right)^2 - \mu \sum_{(\boldsymbol{i},\boldsymbol{\delta})} \sum_{\sigma=1}^{N} \hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} \hat{c}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} \tag{169}$$

The generic hopping is taken fron Eq. 133 with appropriate boundary conditions given by Eq. 134. $\boldsymbol{i}$ runs over the unit cells, $\boldsymbol{\delta}$ over the orbitals in each unit cell and σ from $1 \cdots N$ and encodes the SU(N) symmetry. Note that N corresponds to `N_SUN` in the code. The flavor index is set to unity such that it does not appear in the Hamiltonian. μ corresponds to the chemical potential and is relevant only for the finite temperature code.

If the variable `Mz` is set to `.True.`, then the code will require `N_SUN` to be even and will simulate the following Hamiltonian.

$$\hat{H} = \sum_{(\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')} \sum_{\sigma=1}^{N/2} \sum_{s=1,2} T_{(\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')} \hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma,s} e^{\frac{2\pi i}{\Phi_0}\int_{\boldsymbol{i}+\boldsymbol{\delta}}^{\boldsymbol{j}+\boldsymbol{\delta}'} \boldsymbol{A}(\boldsymbol{l}) d\boldsymbol{l}} \hat{c}_{(\boldsymbol{j},\boldsymbol{\delta}'),\sigma,s} - \sum_{\boldsymbol{i}} \sum_{\delta} \frac{U_\delta}{N} \left(\sum_{\sigma=1}^{N/2} \left[\hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma,2} \hat{c}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma,2} - \hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma,1} \hat{c}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma,1} \right] \right)^2 - \mu \sum_{(\boldsymbol{i},\boldsymbol{\delta})} \sum_{\sigma=1}^{N/2} \sum_{s=1,2} \hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma,s} \hat{c}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma,s}. \tag{170}$$

In this case, the flavor index `N_FL` takes the value 2. Cleary at $N = 2$, both modes correspond to the Hubbard model. For N even and $N > 2$ the models differ. In particular in the latter Hamiltonian the U(N) symmetry is broken down to U(N/2) ⊗U(N/2).

Since this model class works for all predefined lattices (see Fig. 5) it includes the SU(N) periodic Anderson model on the square and Honeycomb lattices. Finally, we note that the executable for this class is given by `Hubbard.out`.

As an example, we can consider the periodic Anderson model. Here we choose the `Bilayer_square` lattice `Ham_U` $=$ `Ham_T2` $= 0$, `Ham_U2`$= U_f$, `Ham_tperp`$= V$ and `Ham_T`$= 1$. The pyALF based python script `Hubbard_PAM.py` produces the data shown in Fig. 7 for the L=8 lattice.

9.2. O(2N) t-V models `tV_mod.F90`

The parameter space for this model class reads:

```
&VAR_tV                  !! Variables for the t-V class
ham_T     = 1.d0          ! Hopping parameter
ham_chem  = 0.d0          ! Chemical potential
ham_V     = 0.5d0         ! interaction strength
ham_T2    = 1.d0          ! For bilayer systems
ham_V2    = 0.5d0         ! For bilayer systems
ham_Tperp = 1.d0          ! For bilayer systems
ham_Vperp = 0.5d0         ! For bilayer systems
/
```

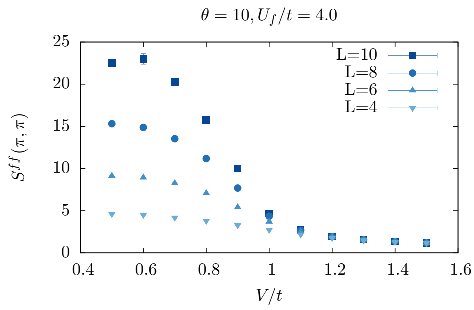

Figure 7: The periodic Anderson model. Here we plot the equal time spin structure factor of the f-electrons at $\boldsymbol{q} = (\pi, \pi)$. This quantity is found in the file `SpinZ_eqJK`. The pyALF based python script `Hubbard_PAM.py` produces the data shown for the L=8 lattice. One sees that for this considered value of U_f/t the competition between the RKKY interaction and Kondo screening drives the system through a magnetic order-disorder transition at $V_c/t \simeq 1$ [87].

In the above `ham_T` and `ham_T2` and `ham_Tperp` correspond to the hopping in the first and second layers respectively and `ham_Tperp` is to the interlayer hopping. The interaction term has an orbital index, `ham_V` for the first and `ham_V2` for the second layers, and `ham_Vperp` for interlayer coupling. Note the we use the same sign conventions here for both the hopping parameters and the interaction strength. This implies a relative minus sign between here and the U_δ interaction strength of the Hubbard model (see Sec. 9.1). Finally `ham_chem` corresponds to the chemical potential. Let us define the operator

$$\hat{b}_{\langle (\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')\rangle} = \sum_{\sigma=1}^{N} \hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} e^{\frac{2\pi i}{\Phi_0}\int_{i+\delta}^{j+\delta'} \boldsymbol{A}(\boldsymbol{l}) d\boldsymbol{l}} \hat{c}_{(\boldsymbol{j},\boldsymbol{\delta}'),\sigma} + \text{H.c.} \tag{171}$$

The model is then defined as follows:

$$\hat{H} = \sum_{\langle (\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')\rangle} T_{(\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')} \hat{b}_{\langle (\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')\rangle} + \sum_{\langle (\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')\rangle} \frac{V_{(\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')}}{N} \left(\hat{b}_{\langle (\boldsymbol{i},\boldsymbol{\delta}),(\boldsymbol{j},\boldsymbol{\delta}')\rangle}\right)^2$$
$$- \mu \sum_{(\boldsymbol{i},\boldsymbol{\delta})} \sum_{\sigma=1}^{N} \hat{c}^{\dagger}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} \hat{c}_{(\boldsymbol{i},\boldsymbol{\delta}),\sigma} . \tag{172}$$

The generic hopping is taken from Eq. 133 with appropriate boundary conditions given by Eq. 134. $\boldsymbol{i}$ runs over the unit cells, $\boldsymbol{\delta}$ over the orbitals in each unit cell and σ from $1 \cdots N$ and encodes the SU(N) symmetry. Note that N corresponds to `N_SUN` in the code. The flavor index is set to unity such that it does not appear in the Hamiltonian. μ corresponds to the chemical potential and is relevant only for the finite temperature code. An example showing how to run this model class can be found in the pyALF based Jupyter notebook `tV_model.ipynb`.

For concreteness sake, the Hamiltonian of the $t-V$ model of $SU(N)$ fermions on the square lattice reads,

$$\hat{H} = -t \sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \hat{b}_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} - \frac{V}{N} \sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \left(\hat{b}_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\right)^2 - \mu \sum_{\boldsymbol{i}} \sum_{\sigma=1}^{N} \hat{c}^{\dagger}_{\boldsymbol{i},\sigma} \hat{c}_{\boldsymbol{i},\sigma} , \tag{173}$$

and can be simulated by setting `ham_T` $= t$, `ham_V` $= V$, and `ham_chem` $= \mu$. Al half-band filling $\mu = 0$, the sign problem is absent for $V > 0$ and for all values of N. For even values of N no sign problem occurs for $V > 0$ and arbitrary chemical potentials.

Note that in the absence of orbital magnetic fields, the model has an O(2N) symmetry. This can be seen by writing the model in a Majorana basis (see e.g. Ref. [17]).

9.3. SU(N) Kondo lattice models `Kondo_mod.F90`

The Kondo lattice model we consider reads is an SU(N) generalization of the SU(2) Kondo-model discussed in [19, 18]. Here we follow the work of Ref. [88]. Let T^a be the $N^2 - 1$ generators of SU(N) that satisfy the normalization condition:

$$\text{Tr}\left[T^a T^b\right] = \frac{1}{2}\delta_{a,b}. \tag{174}$$

For the SU(2) case T^a corresponds to the $T = \frac{1}{2}\boldsymbol{\sigma}$ with $\boldsymbol{\sigma}$ a vector of the three Pauli spin matrices. The Hamiltonian is defined on bilayer square or honeycomb lattices, with hopping restricted to the first layer (i.e conduction orbitals $\boldsymbol{c}_i^\dagger$) and spins, f-orbitals, on the second layer.

$$\hat{H} = -t\sum_{\langle i,j\rangle}\sum_{\sigma=1}^{N}\left(\hat{c}^\dagger_{i,\sigma}e^{\frac{2\pi i}{\Phi_0}\int_i^j \boldsymbol{A}\cdot d\boldsymbol{l}}\hat{c}_{j,\sigma} + H.c.\right) - \mu\sum_{i,\sigma}\hat{c}^\dagger_{i,\sigma}\hat{c}_{i,\sigma} + \frac{U_c}{N}\sum_i\left(\hat{n}^c_i - \frac{N}{2}\right)^2 + \frac{2J}{N}\sum_{i,a=1}^{N^2-1}\hat{T}^{a,c}_i\hat{T}^{a,f}_i. \tag{175}$$

In the above, i is a super-index accounting for the unit cell and orbital,

$$\hat{T}^{a,c}_i = \sum_{\sigma,\sigma'=1}^{N}\hat{c}^\dagger_{i,\sigma}T^a_{\sigma,\sigma'}\hat{c}_{i,\sigma'},\;\; \hat{T}^{a,f}_i = \sum_{\sigma,\sigma'=1}^{N}\hat{f}^\dagger_{i,\sigma}T^a_{\sigma,\sigma'}\hat{f}_{i,\sigma'},\; \text{and } \hat{n}^c_i = \sum_{\sigma=1}^{N}\hat{c}^\dagger_{i,\sigma}\hat{c}_{i,\sigma} \tag{176}$$

Finally, the constraint,

$$\sum_{\sigma=1}^{N}\hat{f}^\dagger_{i,\sigma}\hat{f}_{i,\sigma} \equiv \hat{n}^f_i = \frac{N}{2} \tag{177}$$

holds.

Some rewriting has to be carried out so as to implement the model. First, we use the relation:

$$\sum_a T^a_{\alpha,\beta}T^a_{\alpha',\beta'} = \frac{1}{2}\left(\delta_{\alpha,\beta'}\delta_{\alpha',\beta} - \frac{1}{N}\delta_{\alpha,\beta}\delta_{\alpha',\beta'}\right),$$

to show that in the unconstrained Hilbert space,

$$\frac{2J}{N}\sum_{a=1}^{N^2-1}\hat{T}^{a,c}_i\hat{T}^{a,f}_i = -\frac{J}{2N}\sum_i\left(\hat{D}^\dagger_i\hat{D}_i + \hat{D}_i\hat{D}^\dagger_i\right) + \frac{J}{N}\left(\frac{\hat{n}^c_i}{2} + \frac{\hat{n}^f_i}{2} - \frac{\hat{n}^c_i\hat{n}^f_i}{N}\right)$$

with

$$\hat{D}^\dagger_i = \sum_{\sigma=1}^{N}\hat{c}^\dagger_{i,\sigma}\hat{f}_{i,\sigma}.$$

In the constrained Hilbert space, $\hat{n}^f_i = N/2$, the above gives:

$$\frac{2J}{N}\sum_{a=1}^{N^2-1}\hat{T}^{a,c}_i\hat{T}^{a,f}_i = -\frac{J}{4N}\left[\left(\hat{D}^\dagger_i + \hat{D}_i\right)^2 + \left(i\hat{D}^\dagger_i - i\hat{D}_i\right)^2\right] + \frac{J}{4}. \tag{178}$$

The prefect square form complies with the standards of the ALF. We still have to impose the constraint. To do so, we work in the unconstrained Hilbert and add a Hubbard U-term on the f-orbitals. With this addition, the Hamiltonian we simulate reads:

$$\begin{aligned}\hat{H}_{QMC} =& -t\sum_{\langle i,j\rangle}\sum_{\sigma=1}^{N}\left(\hat{c}^\dagger_{i,\sigma}e^{\frac{2\pi i}{\Phi_0}\int_i^j \boldsymbol{A}\cdot d\boldsymbol{l}}\hat{c}_{j,\sigma} + H.c.\right) - \mu\sum_{i,\sigma}\hat{c}^\dagger_{i,\sigma}\hat{c}_{i,\sigma} + \frac{U_c}{N}\sum_i\left(\hat{n}^c_i - \frac{N}{2}\right)^2 \\ &- \frac{J}{4N}\left[\left(\hat{D}^\dagger_i + \hat{D}_i\right)^2 + \left(i\hat{D}^\dagger_i - i\hat{D}_i\right)^2\right] + \frac{U_f}{N}\sum_i\left(\hat{n}^f_i - \frac{N}{2}\right)^2.\end{aligned} \tag{179}$$

The key point for the efficiency of the code, is to see that

$$\left[\hat{H}_{QMC}, \left(\hat{n}^f_i - \frac{N}{2}\right)^2\right] = 0 \tag{180}$$

such that the constraint is implemented efficiently. In fact, for the finite temperature code at inverse temperature β, the unphysical Hilbert space is suppressed by a factor $e^{-\beta U_f/N}$.

The SU(2) case

The SU(2) case is special and allows for a more efficient implementation than mentioned above. The key point is that for the SU(2) case, the Hubbard term is related to the fermion parity,

$$\left(\hat{n}^f_i - 1\right)^2 = \frac{(-1)^{\hat{n}^f_i} + 1}{2} \tag{181}$$

such that we can omit the *current*-term $\left(i\hat{D}^\dagger_i - i\hat{D}_i\right)^2$ without violating Eq. 180. As in Refs. [18, 19, 89]. the Hamiltonian that one will simulate reads:

$$\begin{aligned}\hat{\mathcal{H}} = & \underbrace{-t\sum_{\langle i,j\rangle,\sigma}\left(\hat{c}^\dagger_{i,\sigma}e^{\frac{2\pi i}{\Phi_0}\int_i^j \boldsymbol{A}\cdot \boldsymbol{dl}}\hat{c}_{j,\sigma} + \text{H.c.}\right) + \frac{U_c}{2}\sum_i (\hat{n}^c_i - 1)^2}_{\equiv \hat{\mathcal{H}}_{tU_c}} \\ & - \frac{J}{4}\sum_i\left(\sum_\sigma \hat{c}^\dagger_{i,\sigma}\hat{f}_{i,\sigma} + \hat{f}^\dagger_{i,\sigma}\hat{c}_{i,\sigma}\right)^2 + \underbrace{\frac{U_f}{2}\sum_i\left(\hat{n}^f_i - 1\right)^2}_{\equiv \hat{\mathcal{H}}_{U_f}}.\end{aligned} \tag{182}$$

The relation to the Kondo lattice model follows from expanding the square of the hybridization to obtain:

$$\hat{\mathcal{H}} = \hat{\mathcal{H}}_{tU_c} + J\sum_{\boldsymbol{i}}\left(\hat{\boldsymbol{S}}^c_{\boldsymbol{i}} \cdot \hat{\boldsymbol{S}}^f_{\boldsymbol{i}} + \hat{\eta}^{z,c}_{\boldsymbol{i}} \cdot \hat{\eta}^{z,f}_{\boldsymbol{i}} - \hat{\eta}^{x,c}_{\boldsymbol{i}} \cdot \hat{\eta}^{x,f}_{\boldsymbol{i}} - \hat{\eta}^{y,c}_{\boldsymbol{i}} \cdot \hat{\eta}^{y,f}_{\boldsymbol{i}}\right) + \hat{\mathcal{H}}_{U_f}. \tag{183}$$

where the η-operators relate to the spin-operators via a particle-hole transformation in one spin sector:

$$\hat{\eta}^\alpha_{\boldsymbol{i}} = \hat{P}^{-1}\hat{S}^\alpha_{\boldsymbol{i}}\hat{P} \;\text{ with }\; \hat{P}^{-1}\hat{c}_{\boldsymbol{i},\uparrow}\hat{P} = (-1)^{i_x+i_y}\hat{c}^\dagger_{\boldsymbol{i},\uparrow} \;\text{ and }\; \hat{P}^{-1}\hat{c}_{\boldsymbol{i},\downarrow}\hat{P} = \hat{c}_{\boldsymbol{i},\downarrow} \tag{184}$$

Since the $\hat{\eta}^f$- and $\hat{S}^f$-operators do not alter the parity $[(-1)^{\hat{n}^f_i}\,]$ of the f-sites,

$$\left[\hat{\mathcal{H}}, \hat{\mathcal{H}}_{U_f}\right] = 0. \tag{185}$$

Thereby, and for positive values of U , doubly occupied or empty f-sites – corresponding to even parity sites – are suppressed by a Boltzmann factor $e^{-\beta U_f/2}$ in comparison to odd parity sites. Choosing βU_f adequately essentially allows to restrict the Hilbert space to odd parity f-sites. In this Hilbert space $\hat{\eta}^{x,f} = \hat{\eta}^{y,f} = \hat{\eta}^{z,f} = 0$ such that the Hamiltonian (182) reduces to the Kondo lattice model.

QMC implementation

The name space for this model class reads:

```
&VAR_Kondo                  !! Variables for the Kondo  class
ham_T      = 1.d0            ! Hopping parameter
ham_chem   = 0.d0            ! Chemical potential
ham_Uc     = 0.d0            ! Hubbard interaction  on  c-orbitals Uc
ham_Uf     = 2.d0            ! Hubbard interaction  on  f-orbials  Uf
ham_JK     = 2.d0            ! Kondo Coupling  J
/
```

Aside from the usual observables we have included the scalar observable `Constraint_scal` that measures

$$\left\langle \sum_i \left(\hat{n}^f_i - \frac{N}{2}\right)^2 \right\rangle \tag{186}$$

U_f has to be chosen large enough such that the above quantity vanishes within statistical uncertainty. For the square lattice, Fig. 8 plots the aforementioned quantity as a function of U_f for the SU(2) model. As apparent $\left\langle \sum_i \left(\hat{n}^f_i - \frac{N}{2}\right)^2 \right\rangle \propto e^{-\beta U_f/2}$.

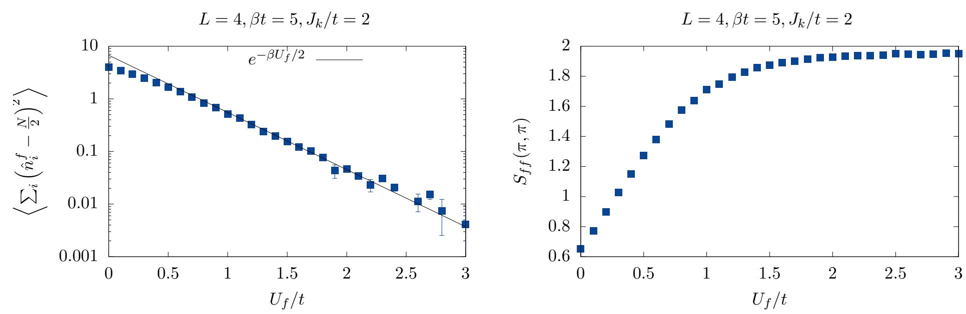

Figure 8: Left: Suppression of charge fluctuations of the f-orbitals as a function of U_f. Right: When charge fluctuations on the f-orbitals vanish, quantities such as the Fourier transform of the f spin-spin correlations at $\boldsymbol{q} = (\pi, \pi)$ converge to their KLM value. Typically, for the SU(2) case $\beta U_f > 10$ suffices to reach converges results. The pyALF script used to produce the data of the plot can be found in `Kondo.py`

9.4. Models with long range Coulomb interactions `LRC_mod.F90`

The model we consider here is defined for `N_FL=1`, arbitrary values of `N_SUN` and supports all the predefined lattices. It reads:

$$\hat{H} = \sum_{i,j} \sum_{\sigma=1}^{N} T_{i,j} \hat{c}^{\dagger}_{i,\sigma} e^{\frac{2\pi i}{\Phi_0} \int_i^j \boldsymbol{A}(\boldsymbol{l}) d\boldsymbol{l}} \hat{c}_{j,\sigma} + \frac{1}{N} \sum_{i,j} \left(\hat{n}_i - \frac{N}{2} \right) V_{i,j} \left(\hat{n}_j - \frac{N}{2} \right) - \mu \sum_i \hat{n}_i \tag{187}$$

In the above, $i = (\boldsymbol{i}, \boldsymbol{\delta}_i)$ and $j = (\boldsymbol{j}, \boldsymbol{\delta}_j)$ are super-indices encoding the unit-cell and orbital and $\hat{n}_i = \sum_{\sigma=1}^{N} \hat{c}^{\dagger}_{i,\sigma} \hat{c}_{i,\sigma}$ For simplicity, the interaction is specified by two parameters, U and α that monitor the strength of the onsite interaction and the magnitude of the Coulomb tail respectively.

$$V_{i,j} \equiv V(\boldsymbol{i} + \boldsymbol{\delta}_i, \boldsymbol{j} + \boldsymbol{\delta}_j) = U \begin{cases} 1 & \text{if } i = j \\ \frac{\alpha \, d_{\text{min}}}{||\boldsymbol{i} - \boldsymbol{j} + \boldsymbol{\delta}_i - \boldsymbol{\delta}_j||} & \text{otherwise} \end{cases} \tag{188}$$

Here d_{min} is the minimal distance between two orbitals. On a torus, some care has be taken in defining the distance. Let the lattice size be given by the vectors $\boldsymbol{L}_1$ and $\boldsymbol{L}_2$ (see Sec. 8.1). Then

$$||\boldsymbol{i}|| = \min_{n_1, n_2 \in \mathbb{Z}} |\boldsymbol{i} - n_1 \boldsymbol{L}_1 - n_2 \boldsymbol{L}_2| \tag{189}$$

The implementation follows Ref. [26] but now supports various lattice geometries. We use the following HS decomposition:

$$e^{-\Delta\tau \hat{H}_V} \propto \int \prod_i d\phi_i e^{-\frac{N\Delta\tau}{4} \sum_{i,j} \phi_i V^{-1}_{i,j} \phi_j - \sum_i i \Delta\tau \phi_i \left(\hat{n}_i - \frac{N}{2} \right)} \tag{190}$$

where ϕ_i is a real variable, V is symmetric, and importantly has to be positive definite for the Gaussian integration to be defined. The partition function reads:

$$Z \propto \int \prod_i d\phi_{i,\tau} \overbrace{e^{-\frac{N\Delta\tau}{4} \sum_{i,j} \phi_{i,\tau} V^{-1}_{i,j} \phi_{j,\tau}}}^{W_B(\phi)} \underbrace{\text{Tr} \left[\prod_\tau e^{-\Delta\tau \hat{H}_T} e^{-\sum_i i \Delta\tau \phi_{i,\tau} \left(\hat{n}_i - \frac{N}{2} \right)} \right]}_{W_F(\phi)}. \tag{191}$$

such that the weight splits into a bosonic and fermionic parts.

For the update, it is convenient to work in a basis where V is diagonal:

$$\text{Diag} \left(\lambda_1, \cdots, \lambda_{\texttt{Ndim}} \right) = O^T V O \tag{192}$$

with $O^T O = 1$ and define:

$$\eta_{i,\tau} = \sum_j O^T_{i,j} \phi_{j,\tau}. \tag{193}$$

On a given time slice, τ_u, we propose a new field configuration with the probability:

$$T^0(\eta \rightarrow \eta') = \begin{cases} \prod_i \left[P P_B(\eta'_{i,\tau_u}) + (1-P)\delta(\eta_{i,\tau_u} - \eta'_{i,\tau_u}) \right] & \text{for } \tau = \tau_u \\ \delta(\eta_{i,\tau} - \eta'_{i,\tau}) & \text{for } \tau \neq \tau_u \end{cases} \tag{194}$$

where

$$P_B(\eta_{i,\tau}) \propto e^{-\frac{N\Delta\tau}{4\lambda_i}\eta^2_{i,\tau}}, \tag{195}$$

$P \in [0,1]$ and δ corresponds to the Dirac δ-function. That is, we carry out simple sampling of the field with probability P and leave the field unchanged with probability $(1-P)$. P is a free parameter that does not change the final result but that allows to adjust the acceptance. We then use the Metropolis-Hasting acceptance-rejection scheme and accept the move with probability

$$\min\left(\frac{T^0(\eta' \rightarrow \eta) W_B(\eta') W_F(\eta')}{T^0(\eta \rightarrow \eta') W_B(\eta) W_F(\eta)}, 1 \right) = \min\left(\frac{W_F(\eta')}{W_F(\eta)}, 1 \right). \tag{196}$$

where

$$W_B(\eta) = e^{-\frac{N\Delta\tau}{4}\sum_{i,\tau}\eta^2_{i,\tau}/\lambda_i} \text{ and } W_F(\eta) = \mathrm{Tr}\left[\prod_\tau e^{-\Delta\tau \hat{H}_T} e^{-\sum_{i,j} i\Delta\tau O_{i,j}\eta_{j,\tau}(\hat{n}_i - \frac{N}{2})} \right] \tag{197}$$

Since a local change on a single time slice in the η basis corresponds to a non-local in space update in the ϕ basis, we use the global update in space routine to carry out the update (see Sec. 2.2.3).

QMC implementation

The name space for this model class reads:

```
&VAR_LRC                        !! Variables for the  Long Range Coulomb class
ham_T          = 1.0           ! Specifies the hopping and chemical potential
ham_T2         = 1.0           ! For bilayer systems
ham_Tperp      = 1.0           ! For bilayer systems
ham_chem       = 1.0           ! Chemical potential
ham_U          = 4.0           ! On-site interaction
ham_alpha      = 0.1           ! Coulomb tail magnitude
Percent_change = 0.1           ! Parameter P
/
```

By setting α to zero we can test this code against the Hubbard code. For a 4×4 square lattice at $\beta t = 5$, $U/t = 4$, and half-band filling, `Hamiltonian_Hubbard_mod.F90` gives $E = -13.188896 \pm 0.001698$ and `Hamiltonian_LRC_mod.F90` $E = -13.198512 \pm 0.040029$. Note that for the Hubbard code we have used the default `Mz = True`. This option breaks SU(2) spin symmetry for a given HS configuration but produces very precise values of the energy. On the other hand, the LRC code is an SU(2) invariant code (as would be choosing `Mz = False`) and produces more fluctuations in the double occupancy. This partly explains the difference in error bars between the two codes. To produce this data, one can run the pyALF python script: `LRC.py`

9.5. $\mathbb{Z}_2$ lattice gauge theories coupled to fermion and $\mathbb{Z}_2$ matter `Z2_mod.F90`

The Hamiltonian we will consider here reads

$$\begin{aligned}\hat{H} = & -t_{Z_2} \sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma} \hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \left(\hat{\Psi}^\dagger_{\boldsymbol{i},\sigma} \hat{\Psi}_{\boldsymbol{j},\sigma} + h.c. \right) - \mu \sum_{\boldsymbol{i},\sigma} \hat{\Psi}^\dagger_{\boldsymbol{i},\sigma} \hat{\Psi}_{\boldsymbol{i},\sigma} - g \sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \hat{\sigma}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} + K \sum_{\Box} \prod_{\langle \boldsymbol{i},\boldsymbol{j}\rangle \in \partial\Box} \hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \\ & + J \sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \hat{\tau}^z_{\boldsymbol{i}} \hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \hat{\tau}^z_{\boldsymbol{j}} - h \sum_{\boldsymbol{i}} \hat{\tau}^x_{\boldsymbol{i}} - t \sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma} \hat{\tau}^z_{\boldsymbol{i}} \hat{\tau}^z_{\boldsymbol{j}} \left(\hat{\Psi}^\dagger_{\boldsymbol{i},\sigma} \hat{\Psi}_{\boldsymbol{j},\sigma} + h.c. \right) + \frac{U}{N} \sum_{\boldsymbol{i}} \left[\sum_\sigma (\hat{\Psi}^\dagger_{\boldsymbol{i},\sigma} \hat{\Psi}_{\boldsymbol{i},\sigma} - 1/2) \right]^2\end{aligned} \tag{198}$$

The model is defined on a square lattice, and describers fermions,

$$\left\{ \hat{\Psi}^\dagger_{\boldsymbol{i},\sigma}, \hat{\Psi}_{\boldsymbol{j},\sigma'} \right\} = \delta_{\boldsymbol{i},\boldsymbol{j}} \delta_{\sigma,\sigma'}, \; \left\{ \hat{\Psi}_{\boldsymbol{i},\sigma}, \hat{\Psi}_{\boldsymbol{j},\sigma'} \right\} = 0, \tag{199}$$

coupled to bond gauge fields,

$$\hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \hat{\sigma}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \left\{ \hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}, \hat{\sigma}^x_{\langle \boldsymbol{i}',\boldsymbol{j}'\rangle} \right\} = 2\left(1 - \delta_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\langle \boldsymbol{i}',\boldsymbol{j}'\rangle}\right) \hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\sigma}^x_{\langle \boldsymbol{i}',\boldsymbol{j}'\rangle} \tag{200}$$

and Z_2 matter fields:

$$\hat{\tau}^z_{\boldsymbol{i}} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \hat{\tau}^x_{\boldsymbol{i}} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \{\hat{\tau}^z_{\boldsymbol{i}}, \hat{\tau}^x_{\boldsymbol{i}'}\} = 2\left(1 - \delta_{\boldsymbol{i},\boldsymbol{i}'}\right) \hat{\tau}^z_{\boldsymbol{i}}\hat{\tau}^x_{\boldsymbol{i}'}. \tag{201}$$

Fermions, gauge fields and Z_2 matter fields commute with each other.

Importantly, the model has a local Z_2 symmetry. Consider:

$$\hat{Q}_{\boldsymbol{i}} = (-1)^{\sum_\sigma \hat{\Psi}^\dagger_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{i},\sigma}}\, \hat{\tau}^x_{\boldsymbol{i}}\; \hat{\sigma}^x_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x}\hat{\sigma}^x_{\boldsymbol{i},\boldsymbol{i}-\boldsymbol{a}_x}\hat{\sigma}^x_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_y}\hat{\sigma}^x_{\boldsymbol{i}}. \tag{202}$$

One can then show that $\hat{Q}^2_{\boldsymbol{i}} = 1$ and that

$$\left[\hat{Q}_{\boldsymbol{i}}, \hat{H}\right] = 0. \tag{203}$$

The above allows us to assign Z_2 charges to the operators. Since $\left\{\hat{Q}_{\boldsymbol{i}}, \hat{\Psi}^\dagger_{\boldsymbol{i},\sigma}\right\} = 0$ we can assign a Z_2 charge to the fermions. Equivalently $\hat{\tau}^z_{\boldsymbol{i}}$ has a Z_2 charge and $\hat{\sigma}^z_{\boldsymbol{i},\boldsymbol{j}}$ carries Z_2 charges at its ends. Since the total fermion number is conserved, we can assign an electric charge to the fermions. Finally, the model has an SU(N) color symmetry. In fact, at zero chemical potential and $U = 0$, the symmetry is enhanced to O(2N) [17]. Aspects of this Hamiltonian were investigated in Refs. [17, 90, 91, 92, 93, 94] and we refer the interested user to these papers for a discussion of the phases and phase transitions supported by the model.

QMC implementation

The name space for this model class reads:

```
&VAR_Z2_Matter              !! Variables for the Z_2 class
ham_T          = 1.0         ! Hopping for fermions
ham_TZ2        = 1.0         ! Hopping for orthogonal fermions
ham_chem       = 0.0         ! Chemical potential for fermions
ham_U          = 0.0         ! Hubbard for fermions
Ham_J          = 1.0         ! Hopping Z2 matter fields
Ham_K          = 1.0         ! Plaquette term for gauge fields
Ham_h          = 1.0         ! sigma^x-term for matter
Ham_g          = 1.0         ! tau^x-term for gauge
Dtau           = 0.1d0       ! Thereby Ltrot=Beta/dtau
Beta           = 10.d0       ! Inverse temperature
Projector      = False       ! To enable projective code
Theta          = 10.0        ! Projection parameter
/
```

We note that the implementation is such that if `Ham_T=0` (`Ham_TZ2=0`) then all the terms involving the matter field (Z_2 gauge field) are automatically set to zero. We warn the user that autocorrelation and warmup times ca be large for this model class. At this point, only the finite temperature code is implemented.

The key point to implement the model, is to define a new bond variable:

$$\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} = \hat{\tau}^z_{\boldsymbol{i}}\hat{\tau}^z_{\boldsymbol{j}}. \tag{204}$$

By construction, the $\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ bond variables have a zero flux constraint:

$$\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x\rangle}\hat{\mu}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_x,\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y\rangle}\hat{\mu}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y,\boldsymbol{i}+\boldsymbol{a}_y\rangle}\hat{\mu}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_y,\boldsymbol{i}\rangle} = 1. \tag{205}$$

Consider a basis where $\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ and $\hat{\tau}^z_{\boldsymbol{i}}$ are diagonal with eigenvalues $\mu_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ and $\tau_{\boldsymbol{i}}$ respectively. The map from $\{\tau_{\boldsymbol{i}}\}$ to $\left\{\mu_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\right\}$ is unique. The reverse however is valid only up to to a global sign. To pin down this sign (and thereby the relative signs between different time slices) we store per time slice the $\mu_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$

fields as well as the value of the Ising field at a reference site $\tau_{\boldsymbol{i}=\boldsymbol{0}}$. Within the ALF, this can be done by adding a dummy operator in the `Op_V` list which will carry this degree of freedom. With this extra degree of freedom we can switch between the two representations, without loosing any information. To compute the Ising part of the action it is certainly more transparent to work with the $\{\tau_{\boldsymbol{i}}\}$ variables. For the fermion determinant, the $\left\{\mu_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\right\}$ are more convenient.

Since flipping $\hat{\tau}^z_{\boldsymbol{i}}$ amounts to changing the sign of the four bond variables emanating from site $\boldsymbol{i}$, the identitiy:

$$\hat{\tau}^x_{\boldsymbol{i}} = \hat{\mu}^x_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x}\hat{\mu}^x_{\boldsymbol{i}+\boldsymbol{a}_x,\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y}\hat{\mu}^x_{\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y,\boldsymbol{i}+\boldsymbol{a}_y} \tag{206}$$

holds. Note that $\left\{\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle},\hat{\mu}^x_{\langle \boldsymbol{i}',\boldsymbol{j}'\rangle}\right\} = 2\left(1-\delta_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\langle \boldsymbol{i}',\boldsymbol{j}'\rangle}\right)\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\mu}^x_{\langle \boldsymbol{i}',\boldsymbol{j}'\rangle}$ such that applying $\hat{\mu}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ on an eigenstate of $\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ flips the field.

The model can then be written as:

$$\begin{aligned}
\hat{H} = &- t_{Z_2}\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma}\hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\left(\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{j},\sigma}+h.c.\right) - \mu\sum_{\boldsymbol{i},\sigma}\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{i},\sigma} - g\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\sigma}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} + K\sum_{\square}\prod_{\langle \boldsymbol{i},\boldsymbol{j}\rangle\in\partial\square}\hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \\
&+ J\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} - h\sum_{\boldsymbol{i}}\hat{\mu}^x_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x}\hat{\mu}^x_{\boldsymbol{i}+\boldsymbol{a}_x,\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y}\hat{\mu}^x_{\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y,\boldsymbol{i}+\boldsymbol{a}_y}\hat{\mu}^x_{\boldsymbol{i}+\boldsymbol{a}_y,\boldsymbol{i}} && (207)\\
&- t\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma}\hat{\mu}^z_{\boldsymbol{i},\boldsymbol{j}}\left(\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{j},\sigma}+h.c.\right) + \frac{U}{N}\sum_{\boldsymbol{i}}\left[\sum_{\sigma}(\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{i},\sigma}-1/2)\right]^2 && (208)
\end{aligned}$$

subject to the constraint of Eq. 205.

To formulate the Monte Carlo, we work in a basis in which $\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$, $\hat{\tau}^z_{\boldsymbol{0}}$ and $\hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ are diagonal:

$$\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}|\underline{s}\rangle = \mu_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}|\underline{s}\rangle, \hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}|\underline{s}\rangle = \sigma_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}|\underline{s}\rangle, \hat{\tau}^z_{\boldsymbol{0}}|\underline{s}\rangle = \tau_{\boldsymbol{0}}|\underline{s}\rangle \tag{209}$$

with $\underline{s} = \left(\left\{\mu_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\right\},\left\{\sigma_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\right\},\tau_{\boldsymbol{0}}\right)$. In this basis,

$$Z = \sum_{\underline{s}_1,\cdots,\underline{s}_{L_\tau}} e^{-S_0(\{\underline{s}_\tau\})}\mathrm{Tr}_F\left[\prod_{\tau=1}^{L_\tau} e^{-\Delta\tau\hat{H}_F(\underline{s}_\tau)}\right] \tag{210}$$

where

$$S_0(\{\underline{s}_\tau\}) = -\ln\left[\prod_{\tau=1}^{L_\tau}\langle\underline{s}_{\tau+1}|e^{-\Delta\tau\hat{H}_I}|\underline{s}_\tau\rangle\right],$$

$$\begin{aligned}
\hat{H}_I = &- g\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\sigma}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} + K\sum_{\square}\prod_{\langle \boldsymbol{i},\boldsymbol{j}\rangle\in\partial\square}\hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} + J\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \\
&- h\sum_{\boldsymbol{i}}\hat{\mu}^x_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x}\hat{\mu}^x_{\boldsymbol{i}+\boldsymbol{a}_x,\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y}\hat{\mu}^x_{\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y,\boldsymbol{i}+\boldsymbol{a}_y}
\end{aligned}$$

and

$$\begin{aligned}
\hat{H}_F(\underline{s}) = &- t_{Z_2}\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma}\sigma_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\left(\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{j},\sigma}+h.c.\right) - \mu\sum_{\boldsymbol{i},\sigma}\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{i},\sigma} - t\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma}\mu_{\boldsymbol{i},\boldsymbol{j}}\left(\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{j},\sigma}+h.c.\right) \\
&+ \frac{U}{N}\sum_{\boldsymbol{i}}\left[\sum_{\sigma}(\hat{\Psi}^{\dagger}_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{i},\sigma}-1/2)\right]^2.
\end{aligned}$$

In the above, $|\underline{s}_{L_\tau+1}\rangle = |\underline{s}_1\rangle$. With a further HS transformation of the Hubbard term (see Sec. 8.3.1) the model is readily implemented in the ALF. Including this HS field, l, (see Eq. 10) yields the configuration space:

$$C = \left(\left\{\mu_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\tau}\right\},\left\{\sigma_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\tau}\right\},\left\{\tau_{\boldsymbol{0},\tau}\right\},\left\{l_{\boldsymbol{i},\tau}\right\}\right) \tag{211}$$

where the variables μ, τ and σ take the values ± 1 and l the values $\pm 1, \pm 2$.

The initial configuration as well as the moves have to respect the zero flux constraint of Eq. 205. Thereby single spin flips of the μ fields are prohibited and the minimal move one can carry out on a given time slice is the following. We randomly choose a site $\boldsymbol{i}$ and propose a move where: $\mu_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x} \rightarrow -\mu_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x}$, $\mu_{\boldsymbol{i},\boldsymbol{i}-\boldsymbol{a}_x} \rightarrow -\mu_{\boldsymbol{i},\boldsymbol{i}-\boldsymbol{a}_x}$, $\mu_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_y} \rightarrow -\mu_{\boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_y}$ and $\mu_{\boldsymbol{i},\boldsymbol{i}-\boldsymbol{a}_y} \rightarrow -\mu_{\boldsymbol{i},\boldsymbol{i}-\boldsymbol{a}_y}$. One can carry out such moves by using the global move in real space option presented in Sec. 2.2.3 and 5.7.1.

9.5.1. Projective approach

The program equally supports a zero temperature implementation. Our choice of the trial wave function does not break any symmetries of the model and reads:

$$|\Psi_T\rangle = |\Psi_T^F\rangle \otimes_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} |+\rangle_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \otimes_{\boldsymbol{i}} |+\rangle_{\boldsymbol{i}}. \tag{212}$$

For the fermion part we use a Fermi see with small dimerization to avoid the negative sign problem at half-filling (see Sec. 7.5). For the Ising part the trial wave function is diagonal in the $\hat{\sigma}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ and $\hat{\tau}^x_{\boldsymbol{i}}$ operators:

$$\hat{\sigma}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}|+\rangle_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} = |+\rangle_{\langle \boldsymbol{i},\boldsymbol{j}\rangle} \text{ and } \hat{\tau}^x_{\boldsymbol{i}}|+\rangle_{\boldsymbol{i}} = |+\rangle_{\boldsymbol{i}}. \tag{213}$$

An alternative choice would be to choose a charge density wave fermionic trial wave function. This violates the partial particle-hole symmetry of the model at $U = \mu = 0$ and effectively imposes the constraint $\hat{Q}_{\boldsymbol{i}} = 1$.

9.5.2. Observables

Aside from the standard observables discussed in Sec. 8.4 the code computes

$$\langle \hat{\sigma}^x_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}\rangle \text{ and } \langle \hat{\tau}^x_{\boldsymbol{j}}\rangle$$

(file `X_scal`),

$$\langle \hat{\sigma}^z_{\langle \boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x\rangle}\hat{\sigma}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_x,\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y\rangle}\hat{\sigma}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y,\boldsymbol{i}+\boldsymbol{a}_y\rangle}\hat{\sigma}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_y,\boldsymbol{i}\rangle}\rangle \text{ and } \langle \hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{i}+\boldsymbol{a}_x\rangle}\hat{\mu}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_x,\boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y\rangle}\hat{\mu}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_x+\boldsymbol{a}_y,\boldsymbol{i}+\boldsymbol{a}_y\rangle}\hat{\mu}^z_{\langle \boldsymbol{i}+\boldsymbol{a}_y,\boldsymbol{i}\rangle}\rangle$$

(file `Flux_scal`), as well as $\langle \hat{Q}_{\boldsymbol{i}}\rangle$ (file `Q_scal`). Note that the flux over a plaquette of the $\hat{\mu}^z_{\langle \boldsymbol{i},\boldsymbol{j}\rangle}$ is equal to unity by construction so that this observable provides a cross-check. The file `Q_eq` contains the two point correlation $\langle \hat{Q}_{\boldsymbol{i}}\hat{Q}_{\boldsymbol{j}}\rangle - \langle \hat{Q}_{\boldsymbol{i}}\rangle\langle \hat{Q}_{\boldsymbol{j}}\rangle$ and `Greenf_eq` the equal time fermion Green function $\langle \hat{\tau}^z_{\boldsymbol{i}}\hat{\Psi}^\dagger_{\boldsymbol{i},\sigma}\hat{\tau}^z_{\boldsymbol{j}}\hat{\Psi}_{\boldsymbol{j},\sigma}\rangle$.

9.5.3. A test case: Z_2 slave spin formulation of the SU(2) Hubbard model

In this subsection, we demonstrate that the code can be used to simulate the attractive Hubbard model in the Z_2-slave spin formulation [95]:

$$\hat{H} = -t\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma}\hat{c}^\dagger_{\boldsymbol{i},\sigma}\hat{c}_{\boldsymbol{j},\sigma} - U\sum_{i}\left(\hat{n}_{\boldsymbol{i},\uparrow} - 1/2\right)\left(\hat{n}_{\boldsymbol{i},\downarrow} - 1/2\right). \tag{214}$$

In the Z_2 slave spin representation, the physical fermion, $\hat{c}_{\boldsymbol{i},\sigma}$, is fractionalized into an Ising spin carrying Z_2 charge and a fermion, $\hat{\Psi}_{\boldsymbol{i},\sigma}$, carrying Z_2 and global $U(1)$ charge:

$$\hat{c}^\dagger_{\boldsymbol{i},\sigma} = \hat{\tau}^z_{\boldsymbol{i}}\hat{\Psi}^\dagger_{\boldsymbol{i},\sigma}. \tag{215}$$

To ensure that we remain in the correct Hilbert space, the constraint:

$$\hat{\tau}^x_i - (-1)^{\sum_\sigma \hat{\Psi}^\dagger_{i,\sigma}\hat{\Psi}_{i,\sigma}} = 0 \tag{216}$$

has to be impose locally. Since $\left(\tau^x_{\boldsymbol{i}}\right)^2 = 1$ it is equivalent to

$$\hat{Q}_{\boldsymbol{i}} = \tau^x_{\boldsymbol{i}}(-1)^{\sum_\sigma \hat{\Psi}^\dagger_{i,\sigma}\hat{\Psi}_{i,\sigma}} = 1. \tag{217}$$

Using

$$(-1)^{\sum_\sigma \hat{\Psi}^\dagger_{i,\sigma}\hat{\Psi}_{i,\sigma}} = \prod_\sigma(1 - 2\hat{\Psi}^\dagger_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{i},\sigma}) = 4\prod_\sigma(\hat{c}^\dagger_{\boldsymbol{i},\sigma}\hat{c}_{\boldsymbol{i},\sigma} - 1/2), \tag{218}$$

the Z_2 slave spin representation of the Hubbard model now reads:

$$\hat{H}_{Z_2} = -t\sum_{\langle \boldsymbol{i},\boldsymbol{j}\rangle,\sigma}\hat{\tau}^z_{\boldsymbol{i}}\hat{\tau}^z_{\boldsymbol{j}}\hat{\Psi}^\dagger_{\boldsymbol{i},\sigma}\hat{\Psi}_{\boldsymbol{j},\sigma} - \frac{U}{4}\sum_{\boldsymbol{i}}\hat{\tau}^x_{\boldsymbol{i}}. \tag{219}$$

Importantly, the constraint commutes with Hamiltonian:

$$\left[\hat{H}_{Z_2}, \hat{Q}_{\boldsymbol{i}}\right] = 0. \tag{220}$$

Hence one can foresee that the constraint will be dynamically imposed (we expect a finite temperature Ising phase transitions below which $\hat{Q}_{\boldsymbol{i}}$ orders) and that at $T = 0$ on a finite lattice both models should give the same results.

A test run for the 8×8 lattice at $U/t = 4$ and $\beta t = 40$ gives:

k	$\langle n_k \rangle_H$	$\langle n_k \rangle_{H_{Z_2}}$
$(0, 0)$	$1.93348548 \pm 0.00011322$	$1.93333895 \pm 0.00010405$
$(\pi/4, \pi/4)$	$1.90120688 \pm 0.00014854$	$1.90203726 \pm 0.00017943$
$(\pi/2, \pi/2)$	$0.99942957 \pm 0.00091377$	$1.00000000 \pm 0.00000000$
$(3\pi/4, 3\pi/4)$	$0.09905425 \pm 0.00015940$	$0.09796274 \pm 0.00017943$
(π, π)	$0.06651452 \pm 0.00011321$	$0.06666105 \pm 0.00010405$

Here a Trotter time step of $\Delta\tau t = 0.05$ was used so as to minimize the systematic error which should be different between the two codes. The Hamiltonian is invariant under a partial particle-hole transformation (see Ref. [17]. Since $\hat{Q}_{\boldsymbol{i}}$ is odd under this transformation $\langle \hat{Q}_{\boldsymbol{i}} \rangle = 0$. To asses if the constraint is well imposed, the code, for this special case, computes the correlation function:

$$S_Q(\boldsymbol{q}) = \sum_{\boldsymbol{i}} \langle \hat{Q}_{\boldsymbol{i}} \hat{Q}_{\boldsymbol{0}} \rangle. \tag{221}$$

For the above run we obtain $S_Q(\boldsymbol{q} = \boldsymbol{0}) = 63.42 \pm 1.7$ which, for this 8×8 lattice, complies with a ferromagnetic ordering of the Ising $\hat{Q}_{\boldsymbol{i}}$ variables. The pyALF python script that produces this data can be found in `Z2_Matter.py` This code was used in Ref. [94].

10. Maximum Entropy

If we want to compare the data we obtain from Monte Carlo simulations with experiments, we must extract spectral information from the imaginary-time output. This can be achieve through the maximum entropy method (MaxEnt), which generically computes the image $A(\omega)$ for a given data set $g(\tau)$ and kernel $K(\tau, \omega)$:

$$g(\tau) = \int_{\omega_{\text{start}}}^{\omega_{\text{end}}} \mathrm{d}\omega K(\tau, \omega) A(\omega). \tag{222}$$

The ALF package includes a standard implementation of the stochastic MaxEnt, as formulated in the article of K. Beach [57], in the module `Libraries/Modules/maxent_stoch_mod.F90`. Its wrapper is contained in `Analysis/Max_SAC.F90` and the Green function is read from the output of the `cov_tau.F90` analysis program.

10.1. General setup

The stochastic MaxEnt is essentially a parallel-tempering Monte Carlo simulation. For a discrete set of τ_i points, $i \in 1 \cdots n$, the goodness-of-fit functional, which we take as the energy reads

$$\chi^2(A) = \sum_{i,j=1}^{n} \left[g(\tau_i) - \overline{g(\tau_i)} \right] C^{-1}(\tau_i, \tau_j) \left[g(\tau_j) - \overline{g(\tau_j)} \right], \tag{223}$$

with $\overline{g(\tau_i)} = \int \mathrm{d}\omega K(\tau_i, \omega) A(\omega)$ and C the covariance matrix. The set of inverse temperatures considered in the parallel tempering is given by $\alpha_m = \alpha_{st} R^m$, for $m = 1 \cdots N_\alpha$ and a constant R. The phase space corresponds to all possible spectral functions satisfying a given sum rule and the required positivity. Finally, the partition function reads $Z = \int \mathcal{D}A \, e^{-\alpha \chi^2(A)}$ [57].

In the code, the spectral function is parametrized by a set of N_γ Dirac δ functions:

$$A(\omega) = \sum_{i=1}^{N_\gamma} a_i \delta\left(\omega - \omega_i\right). \tag{224}$$

In order to produce a histogram of $A(\omega)$ we divide the frequency range in `Ndis` intervals.

Besides the parameters included in the namelist `VAR_Max_Stoch` set in the file `parameters` (see Sec. 5.7), also the variable `N_cov`, from the namelist `VAR_errors`, is required to run the maxent code. Recalling: `N_cov = 1` (`N_cov = 0`) sets that the covariance will (will not) be taken into account.

Input files

Aside for the aforementioned parameter file, the MaxEnt program reads requires the output of the analysis of the time displaced functions. After running `Anaylsis/ana.out` a directory named `${Variable_name}_${kx}_${ky}` is generate for each k-point. In this directory the file `g_${kx}_${ky}` contains the required information for the MaxEnt code and is formatted as follows:

```
 < # tau-points >  <# of bins >  <beta>  <Norb>  <Channel>
do tau = 1,  # tau-points
    τ ,    ∑_α ⟨S^(corr)_{α,α}(k,τ)⟩,    error
enddo
do tau1 = 1,  # tau-points
  do tau2 = 1,  # tau-points
      C( τ_1,τ_2 )
  enddo
enddo
```

Output files

The code produces the following output files:

- The files `Aom_n` correspond to the average spectral function at inverse temperature α_n. This corresponds to $\langle A_n(\omega)\rangle = \frac{1}{Z}\int \mathcal{D}A(\omega)\; e^{-\alpha_n\chi^2(A)}A(\omega)$. The file contains three columns: ω, $\langle A_n(\omega)\rangle$, and $\Delta\langle A_n(\omega)\rangle$.

- The files `Aom_ps_n` contain the average image over the inverse temperatures α_n to α_{N_γ} see Ref. [57] for more details. Its first three columns have the same meaning as for the files `Aom_n`.

- The file `Green` contains the Green function, obtained from the spectral function through

$$G(\omega) = -\frac{1}{\pi}\int \mathrm{d}\Omega \frac{A(\Omega)}{\omega - \Omega + i\delta}\,, \tag{225}$$

  where $\delta = \Delta\omega = (\omega_{\mathrm{end}} - \omega_{\mathrm{start}})/$`Ndis` and the image corresponds to that of the file `Aom_ps_m` with $m = N_\alpha - 10$. The first column of the `Green` file is a place holder for post-processing. The last three columns correspond to $\omega, \mathrm{Re}\,G(\omega), -\mathrm{Im}\,G(\omega)/\pi$.

- One of the most important output files is `energies`, which lists $\alpha_n, \langle\chi^2\rangle, \Delta\langle\chi^2\rangle$.

- `best_fit` gives the values of a_i and ω_i (recall that $A(\omega) = \sum_{i=1}^{N_\gamma} a_i\delta\left(\omega - \omega_i\right)$) corresponding to the last configuration of the lowest temperature run.

- The file `data_out` facilitates crosschecking. It lists τ, $g(\tau)$, $\Delta g(\tau)$, and $\int \mathrm{d}\omega K(\tau,\omega)A(\omega)$, where the image corresponds to the best fit (i.e. the lowest temperature). This data should give an indication of how good the fit actually is. Note that `data_out` contains only the data points that have passed the tolerance test.

- Two dump files are also generated, `dump_conf` and `dump_Aom`. Since the MaxEnt is a Monte Carlo code, it is possible to improve the data by continuing a previous simulation. The data in the dump files allow you to do so. These files are only generated if the variable `checkpoint` is set to `.true.`.

The essential question is: Which image should one use? There is no final answer to this question in the context of the stochastic MaxEnt. The only rule of thumb is to consider temperatures for which the χ^2 is comparable to the number of data points.

10.2. Single-particle quantities

For the single-particle Green function,

$$\langle \hat{c}_k(\tau)\hat{c}_k^\dagger(0)\rangle = \int \mathrm{d}\omega K_p(\tau,\omega) A_p(k,\omega), \tag{226}$$

with

$$K_p(\tau,\omega) = \frac{1}{\pi}\frac{e^{-\tau\omega}}{1+e^{-\beta\omega}} \tag{227}$$

and, in the Lehmann representation,

$$A_p(k,\omega) = \frac{\pi}{Z}\sum_{n,m} e^{-\beta E_n}\left(1+e^{-\beta\omega}\right)|\langle n|c_n|m\rangle|^2 \delta\left(E_m - E_n - \omega\right). \tag{228}$$

Here $\left(\hat{H} - \mu\hat{N}\right)|n\rangle = E_n|n\rangle$. Note that $A_p(k,\omega) = -\operatorname{Im} G^{\mathrm{ret}}(k,\omega)$, with

$$G^{\mathrm{ret}}(k,\omega) = -i\int \mathrm{d}t\Theta(t)e^{i\omega t}\langle\{\hat{c}_k(t),\hat{c}_k^\dagger(0)\}\rangle. \tag{229}$$

Finally the sum rule reads

$$\int \mathrm{d}\omega A_p(k,\omega) = \pi\langle\{\hat{c}_k,\hat{c}_k^\dagger\}\rangle = \pi\left(\langle\hat{c}_k(\tau=0)\hat{c}_k^\dagger(0)\rangle + \langle\hat{c}_k(\tau=\beta)\hat{c}_k^\dagger(0)\rangle\right) \tag{230}$$

Using the `Max_Sac.F90` with `Channel="P"` will load the above kernel in the MaxEnt library. In this case the back transformation is set to unity. Note that for each configuration of fields we have $\langle\langle\hat{c}_k(\tau = 0)\hat{c}_k^\dagger(0)\rangle\rangle_C + \langle\langle\hat{c}_k(\tau=\beta)\hat{c}_k^\dagger(0)\rangle\rangle_C = \langle\langle\{\hat{c}_k,\hat{c}_k^\dagger\}\rangle\rangle_C = 1$, hence, if both the $\tau = 0$ and $\tau = \beta$ data points are included, the covariance matrix will have a zero eigenvalue and the χ^2 measure is not defined. Therefore, for the particle channel the program omits the $\tau = \beta$ data point. There are special particle-hole symmetric cases where the $\tau = 0$ data point shows no fluctuations – in such cases the code omits the $\tau = 0$ data point as well.

10.3. Particle-hole quantities

Imaginary-time formulation

For particle-hole quantities such as spin-spin or charge-charge correlations, the kernel reads

$$\langle\hat{S}(q,\tau)\hat{S}(-q,0)\rangle = \frac{1}{\pi}\int \mathrm{d}\omega\frac{e^{-\tau\omega}}{1-e^{-\beta\omega}}\chi''(q,\omega). \tag{231}$$

This follows directly from the Lehmann representation

$$\chi''(q,\omega) = \frac{\pi}{Z}\sum_{n,m} e^{-\beta E_n}|\langle n|\hat{S}(q)|m\rangle|^2\delta(\omega + E_n - E_m)\left(1-e^{-\beta\omega}\right). \tag{232}$$

Since the linear response to a Hermitian perturbation is real, $\chi''(q,\omega) = -\chi''(-q,-\omega)$ and hence $\langle\hat{S}(q,\tau)\hat{S}(-q,0)\rangle$ is a symmetric function around $\beta = \tau/2$ for systems with inversion symmetry – the ones we consider here. The analysis file `cov_tau_ph.F90` produced at compilation time uses this to define an improved estimator.

The stochastic MaxEnt requires a sum rule, and hence the kernel and image have to be adequately redefined. Let us consider $\coth(\beta\omega/2)\chi''(q,\omega)$. For this quantity, we have the sum rule, since

$$\int \mathrm{d}\omega \coth(\beta\omega/2)\chi''(q,\omega) = 2\pi\langle\hat{S}(q,\tau=0)\hat{S}(-q,0)\rangle, \tag{233}$$

which is just the first point in the data. Therefore,

$$\langle\hat{S}(q,\tau)\hat{S}(-q,0)\rangle = \int \mathrm{d}\omega\, \underbrace{\frac{1}{\pi}\frac{e^{-\tau\omega}}{1-e^{-\beta\omega}}\tanh(\beta\omega/2)}_{K_{pp}(\tau,\omega)}\underbrace{\coth(\beta\omega/2)\chi''(q,\omega)}_{A(\omega)} \tag{234}$$

and one computes $A(\omega)$. Note that since χ'' is an odd function of ω one restricts the integration range positive values of ω. Hence:

$$\langle \hat{S}(q,\tau)\hat{S}(-q,0)\rangle = \int_0^\infty \mathrm{d}\omega \underbrace{\left(K(\tau,\omega) + K(\tau,-\omega)\right)}_{K_{ph}(\tau,\omega)} A(\omega). \tag{235}$$

In the code, ω_{start} is set to zero by default and the kernel K_{ph} is defined in the routine `XKER_ph`.

In general, one would like to produce the dynamical structure factor that gives the susceptibility according to

$$S(q,\omega) = \chi''(q,\omega)/\left(1 - e^{-\beta\omega}\right). \tag{236}$$

In the code the routine `BACK_TRANS_ph` transforms the image A to the desired quantity:

$$S(q,\omega) = \frac{A(\omega)}{1 + e^{-\beta\omega}}. \tag{237}$$

Matsubara-frequency formulation

The ALF library uses imaginary time. It is however possible to formulate the MaxEnt in Matsubara frequencies. Consider:

$$\chi(q, i\Omega_m) = \int_0^\beta \mathrm{d}\tau e^{i\Omega_m\tau}\langle \hat{S}(q,\tau)\hat{S}(-q,0)\rangle = \frac{1}{\pi}\int \mathrm{d}\omega \frac{\chi''(q,\omega)}{\omega - i\Omega_m}. \tag{238}$$

Using the fact that $\chi''(q,\omega) = -\chi''(-q,-\omega) = -\chi''(q,-\omega)$ one obtains

$$\begin{aligned} \chi(q, i\Omega_m) &= \frac{1}{\pi}\int_0^\infty \mathrm{d}\omega \left(\frac{1}{\omega - i\Omega_m} - \frac{1}{-\omega - i\Omega_m}\right)\chi''(q,\omega) \\ &= \frac{2}{\pi}\int_0^\infty \mathrm{d}\omega \frac{\omega^2}{\omega^2 + \Omega_m^2}\frac{\chi''(q,\omega)}{\omega} \\ &\equiv \int_0^\infty \mathrm{d}\omega K(\omega, i\Omega_m) A(q,\omega), \end{aligned} \tag{239}$$

with

$$K(\omega, i\Omega_m) = \frac{\omega^2}{\omega^2 + \Omega_m^2} \quad \text{and} \quad A(q,\omega) = \frac{2}{\pi}\frac{\chi''(q,\omega)}{\omega}. \tag{240}$$

The above definitions produce an image that satisfies the sum rule:

$$\int_0^\infty \mathrm{d}\omega A(q,\omega) = \frac{1}{\pi}\int_{-\infty}^\infty \mathrm{d}\omega \frac{\chi''(q,\omega)}{\omega} \equiv \chi(q, i\Omega_m = 0). \tag{241}$$

10.4. Particle-Particle quantities

Similarly to the particle-hole channel, the particle-particle channel is also a bosonic correlation function. Here, however, we do not assume that the imaginary time data is symmetric around the $\tau = \beta/2$ point. We use the kernel K_{pp} defined in Eq. (234) and consider the whole frequency range. The back transformation yields

$$\frac{\chi''(\omega)}{\omega} = \frac{\tanh\left(\beta\omega/2\right)}{\omega} A(\omega). \tag{242}$$

10.5. Zero-temperature, projective code

In the zero temperature limit, the spectral function associated to an operator $\hat{O}$ reads:

$$A_o(\omega) = \pi \sum_n |\langle n|\hat{O}|0\rangle|^2 \delta(E_n - E_0 - \omega), \tag{243}$$

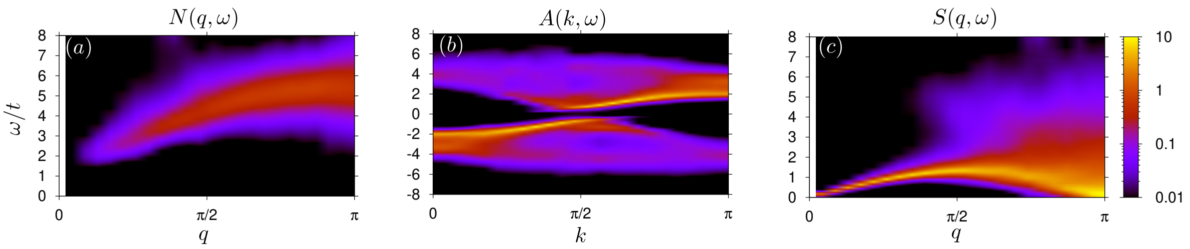

Figure 9: Dynamics of the one-dimensional half-filled Hubbard model on a 46-site chain and at U/t=4 and at $\beta t = 10$. (a) Dynamical charge structure factor, (b) single particle spectral function and (c) dynamical spin structure factor. As apparent in the pyALF python script `Hubbard_1D.py` we consider 400 bins of 200 sweeps each and take into account the covariance matrix for the MaxEnt. The parameters for the MaxEnt that differ for the default values are also listed in the python script.

such that

$$\langle 0|\hat{O}^\dagger(\tau)\hat{O}(0)|0\rangle = \int \mathrm{d}\omega K_0(\tau,\omega)A_0(\omega), \tag{244}$$

with

$$K_0(\tau,\omega) = \frac{1}{\pi}e^{-\tau\omega}. \tag{245}$$

The zeroth moment of the spectral function reads

$$\int \mathrm{d}\omega A_o(\omega) = \pi\langle 0|\hat{O}^\dagger(0)\hat{O}(0)|0\rangle, \tag{246}$$

and hence corresponds to the first data point.

In the zero-temperature limit one does not distinguish between particle, particle-hole, or particle-particle channels. Using the `Max_Sac.F90` with `Channel="T0"` loads the above kernel in the MaxEnt library. In this case the back transformation is set to unity. The code will also cut-off the tail of the imaginary time correlation function if the relative error is greater that the variable `Tolerance`.

10.6. Dynamics of the one-dimensional half-filled Hubbard model

To conclude this section, we show the example of the one dimensional Hubbard model that is know to show spin-charge separation (see Ref. [96] and references therein). The data of Fig. 9 was produced with the pyALF python script `Hubbard_1D.py`, and the spectral functions plots with the bash `Spectral.sh`.

11. Conclusions and Future Directions

In its present form, the auxiliary-field QMC code of the ALF project allows us to simulate a large class of non-trivial models, both efficiently and at minimal programming cost. ALF 2.0 contains many advanced functionalities, including a projective formulation, various updating schemes, better control of Trotter errors, predefined structures that facilitate reuse, a large class of models, continuous fields and, finally, stochastic analytical continuation code. Also the usability of the code has improved in comparison with ALF 1.0. In particular the pyALF project provides a Python interface to the ALF which substantially facilitates running the code for established models.

There are further capabilities that we would like to see in future versions of ALF. Introducing time-dependent Hamiltonians, for instance, will require some rethinking, but will allow, for example, to access entanglement properties of interacting fermionic systems [31, 97, 32, 33]. Moreover, the auxiliary field approach is not the only method to simulate fermionic systems. It would desirable to include additional lattice fermion algorithms such as the CT-INT [98, 49]. Lastly, at the more technical level, improved IO (eg. HDF5 support), post-processing, object oriented programming, as well as increased compatibility with other software projects are all certainly desirable improvements to look forward to.

Acknowledgments

We are very grateful to B. Danu, S. Beyl, M. Hohenadler, F. Parisen Toldin, M. Raczkowski, T. Sato, Z. Wang, M. Weber for constant support during the development of this project. We equally thank G. Hager, M. Wittmann, and G. Wellein for useful discussions and support. FFA would also like to thank T. Lang and Z. Y. Meng for developments of the auxiliary field code as well as T. Grover. MB, FFA and FG thank the Bavarian Competence Network for Technical and Scientific High Performance Computing (KONWIHR) for financial support. FG, JH, and JS thank the SFB-1170 for financial support under projects Z03 and C01. JSEP thanks the DFG for financial support under the project AS120/14-1 dedicated to the further development of the ALF-library. Part of the optimization of the code was carried out during the Porting and Tuning Workshop 2016 offered by the Forschungszentrum Jülich. Calculations to extensively test this package were carried out both on SuperMUC-NG at the Leibniz Supercomputing Centre and on JURECA [99] at the Jülich Supercomputing Centre. We thank both institutions for generous allocation of computing time.

A. Performance, memory requirements and parallelization

As mentioned in the introduction, the auxiliary field QMC algorithm scales linearly in inverse temperature β and cubic in the volume N_{dim}. Using fast updates, a single spin flip requires $(N_{\mathrm{dim}})^2$ operations to update the Green function upon acceptance. As there are $L_{\mathrm{Trotter}} \times N_{\mathrm{dim}}$ spins to be visited, the total computational cost for one sweep is of the order of $\beta(N_{\mathrm{dim}})^3$. This operation dominates the performance, see Fig. 10. A profiling analysis of our code shows that 80-90% of the CPU time is spend in ZGEMM calls of the BLAS library provided in the MKL package by Intel. Consequently, the single-core performance is next to optimal.

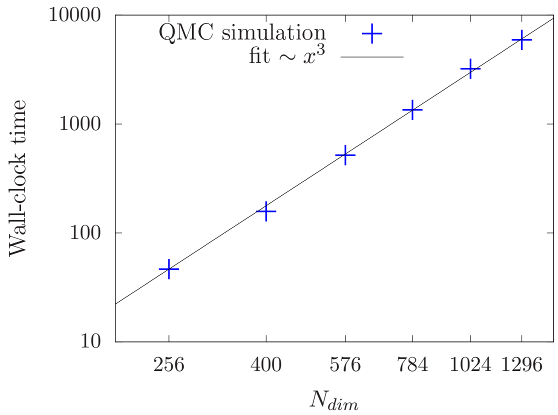

Figure 10: Volume scaling behavior of the auxiliary field QMC code of the ALF project on SuperMUC (phase 2/Haswell nodes) at the LRZ in Munich. The number of sites N_{dim} corresponds to the system volume. The plot confirms that the leading scaling order is due to matrix multiplications such that the runtime is dominated by calls to ZGEMM.

For the implementation which scales linearly in β, one has to store $L_{\mathrm{Trotter}}/\texttt{NWrap}$ intermediate propagation matrices of dimension $N \times N$. For large lattices and/or low temperatures this dominates the total memory requirements that can exceed 2 GB memory for a sequential version.

At the heart of Monte Carlo schemes lies a random walk through the given configuration space. This is easily parallelized via MPI by associating one random walker to each MPI task. For each task, we start from a random configuration and have to invest the autocorrelation time T_{auto} to produce an equilibrated configuration. Additionally we can also profit from an OpenMP parallelized version of the BLAS/LAPACK library for an additional speedup, which also effects equilibration overhead $N_{\mathrm{MPI}} \times T_{\mathrm{auto}}/N_{\mathrm{OMP}}$, where N_{MPI} is the number of cores and N_{OMP} the number of OpenMP threads. For a given number of independent measurements N_{meas}, we therefore need a wall-clock time given by

$$T = \frac{T_{\mathrm{auto}}}{N_{\mathrm{OMP}}} \left(1 + \frac{N_{\mathrm{meas}}}{N_{\mathrm{MPI}}} \right) . \tag{247}$$

As we typically have $N_{\rm meas}/N_{\rm MPI} \gg 1$, the speedup is expected to be almost perfect, in accordance with the performance test results for the auxiliary field QMC code on SuperMUC (see Fig. 11 (left)).

For many problem sizes, 2 GB memory per MPI task (random walker) suffices such that we typically start as many MPI tasks as there are physical cores per node. Due to the large amount of CPU time spent in MKL routines, we do not profit from the hyper-threading option. For large systems, the memory requirement increases and this is tackled by increasing the amount of OpenMP threads to decrease the stress on the memory system and to simultaneously reduce the equilibration overhead (see Fig. 11 (right)). For the displayed speedup, it was crucial to pin the MPI tasks as well as the OpenMP threads in a pattern which keeps the threads as compact as possible to profit from a shared cache. This also explains the drop in efficiency from 14 to 28 threads where the OpenMP threads are spread over both sockets.

We store the field configurations of the random walker as checkpoints, such that a long simulation can be easily split into several short simulations. This procedure allows us to take advantage of chained jobs using the dependency chains provided by the batch system.

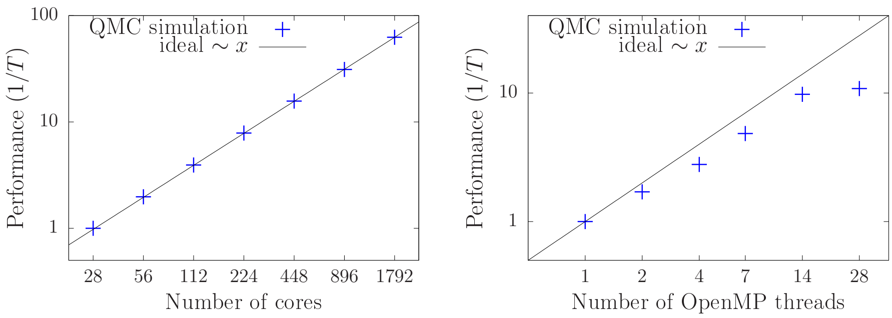

Figure 11: MPI (left) and OpenMP (right) scaling behavior of the auxiliary field QMC code of the ALF project on SuperMUC (phase 2/Haswell nodes) at the LRZ in Munich. The MPI performance data was normalized to 28 cores and was obtained using a problem size of $N_{\rm dim} = 400$. This is a medium to small system size that is the least favorable in terms of MPI synchronization effects. The OpenMP performance data was obtained using a problem size of $N_{\rm dim} = 1296$. Employing 2 and 4 OpenMP threads introduces some synchronization/management overhead such that the per-core performance is slightly reduced, compared to the single thread efficiency. Further increasing the amount of threads to 7 and 14 keeps the efficiency constant. The drop in performance of the 28 thread configuration is due to the architecture as the threads are now spread over both sockets of the node. To obtain the above results, it was crucial to pin the processes in a fashion that keeps the OpenMP threads as compact as possible.

References

[1] R. Blankenbecler, D. J. Scalapino, and R. L. Sugar, Phys. Rev. D **24**, 2278 (1981).

[2] S. White, D. Scalapino, R. Sugar, E. Loh, J. Gubernatis, and R. Scalettar, Phys. Rev. B **40**, 506 (1989).

[3] G. Sugiyama and S. Koonin, Annals of Physics **168**, 1 (1986).

[4] S. Sorella, S. Baroni, R. Car, and M. Parrinello, EPL (Europhysics Letters) **8**, 663 (1989).

[5] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth, Phys. Lett. **B195**, 216 (1987).

[6] F. Assaad and H. Evertz, in *Computational Many-Particle Physics*, Vol. 739 of *Lecture Notes in Physics*, edited by H. Fehske, R. Schneider, and A. Weiße (Springer, Berlin Heidelberg, 2008), pp. 277–356.

[7] D. J. Scalapino, in *Handbook of High-Temperature Superconductivity: Theory and Experiment*, edited by J. R. Schrieffer and J. S. Brooks (Springer New York, New York, NY, 2007), pp. 495–526.

[8] J. P. F. LeBlanc, A. E. Antipov, F. Becca, I. W. Bulik, G. K.-L. Chan, C.-M. Chung, Y. Deng, M. Ferrero, T. M. Henderson, C. A. Jiménez-Hoyos, E. Kozik, X.-W. Liu, A. J. Millis, N. V. Prokof'ev, M. Qin, G. E. Scuseria, H. Shi, B. V. Svistunov, L. F. Tocchio, I. S. Tupitsyn, S. R. White, S. Zhang, B.-X. Zheng, Z. Zhu, and E. Gull, Phys. Rev. X **5**, 041041 (2015).

[9] M. Hohenadler, T. C. Lang, and F. F. Assaad, Phys. Rev. Lett. **106**, 100403 (2011).

[10] D. Zheng, G.-M. Zhang, and C. Wu, Phys. Rev. B **84**, 205121 (2011).

[11] F. F. Assaad and I. F. Herbut, Phys. Rev. X **3**, 031010 (2013).

[12] F. Parisen Toldin, M. Hohenadler, F. F. Assaad, and I. F. Herbut, Phys. Rev. B **91**, 165108 (2015).

[13] Y. Otsuka, S. Yunoki, and S. Sorella, Phys. Rev. X **6**, 011029 (2016).

[14] S. Chandrasekharan and A. Li, Phys. Rev. D **88**, 021701 (2013).

[15] V. Ayyar and S. Chandrasekharan, Phys. Rev. D **91**, 065035 (2015).

[16] Z.-X. Li, Y.-F. Jiang, S.-K. Jian, and H. Yao, ArXiv:1512.07908 (2015).

[17] F. F. Assaad and T. Grover, Phys. Rev. X **6**, 041049 (2016).

[18] F. F. Assaad, Phys. Rev. Lett. **83**, 796 (1999).

[19] S. Capponi and F. F. Assaad, Phys. Rev. B **63**, 155114 (2001).

[20] Y. Schattner, S. Lederer, S. A. Kivelson, and E. Berg, Phys. Rev. X **6**, 031028 (2016).

[21] X. Y. Xu, K. Sun, Y. Schattner, E. Berg, and Z. Y. Meng, Phys. Rev. X **7**, 031058 (2017).

[22] E. Berg, M. A. Metlitski, and S. Sachdev, Science **338**, 1606 (2012).

[23] H.-K. Tang, X. Yang, J. Sun, and H.-Q. Lin, Europhys. Lett. **107**, 40003 (2014).

[24] F. F. Assaad, Phys. Rev. B **71**, 075103 (2005).

[25] T. C. Lang, Z. Y. Meng, A. Muramatsu, S. Wessel, and F. F. Assaad, Phys. Rev. Lett. **111**, 066401 (2013).

[26] M. Hohenadler, F. Parisen Toldin, I. F. Herbut, and F. F. Assaad, Phys. Rev. B **90**, 085146 (2014).

[27] H.-K. Tang, E. Laksono, J. N. B. Rodrigues, P. Sengupta, F. F. Assaad, and S. Adam, Phys. Rev. Lett. **115**, 186602 (2015).

[28] M. Rigol, A. Muramatsu, G. G. Batrouni, and R. T. Scalettar, Phys. Rev. Lett. **91**, 130403 (2003).

[29] D. Lee, Progress in Particle and Nuclear Physics **63**, 117 (2009).

[30] T. Grover, Phys. Rev. Lett. **111**, 130402 (2013).

[31] P. Broecker and S. Trebst, Journal of Statistical Mechanics: Theory and Experiment **2014**, P08015 (2014).

[32] F. F. Assaad, T. C. Lang, and F. Parisen Toldin, Phys. Rev. B **89**, 125121 (2014).

[33] F. F. Assaad, Phys. Rev. B **91**, 125146 (2015).

[34] C. Wu and S.-C. Zhang, Phys. Rev. B **71**, 155115 (2005).

[35] E. F. Huffman and S. Chandrasekharan, Phys. Rev. B **89**, 111101 (2014).

[36] Z.-X. Li, Y.-F. Jiang, and H. Yao, Phys. Rev. B **91**, 241117 (2015).

[37] Z. C. Wei, C. Wu, Y. Li, S. Zhang, and T. Xiang, Phys. Rev. Lett. **116**, 250601 (2016).

[38] J. Hubbard, Phys. Rev. Lett. **3**, 77 (1959).

[39] M. Troyer and U.-J. Wiese, Phys. Rev. Lett. **94**, 170201 (2005).

[40] S. Duane and J. B. Kogut, Phys. Rev. Lett. **55**, 2774 (1985).

[41] J. Hirsch, Phys. Rev. B **28**, 4059 (1983).

[42] A. D. Sokal, Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms, 1989, lecture notes from Cours de Troisième Cycle de la Physique en Suisse Romande. Updated in 1996 for the Cargèse Summer School on "Functional Integration: Basics and Applications".

[43] H. G. Evertz, G. Lana, and M. Marcu, Phys. Rev. Lett. **70**, 875 (1993).

[44] A. W. Sandvik, Phys. Rev. B **59**, R14157 (1999).

[45] O. Syljuåsen and A. Sandvik, Phys. Rev. E **66**, 046701 (2002).

[46] J. E. Hirsch and R. M. Fye, Phys. Rev. Lett. **56**, 2521 (1986).

[47] E. Gull, A. J. Millis, A. I. Lichtenstein, A. N. Rubtsov, M. Troyer, and P. Werner, Rev. Mod. Phys. **83**, 349 (2011).

[48] F. F. Assaad, in *DMFT at 25: Infinite Dimensions: Lecture Notes of the Autumn School on Correlated Electrons*, edited by E. Pavarini, E. Koch, D. Vollhardt, and A. Lichtenstein (Verlag des Forschungszentrum Jülich, Jülich, 2014), Vol. 4, Chap. 7. Continuous-time QMC Solvers for Electronic Systems in Fermionic and Bosonic Baths, iSBN 978-3-89336-953-9.

[49] F. F. Assaad and T. C. Lang, Phys. Rev. B **76**, 035116 (2007).

[50] R. T. Scalettar, D. J. Scalapino, and R. L. Sugar, Phys. Rev. B **34**, 7911 (1986).

[51] S. Beyl, F. Goth, and F. F. Assaad, Phys. Rev. B **97**, 085144 (2018).

[52] S. Dürr, Z. Fodor, J. Frison, C. Hoelbling, R. Hoffmann, S. D. Katz, S. Krieg, T. Kurth, L. Lellouch, T. Lippert, K. K. Szabo, and G. Vulvert, Science **322**, 1224 (2008).

[53] F. F. Assaad, in *Lecture notes of the Winter School on Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms.*, edited by J. Grotendorst, D. Marx, and A. Muramatsu. (Publication Series of the John von Neumann Institute for Computing, Jülich, 2002), Vol. 10, pp. 99–155.

[54] Y. Motome and M. Imada, Journal of the Physical Society of Japan **66**, 1872 (1997).

[55] F. F. Assaad, M. Imada, and D. J. Scalapino, Phys. Rev. B **56**, 15001 (1997).

[56] A. Sandvik, Phys. Rev. B **57**, 10287 (1998).

[57] K. S. D. Beach, eprint arXiv:cond-mat/0403055 (2004).

[58] R. M. Fye, Phys. Rev. B **33**, 6271 (1986).

[59] M. Iazzi and M. Troyer, Phys. Rev. B **91**, 241118 (2015).

[60] S. M. A. Rombouts, K. Heyde, and N. Jachowicz, Phys. Rev. Lett. **82**, 4155 (1999).

[61] E. Gull, P. Werner, O. Parcollet, and M. Troyer, EPL (Europhysics Letters) **82**, 57003 (2008).

[62] S. Rombouts, K. Heyde, and N. Jachowicz, Physics Letters A **242**, 271 (1998).

[63] D. Rost, E. V. Gorelik, F. Assaad, and N. Blümer, Phys. Rev. B **86**, 155109 (2012).

[64] D. Rost, F. Assaad, and N. Blümer, Phys. Rev. E **87**, 053305 (2013).

[65] N. Blümer, arXiv:0801.1222 (2008).

[66] F. Goth, arXiv:2009.04491 (2020).

[67] K. Hukushima and K. Nemoto, Journal of the Physical Society of Japan **65**, 1604 (1996).

[68] C. J. Geyer, in *Computing Science and Statistics: Proceedings of the 23rd Symposium on the Interface* (American Statistical Association, New York, 1991), pp. 156–163.

[69] C. W. Gardiner, *Handbook of Stochastic Methods* (Springer, ADDRESS, 1985).

[70] S. Beyl, Doctoral thesis, Universität Würzburg, 2020.

[71] Z. Wang, R. S. K. Mong, and F. F. Zaletel, Michael P. and. Assaad, arXiv:20**** .

[72] Z. Bai, C. Lee, R.-C. Li, and S. Xu, Linear Algebra and its Applications **435**, 659 (2011).

[73] A. van der Sluis, Numerische Mathematik **14**, 14 (1969).

[74] M. Feldbacher and F. F. Assaad, Phys. Rev. B **63**, 073105 (2001).

[75] D. Ixert, F. F. Assaad, and K. P. Schmidt, Phys. Rev. B **90**, 195133 (2014).

[76] J. W. Negele and H. Orland, *Quantum Many body systems*, *Frontiers in physics* (Addison-Wesley, Redwood City, Calif. u.a., 1988).

[77] W. Krauth, *Statistical Mechanics: Algorithms and Computations* (Oxford University Press, ADDRESS, 2006).

[78] C. J. Geyer, Statistical Science **7**, 473 (1992).

[79] R. M. Neal, (1993).

[80] M. Bercx, J. S. Hofmann, F. F. Assaad, and T. C. Lang, Phys. Rev. B **95**, 035108 (2017).

[81] B. Efron and C. Stein, Ann. Statist. **9**, 586 (1981).

[82] S. Chakravarty, B. I. Halperin, and D. R. Nelson, Phys. Rev. Lett. **60**, 1057 (1988).

[83] M. B. Thompson, arXiv:1011.0175 (2010).

[84] I. Milat, F. Assaad, and M. Sigrist, Eur. Phys. J. B **38**, 571 (2004), http://xxx.lanl.gov/cond-mat/0312450.

[85] M. Bercx, T. C. Lang, and F. F. Assaad, Phys. Rev. B **80**, 045412 (2009).

[86] A. Parola, S. Sorella, M. Parrinello, and E. Tosatti, Phys. Rev. B **43**, 6190 (1991).

[87] M. Vekić, J. W. Cannon, D. J. Scalapino, R. T. Scalettar, and R. L. Sugar, Phys. Rev. Lett. **74**, 2367 (1995).

[88] M. Raczkowski and F. F. Assaad, Phys. Rev. Research **2**, 013276 (2020).

[89] K. S. D. Beach, P. A. Lee, and P. Monthoux, Phys. Rev. Lett. **92**, 026401 (2004).

[90] S. Gazit, M. Randeria, and A. Vishwanath, Nat Phys **13**, 484 (2017).

[91] S. Gazit, F. F. Assaad, S. Sachdev, A. Vishwanath, and C. Wang, Proceedings of the National Academy of Sciences (2018).

[92] S. Gazit, F. F. Assaad, and S. Sachdev, arXiv:1906.11250 (2019).

[93] M. Hohenadler and F. F. Assaad, Phys. Rev. Lett. **121**, 086601 (2018).

[94] M. Hohenadler and F. F. Assaad, Phys. Rev. B **100**, 125133 (2019).

[95] A. Rüegg, S. D. Huber, and M. Sigrist, Phys. Rev. B **81**, 155118 (2010).

[96] A. Abendschein and F. F. Assaad, Phys. Rev. B **73**, 165119 (2006).

[97] F. F. Assaad, Nat Phys **10**, 905 (2014).

[98] A. N. Rubtsov, V. V. Savkin, and A. I. Lichtenstein, Phys. Rev. B **72**, 035122 (2005).

[99] Jülich Supercomputing Centre, Journal of large-scale research facilities **2**, A62 (2016).

License

The ALF code is provided as an open source software such that it is available to all and we hope that it will be useful. If you benefit from this code we ask that you acknowledge the ALF collaboration as mentioned on our homepage alf.physik.uni-wuerzburg.de. The git repository at alf.physik.uni-wuerzburg.de gives us the tools to create a small but vibrant community around the code and provides a suitable entry point for future contributors and future developments. The homepage is also the place where the original source files can be found. With the coming public release it was necessary to add copyright headers to our source files. The Creative Commons licenses are a good way to share our documentation and it is also well accepted by publishers. Therefore this documentation is licensed to you under a CC-BY-SA license. This means you can share it and redistribute it as long as you cite the original source and license your changes under the same license. The details are in the file license.CCBYSA that you should have received with this documentation. The source code itself is licensed under a GPL license to keep the source as well as any future work in the community. To express our desire for a proper attribution we decided to make this a visible part of the license. To that end we have exercised the rights of section 7 of GPL version 3 and have amended the license terms with an additional paragraph that expresses our wish that if an author has benefitted from this code that he/she should consider giving back a citation as specified on alf.physik.uni-wuerzburg.de. This is not something that is meant to restrict your freedom of use, but something that we strongly expect to be good scientific conduct. The original GPL license can be found in the file license.GPL and the additional terms can be found in license.additional. In favour to our users, the ALF code contains part of the lapack implementation version 3.6.1 from http://www.netlib.org/lapack. Lapack is licensed under the modified BSD license whose full text can be found in license.BSD.
With that being said, we hope that the ALF code will prove to you to be a suitable and high-performance tool that enables you to perform quantum Monte Carlo studies of solid state models of unprecedented complexity.

The ALF project's contributors.

COPYRIGHT