German Research School
for Simulation Sciences

Simulations of strongly correlated materials: Clusters and DMFT using a Lanczos solver

Master's Thesis

Mihaela Monica Bugeanu

January 2012

Supervisor
Prof. Dr. Erik Koch

Examiner
Prof. Dr. Erik Koch

Co-Examiner
Prof. Dr. Eva Pavarini

Contents

Introduction

In order to fully understand the properties of a material, one would acquire knowledge about the wave function of the system. This can be done by solving the many-body Hamiltonian

$$H = \sum_{\alpha=1}^{N_n} \frac{\vec{P}_\alpha^2}{2M_\alpha} + \sum_{j=1}^{N_e} \frac{\vec{p}_j^2}{2m} - \sum_{j=1}^{N_e}\sum_{\alpha=1}^{N_n} \frac{Z_\alpha e^2}{|\vec{r}_j - \vec{R}_\alpha|} + \sum_{j<k}^{N_e} \frac{e^2}{|\vec{r}_j - \vec{r_k}|} + \sum_{\alpha<\beta}^{N_n} \frac{Z_\alpha Z_\beta e^2}{|\vec{R}_\alpha - \vec{R}_\beta|}, \tag{0.1}$$

where for each nucleus α out of the total number N_n we have the atomic number Z_α, the mass M_α, the position $\vec{R}_\alpha$ and the momentum $\vec{P}_\alpha$ and for each of the N_e electrons we have the momentum $\vec{p}_j$, the mass m and the position $\vec{r}_j$. Although this would give us information about every property of the system, it is unfeasible to solve this Hamiltonian computationally.

Using the Born Oppenheimer approximation, we solve only the electronic Hamiltonian:

$$H = \sum_{j=1}^{N_e} \frac{\vec{p}_j^2}{2m} - \sum_{j=1}^{N_e}\sum_{\alpha=1}^{N_n} \frac{Z_\alpha e^2}{|\vec{r}_j - \vec{R}_\alpha|} + \sum_{j<k}^{N_e} \frac{e^2}{|\vec{r}_j - \vec{r_k}|}, \tag{0.2}$$

since the electrons move much faster than the nuclei. Moreover, we assume that the nuclei form a regular lattice.

Using a linear combination of atomic orbitals ansatz and second quantization we arrive at the Hubbard Hamiltonian:

$$H = \sum_{i,\sigma} \epsilon c_{i\sigma}^\dagger c_{i\sigma} - \sum_{\langle ij\rangle,\sigma} t_{ij} c_{i\sigma}^\dagger c_{j\sigma} + U \sum_i n_{i\uparrow} n_{i\downarrow}, \tag{0.3}$$

which consists of an on site energy ϵ, which can be seen as the interaction of the electron with the nuclei on the lattice. A kinetic term connecting two different sites, with the corresponding matrix element t_{ij}, which describes the movement of the electrons and the

potential energy U, which contains the Coulomb repulsion between electrons. In this case the interaction between electrons is considered local and only nearest-neighbour hopping has been introduced. Using this model the Hamiltonian is sparse and could more easily be solved numerically.

The two limiting cases for this model are represented by the band limit $U/t \ll 1$ when the hopping term takes precedence over the correlations and the problem can be treated as a system of free electrons. The other extreme case is the atomic limit, when $U/t \gg 1$, which means that the kinetic energy of the system can be neglected. In this case the potential energy will stop the electrons from occupying the same site.

Even if the Hubbard Hamiltonian is a simplification of the many body interaction term, it still has to be solved on an infinite lattice. This means an infinite dimensional Hilbert space, which cannot be solved using a computer. To be able to solve the Hamiltonian computationally we have to reduce it to a finite size.

To get a finite dimensional Hilbert space we can cut the lattice and solve the finite cluster that we get. The bigger the cluster, the closer we get to solving the infinite lattice. Choosing the cluster has to be done such that the finite size effects are reduced. This is done by choosing the proper shape and boundary conditions for the cluster. The Hilbert space is reduced to the configurations contained in the cell and via the boundary condition, copies of the cell are made to span the entire lattice. This approach is shown in Chapter 1, where we provide also an overview of the properties a cluster can have.

A modern alternative to finite clusters is the dynamical mean field theory (DMFT), that maps the entire lattice onto a single site coupled to a self consistent dynamic medium. This way we can account for the dynamics of the system, while solving a smaller Hamiltonian. In this case the parameters that characterize the impurity solver, the bath parameters have to be determined such that they emulate the actual system. This topic is covered in Chapter 4, in particular we discuss how the self consistency loop looks like. A specific method, the combined continued fraction method is shown in detail in Chapter 5.

After getting the finite size Hamiltonian we want to solve it. In Chapter 3 we present the Lanczos method, which is a direct method for getting the ground state information. It applies the Hamiltonian to a vector in order to find the ground state, again and again. This operation can be computationally expensive even for small systems, since the size of the Hilbert space increases exponentially, that is why we need an optimized implementation. We use a C++ implementation for the solver. The same method is used to calculate the spectral function used for the DMFT self consistency loop.

Both methods for reducing the Hamiltonian require solving a finite size Hamiltonian, however setting up the many body Hamiltonian is specific to every system. All need some basic blocks as the number of particles, the single particle Hamiltonian, the interaction parameters, which could be easily scripted. In Chapter 2 we introduce Lua, which is a small interpreted language that provides us with enough flexibility for that. The important quality of Lua is that it has a very small interpreter and can be embedded easily in C programs. That is why this is used for specific but not so costly operations, like setting

up the system, while the computational intensive parts of the program can still be done using a high level language that can be optimized and parallelized.

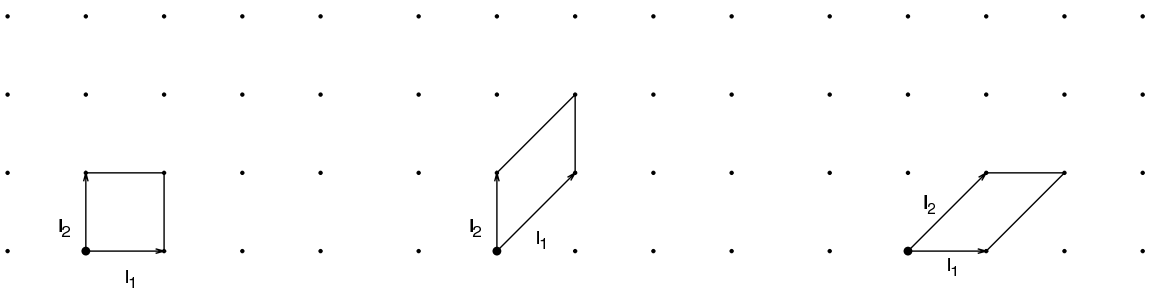
CHAPTER 1

Lattices

Since one cannot investigate any kind of infinite systems, we restrict ourselves in this thesis to studying crystal structures with inherent periodicity. If we want to describe a d-dimensional periodic structure we only need an origin and d linearly independent vectors that correspond to the periodicity of our lattice. For simplification we will consider a 2 dimensional (2D) lattice with defining vectors, $\vec{l}_1$ and $\vec{l}_2$ and the origin in $(0,0)$. Due to periodicity, we have the following relation $H(\vec{r}) = H(\vec{r}+m_1\cdot\vec{l}_1+m_2\cdot\vec{l}_2)$, with $m_1, m_2 \in Z$, for any starting position $\vec{r}$. This relation tells us that every integer linear combination of the lattice vectors results in an equivalent point on the crystal.

For a better understanding let us assume that our underlying lattice is defined by the lattice vectors in Eq. (1.1).

$$\vec{l}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \vec{l}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} . \tag{1.1}$$



(a) $\vec{l}_1 = (1,0)$, $\vec{l}_2 = (0,1)$ **(b)** $\vec{l}_1 = (1,1)$, $\vec{l}_2 = (0,1)$ **(c)** $\vec{l}_1 = (1,0)$, $\vec{l}_2 = (1,1)$

Figure 1.1.: Various choices for lattice vectors that span the same lattice

First we look into the choice of these lattice vectors. We can span the same lattice also by the vectors $\vec{l}_1' = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\vec{l}_2' = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ or the lattice vectors $\vec{l}_1'' = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\vec{l}_2'' = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. We can see, though, that the latter two sets of lattice vectors can be expressed as an integer linear combination of the first set, as follows $\vec{l}_1' = \vec{l}_1 + \vec{l}_2$, $\vec{l}_2' = \vec{l}_2$ and $\vec{l}_1'' = \vec{l}_1$, $\vec{l}_2'' = \vec{l}_1 + \vec{l}_2$. The Figure Fig. 1.1 illustrates these lattice vectors.

In order to have equivalent lattices we have to be able to express any point $n_1 \cdot \vec{l}_1 + n_2 \cdot \vec{l}_2$ in terms of the lattice vectors $\vec{l}_1'$ and $\vec{l}_2'$ or $\vec{l}_1''$ and $\vec{l}_2''$ respectively. We will now demonstrate that the lattice described by $\vec{l}_1$ and $\vec{l}_2$ is equivalent to the one described by $\vec{l}_1'$ and $\vec{l}_2'$. We make use of the transformation from one set of lattice vectors to the other:

$$\begin{array}{lcllcll} n_1 \cdot \vec{l}_1 + n_2 \cdot \vec{l}_2 & = & n_1' \cdot \vec{l}_1' + n_2' \cdot \vec{l}_2' & \Leftrightarrow & L \cdot (n_1\, n_2)^T & = & L' \cdot (n_1'\, n_2')^T \\ \vec{l}_1' = \vec{l}_1 + \vec{l}_2, & & \vec{l}_2' = \vec{l}_2 & \Leftrightarrow & L^T = C^T \cdot L'^T & \Leftrightarrow & L = L' \cdot C \\ L \cdot (n_1\, n_2)^T & = & L' \cdot C \cdot (n_1\, n_2)^T & \Leftrightarrow & (n_1'\, n_2')^T & = & C \cdot (n_1\, n_2)^T \end{array} \tag{1.2}$$

The conditions that follow for the C matrix in Eq. (1.2) are the following:

- to get from lattice L to L', matrix C has to be an integer matrix
- to get from lattice L' to L, matrix C^{-1} has to be an integer matrix

If we want to fulfill the last condition we get that it is sufficient if C is an integer matrix and $\det C = \pm 1$. Note that for convenience we can rearrange the lattice vectors, such that the determinant is positive.

1.1. Eigenvectors of lattices and the Brillouin zone

The Hamiltonian for a periodic lattice of Hydrogen nuclei and one electron is given by:

$$H = -\frac{1}{2}\nabla^2 - \sum \frac{1}{\left|\vec{r} - \vec{R}_i\right|}\,,\ \vec{R}_i = \sum_{n_1,n_2,\ldots,n_d} n_1\vec{l}_1 + n_2\vec{l}_2 + \ldots + n_d\vec{l}_d,$$

where $\vec{r}$ is the electron coordinate and $\vec{R}_i$ is the coordinate of the nuclei, positioned at a lattice point. We already know the Hamiltonian for one hydrogen atom and we want to use the eigenfunctions as a basis for our many site Hamiltonian:

$$H_i = -\frac{1}{2}\nabla^2 - \frac{1}{\left|\vec{r} - \vec{R}_i\right|}.$$

We denote the ground state eigenfunction of each site with φ_0^i and the eigenenergies ε_{at}. The eigenenergies are the same for every site since our system is composed out of identical atoms. The matrix elements for the many sites Hamiltonian with respect to these atomic orbitals are given by the relations

$$\begin{aligned}
\langle \varphi_0^i \left| H \right| \varphi_0^i \rangle &= \varepsilon_{at} - \sum_{\substack{j \\ j \neq i}} \langle \varphi_0^i \left| \frac{1}{|r - R_j|} \right| \varphi_0^i \rangle \quad &= \varepsilon_i, \\
\langle \varphi_0^k \left| H \right| \varphi_0^i \rangle &= -\sum_{\substack{j \\ j \neq i}} \langle \varphi_0^k \left| \frac{1}{|r - R_j|} \right| \varphi_0^i \rangle \quad &= -t_{ki}.
\end{aligned}$$

The energies ε_i are actually not dependent on the position due to the equivalence of the sites. The element t_{ki} describes the interaction between site k and site i. Since the hamiltonian H_i is hermitian, $H_i^\dagger = H_i$, the interaction between site k and site i becomes symmetric, $t_{ki} = t_{ik}$. We now make the tight binding approximation of having only nearest neighbour hopping and no overlap between φ_i and φ_j for $i \neq j$, which leads to a tridiagonal representation of the Hamiltonian as in Eq. (1.3).

$$H = \begin{pmatrix}
\ddots & \ddots & \ddots & \ddots & \ddots & & & & \\
 & 0 & -t & \varepsilon & -t & 0 & & & \\
 & & 0 & -t & \varepsilon & -t & 0 & & \\
 & & & 0 & -t & \varepsilon & -t & 0 & \\
 & & & & \ddots & \ddots & \ddots & \ddots & \ddots
\end{pmatrix}. \tag{1.3}$$

The solution is given by plane waves of the form $\psi(R_i) = e^{i\vec{k}\vec{R}_i}$, where R_i are the lattice sites. Transferring back to the original problem, we just have to multiply the solution found as a linear combination of atomic orbitals:

$$\psi_k(\vec{r}) = \sum e^{i\vec{k}_n \vec{R}_i} \varphi_0^i(\vec{r}).$$

We would now like to know what values we are allowed to choose for the $\vec{k}$-vectors. We look into the condition that the wave function at one specific (arbitrary) point in the lattice

should have the same value for every $\vec{k}$ and compute this way possible k-points.

$$\begin{aligned}
\psi_k(\vec{R}_i) &= \psi_{k+g}(\vec{R}_i) \\
e^{i\vec{k}\vec{R}_i} &= e^{i(\vec{k}+\vec{g})\vec{R}_i} \\
e^{i\vec{k}\vec{R}_i} &= e^{i\vec{k}\vec{R}_i} \cdot e^{i\vec{g}\cdot\vec{R}_i} \\
1 &= e^{i\vec{g}\cdot\sum_j (n_j\cdot\vec{l}_j)} \\
1 &= \prod_j e^{i\vec{g}\cdot(n_j\cdot\vec{l}_j)}, \forall n_j \in Z
\end{aligned}$$

for any lattice vector $\vec{l_j}$. Which results in the equation Eq. (1.4).

$$e^{i\vec{g}\vec{l}_j} = 1, \tag{1.4}$$

which does not have a unique solution. Any integer combination of $\vec{g}$ values that fulfills the Eq. (1.4) also is a solution of that equation, leading to infinite many k-values. Assuming we have two $\vec{k}$ values and an integer linear combination of them. We check now if this linear combination fulfills the periodicity condition:

$$e^{i(a_1\vec{k}_1+a_2\vec{k}_2)\cdot\vec{l}_j} = e^{ia_1\vec{k}_1\cdot\vec{l}_j} e^{ia_2\vec{k}_2\cdot\vec{l}_j} = 1\,,\ a_1, a_2 \in Z. \tag{1.5}$$

This leaves us with the question of choosing the $\vec{k}$ values properly. In order to find linearly independent $\vec{k}$ values we look only for values in the first period of the trigonometric functions, which then give rise to the shortest k-values possible.

To this end we can write one matrix equation connecting the $\vec{l}$ vectors to the $\vec{k}$ vectors, Eq. (1.6), for simplicity illustrated on a 2D example.

$$\begin{aligned}
L^T \cdot K &= 2\pi I \\
\begin{pmatrix} \vec{l}_1 & \vec{l}_2 \end{pmatrix}^T \begin{pmatrix} \vec{k}_1 & \vec{k}_2 \end{pmatrix} &= 2\pi I \\
\begin{pmatrix} \vec{l}_1\cdot\vec{k}_1 & \vec{l}_1\cdot\vec{k}_2 \\ \vec{l}_2\cdot\vec{k}_1 & \vec{l}_2\cdot\vec{k}_2 \end{pmatrix} &= \begin{pmatrix} 2\pi & 0 \\ 0 & 2\pi \end{pmatrix}
\end{aligned} \tag{1.6}$$

Brillouin zone. The system in Eq. (1.6) has a unique solution for a given set of independent lattice vectors. The unit cell in $\vec{k}$-space spanned by these vectors is called the Brillouin zone of the lattice and the $\vec{k}$ values are called reciprocal vectors to the original lattice. Any $\vec{k}$ value outside of this interval can be mapped back into the Brillouin zone using integer linear combinations of the reciprocal lattice vectors.

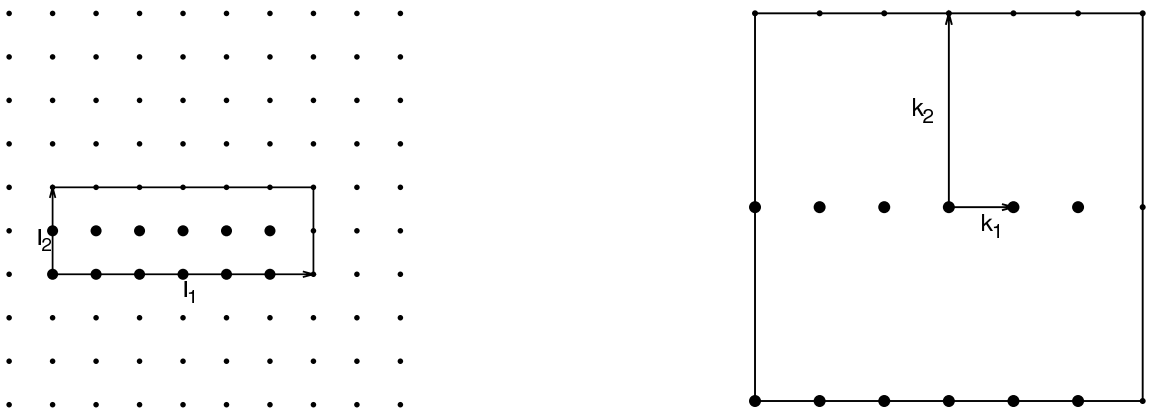
For example in our simple 2D lattice case, defined by the unit vectors $\vec{l}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\vec{l}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, the reciprocal lattice cell would be $[0, +2\pi) \times [0, +2\pi)$. If we look at the k-value $\vec{k}_1 = 3\pi$ this can be mapped back to the Brillouin zone (BZ) by just subtracting 2π, $\psi_{k_1}(r) = \psi_{k_2}(r)$, with $k_2 = k_1 - 2\pi = \pi$. Since most of the systems that we study are inversion symmetric, it is convenient to shift the Brillouin zone to $[-\pi, +\pi) \times [-\pi, +\pi)$, in order to make it also symmetric.

1.2. Super-Cells

Solving the non-interacting part of the Hamiltonian for the infinite system is possible, but for completely understanding the properties of our crystal structure, we have to also take into account the interaction between electrons. If we include the interaction between electrons, solving the infinite problem is no longer possible, in general. We therefore solve finite systems with a properly chosen shape of the lattice vectors and boundary conditions, such that an extrapolation to infinite size should have results that lie close to the infinite solution. In this section we discuss the choice of the lattices, while the boundary conditions will be discussed later.

1.2.1. Points and Eigenvalues

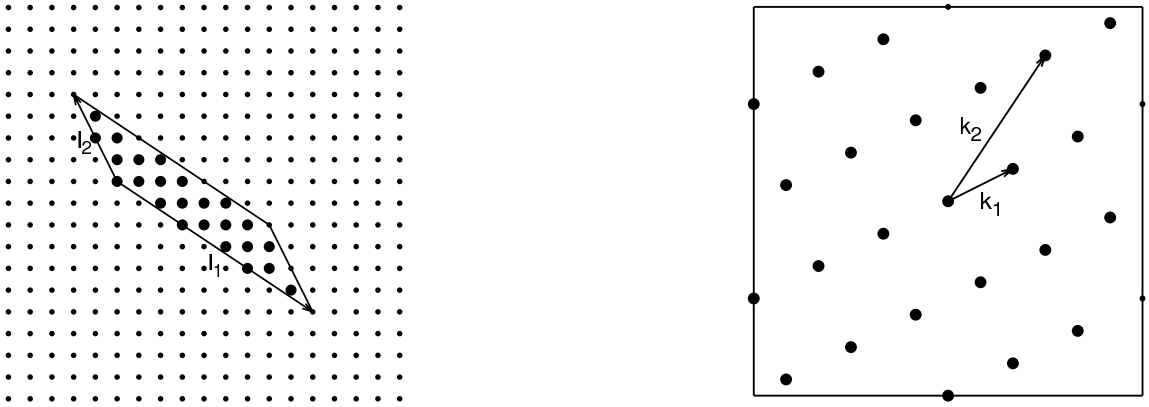
Using integer linear combinations of the two basis vectors of the lattice, we construct two new vectors that span another lattice. Using these lattices we solve our many-body problem. The cell spanned by the new vectors is called a super-cell and the vectors are called super-cell vectors.



(a) Super-cell Points **(b)** K-Points

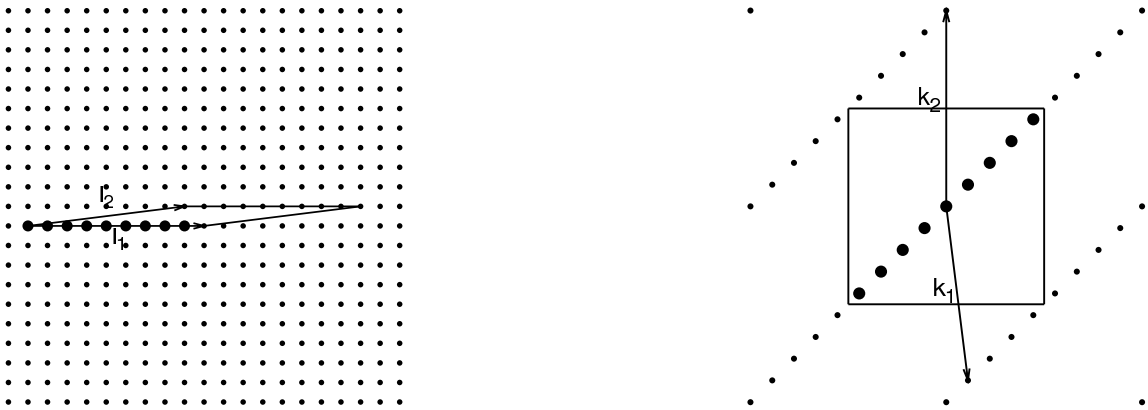
Figure 1.2.: $\vec{l}_1 = (6,0)^T$, $\vec{l}_2 = (0,2)^T$

In the Fig. 1.2, Fig. 1.3 and Fig. 1.4 we show some basic examples of super-cells. On the left we can see the underlying lattice points, the two defining vectors of the super-cell and highlighted the points of the lattice that lie within one super-cell. On the right in each figure, we can see the reciprocal vectors of the super-cell and the $\vec{k}$ values that are within the first (centered) Brillouin zone of the underlying lattice. We can observe that, while the super-cell contains more than one lattice point, the Brillouin zone of the lattice contains more than one $\vec{k}$ value of the super-lattice. We will show below that the number of points in the super-cell and the number of $\vec{k}$ points in the Brillouin zone of the lattice is actually the same, but first we have to introduce some basic notations found in Tab. 1.1 for the matrices involved.



(a) Super-cell Points **(b)** K-Points

Figure 1.3.: $\vec{l_1} = (9, -6)^T$, $\vec{l_2} = (-2, 4)^T$



(a) Super-cell Points **(b)** K-Points

Figure 1.4.: $\vec{l_1} = (9, 0)^T$, $\vec{l_2} = (8, 1)^T$

L	=	$(l_1\, l_2)$	the matrix which contains the lattice vectors, as columns
L_{sc}	=	$(sc_1\, sc_2)$	the matrix which contains the super-cell vectors, as columns
K	=	$(k_1\, k_2)$	the K matrix which contains the lattice $\vec{k}$ vectors, as columns
K_{sc}	=	$(sk_1\, sk_2)$	the K matrix which contains the super-lattice $\vec{k}$ vectors, as columns
C	=	$(c_1\, c_2)$	the transformation matrix from the lattice to the super-lattice vectors $C^T \cdot L^T = L_{sc}^T$
V_l	=	$\lvert\det L\rvert$	the volume of the unit cell of the lattice
V_{sc}	=	$\lvert\det L_{sc}\rvert$	the volume of the unit cell of the super-lattice
V_{Kl}	=	$\lvert\det K\rvert$	the volume of the Brillouin zone of the lattice
V_{Ksc}	=	$\lvert\det K_{sc}\rvert$	the volume of the Brillouin zone of the super-lattice

Table 1.1.: Notation used in examples

1.2.2. Equivalent super-lattices

Building super-cells, one could also think about whether two super-cells could be equivalent, like in the lattice case described at the beginning of the chapter. We have the following two images to support this case. The super-cells in Fig. 1.5 are equivalent, because they both span the same super-lattice in real space and thus have also the same points in $\vec{k}$-space. First we will show that they span the same super-lattice, the $\vec{k}$-space lattice will then implicitly follow. We compute the transformation matrix, C from one set of super-cell lattice vectors to the other and check its determinant, like in Eq. (1.2). The transformation is given by the equation:

$$C^T \cdot L_{sc1}^T = L_{sc2}^T$$
$$\begin{pmatrix} 0 & 1 \\ -1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 4 & 0 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 0 & 8 \end{pmatrix}.$$

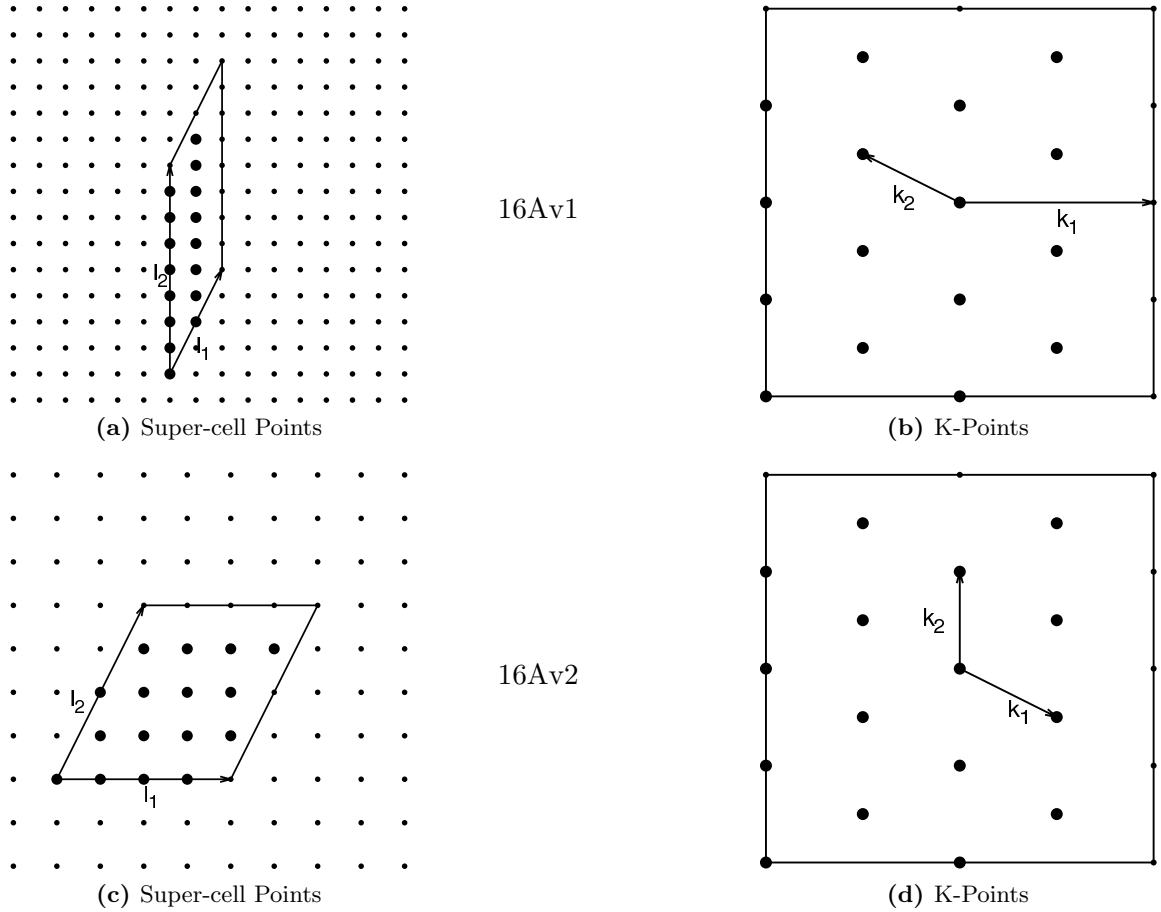
Computing the determinant of the transformation matrix C, we see that it is equal to 1, which tells us that the transformation keeps the lattice spanned equivalent. Now for the relation between the K matrices of the two equivalent lattices we use the equation Eq. (1.6):

$$L_{sc1}^T \cdot K_{sc1} = 2\pi I$$
$$L_{sc2}^T \cdot K_{sc2} = 2\pi I$$
$$C^T \cdot L_{sc1}^T = L_{sc2}^T$$

$$C^T \cdot L_{sc1}^T \cdot K_{sc1} = 2\pi C^T$$
$$L_{sc2}^T \cdot K_{sc1} \cdot (C^T)^{-1} = 2\pi I$$

$$K_{sc1} \cdot (C^T)^{-1} = K_{sc2}, \tag{1.7}$$

which gives us a unitary transformation between the two K matrices. Note that equation Eq. (1.7) can be used with any transformation matrix C, not only unitary ones.



(a) Super-cell Points **(b)** K-Points

(c) Super-cell Points **(d)** K-Points

Figure 1.5.: Super cells and their k-points for 16A
Top: $\vec{l_1} = (2,4)^T$, $\vec{l_2} = (0,8)^T$
Bottom: $\vec{l_1} = (4,0)^T$, $\vec{l_2} = (2,4)^T$

1.2.3. Volume of super-cells

We now look into the relation of the volume of the super-cell with respect to the volume of the lattice cell. The equation that connects the super-cell to the lattice is:

$$C^T L^T = L_{sc}^T.$$

The volume of the super-lattice unit cell is defined by:

$$V_{sc} = \left|\det\left(L_{sc}^T\right)\right| = \left|\det\left(C^T \cdot L^T\right)\right| = \left|\det\left(C^T\right) \cdot \det\left(L^T\right)\right| = \left|\det C^T\right| \cdot V_l = c \cdot V_l. \quad (1.8)$$

Using the relation derived in Eq. (1.7), we can compute the volume of the Brillouin zone of the super-lattice with respect to the one of the underlying lattice.

$$K_{sc} = K \cdot (C^T)^{-1} = \left|\det K_{sc}^T\right| = \left|\det\left(K \cdot (C^T)^{-1}\right)^T\right| = \left|\det\left((C^T)^{-1}\right)^T \cdot \det\left(K\right)^T\right|$$
$$= \left|\det\left((C^T)^{-1}\right)^T\right| \cdot V_{Kl} = c^{-1} \cdot V_{Kl}. \tag{1.9}$$

Eq. (1.8) and Eq. (1.9) show us that the volume spanned by the super-cell $\vec{k}$ vectors is smaller than the original one by the exact same factor as the volume of the super-cell is bigger than the lattice volume. This relation tells us that the number of $\vec{k}$ points of the super-cell that lie within the Brillouin zone of the lattice is the same as the number of sites of the lattice that lie within the unit cell of the super-lattice.

1.2.4. Properties of super-cells

In order to asses the quality of the super-cells we use the properties and grading system introduced in [**?**], [**?**].

Squareness. The squareness of a super-cell is defined as the parameter

$$\sigma = \sqrt{\frac{2l_1 l_2}{d_1 d_2}},$$

where l_1, l_2 are the length of the tile and d_1, d_2 its diagonals. Note, that the definition given here is different from the one in [**?**], including the square root to make the squareness a dimensionless unit of length. Let's take a look at the two equivalent lattices in Fig. 1.5. Using the top configuration we get $\sigma(16Av1) = 1.05$, while using the bottom configuration we get $\sigma(16Av2) = 1.14$. Meaning that although the lattices are equivalent, the squareness is not unique. If we take into account all unitary transformations and define the squareness as the minimum over all σ of equivalent lattices, we get a well defined value. This is the value that we will refer from now on. Obviously the squareness of a square tile is $\sigma = 1$ and tiles that have a squareness of $0.95 \leq \sigma \leq 1.05$ are considered good enough, [**?**]. However, if a lattice has squareness $\sigma \approx 1$ this does not necessarily mean that it is a square or close to a square. In Fig. 1.6 we have a lattice that has a squareness of $\sigma = 0.99$ and we can see that it doesn't resemble a square at all. The squareness does not quite measure what its name suggests.

Bipartite lattices. If the lattice points can be divided into two equivalent sub-lattices and if hopping to the nearest neighbour is always done from one sub-lattice to the other, we call the lattice bipartite. The square lattice in Fig. 1.7a is bipartite: the full circles belong to sub-lattice A, open circles to sub-lattice B. We would like to use bipartite lattices because these best fit the anti-ferromagnetic properties of our system. Electrons with spin up and spin down will arrange themselves such that neighbouring spins will have opposite orientation. The underlying infinite square lattice is bipartite, only the super-lattice vectors can alter this state. The super-lattice vectors should only connect points of the same sub-lattice, otherwise our super-cell is not bipartite any more.

A super-lattice described by $\vec{l_1}, \vec{l_2}, \ldots, \vec{l_d}$ is bipartite if it fulfills the following condition:

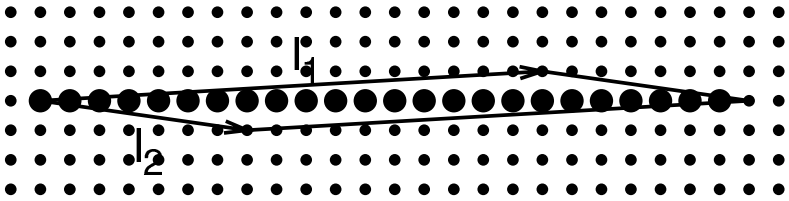
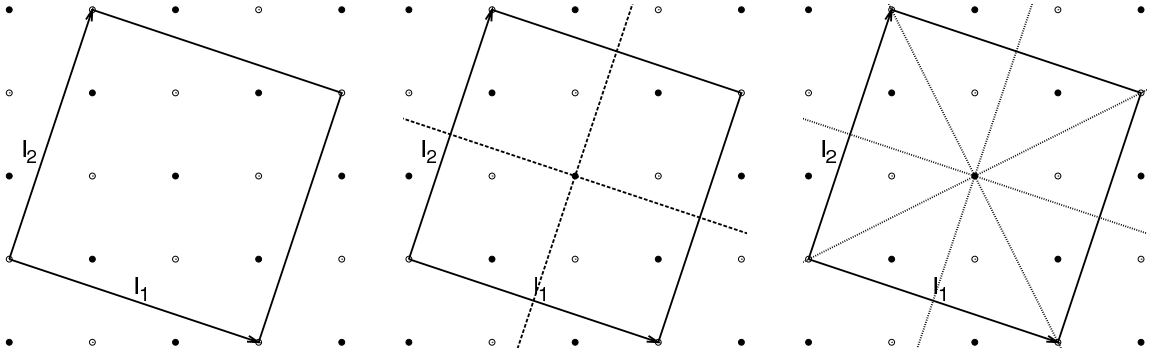


Figure 1.6.: Squareness close to one, although not square.
$\sigma = 0.99$, $\vec{l}_1 = (1, 7)^T$, $\vec{l}_2 = (-1, 17)^T$

- the sum of the elements of each lattice vector is even, $\sum_{i=1}^{d} l_j(i) = \text{even}, \forall j = 1, .., d$



(a) Example of bipartite lattice.

(b) D_4 - 4 rotation axes.

(c) D_4 - 4 reflection axes.

Figure 1.7.: Bipartite lattice and symmetries of square super-cell $\vec{l}_1 = (3, -1)^T$, $\vec{l}_2 = (1, 3)^T$, which keeps the following properties.(a) **bipartite**: the lattice keeps the bipartite properties of the underlying lattice, (b) **D4-rotations**: the lattice has also 4 rotation angles under which it remains the same $\alpha = \frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi$, (c) **D4-reflections**: mirror planes for the lattice

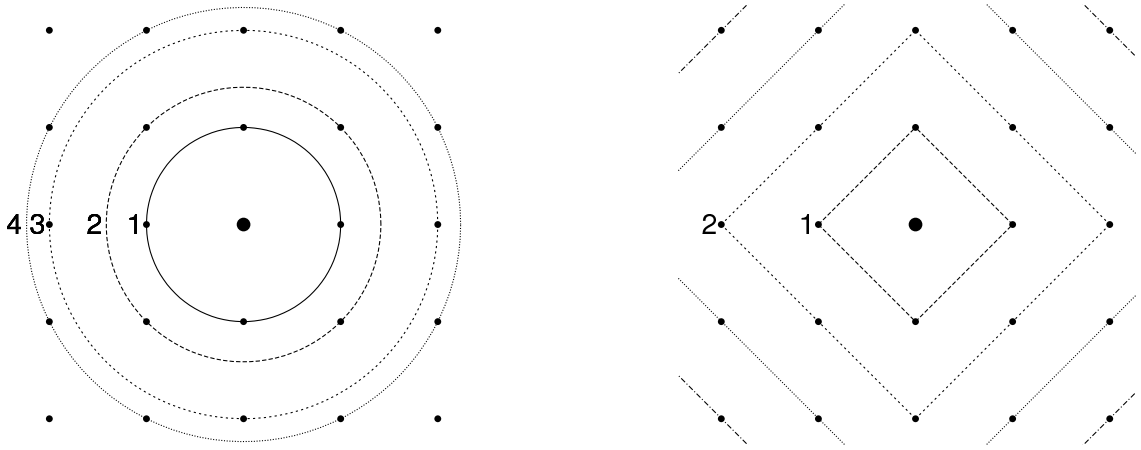
Symmetries. The main group of symmetries are *group* D_n, dihedral symmetries and *group* C_n, cyclic symmetries. Cyclic symmetries remain invariant under n rotations. The angle of each rotation is $\alpha = 360/n$. Dihedral symmetries have besides the rotational symmetry also reflection symmetry. An object having D_n symmetry would then have n rotations with angles of $\alpha = i * 360/n, i = 0, 1, 2, 3, \ldots, n-1$ and n reflection axes, each with angle $\beta = 180/n$ between them. A simple example is the 4×4 lattice. It has 4 rotation axes, of each 90°, as seen in Fig. 1.7b and 4 reflection axes separated by an angle of 45°, as in Fig. 1.7c.

Imperfection. In order to define the imperfection we first have to introduce the nearest neighbour levels. We differentiate between two ways of computing the distance between two points. We can use the geometrical interpretation with the Euclidean distance resulting in circles of nearest neighbours, as in Fig. 1.8a and the topological interpretation, where the Manhattan distance is used, resulting in nearest neighbour squares, as in Fig. 1.8b. The difference becomes clear when we want to count the neighbours on the different levels.

The result can be seen in Tab. 1.2. A "perfect" infinite lattice, would then have complete sets of nearest neighbours, indepdendent of the distance used. In the finite lattice case this cannot happen because our lattice cell has a certain shape and is finite. In this case we define a "perfect" lattice as a lattice that has complete nearest neighbour sets until the last one, which can be incomplete. To measure the imperfection we allow for neighbours to be artificially moved from the outer most level to the inner most incomplete level, one level at a time until we generate a "perfect" structure. The number of moves we make is the **ferromagnetic imperfection**. Thus a perfect lattice has the imperfection $I_F = 0$. Note that when actually computing the imperfection, one has to take the closest distance into account, including to copies of the cell.

For bipartite lattices another type of imperfection might be interesting, the **antiferromagnetic imperfection**, I_B, taking into account the fact that the lattice can be split up in two sub-lattices. We compute the imperfection individually on each sub-lattice. In these terms, the imperfection of a bipartite lattice can be better than the ferromagnetic imperfection. This can also be computed using the geometrical and the topological nearest neighbours.

We illustrate in table Tab. 1.3 a summary of the properties for the lattices found in Fig. 1.9.



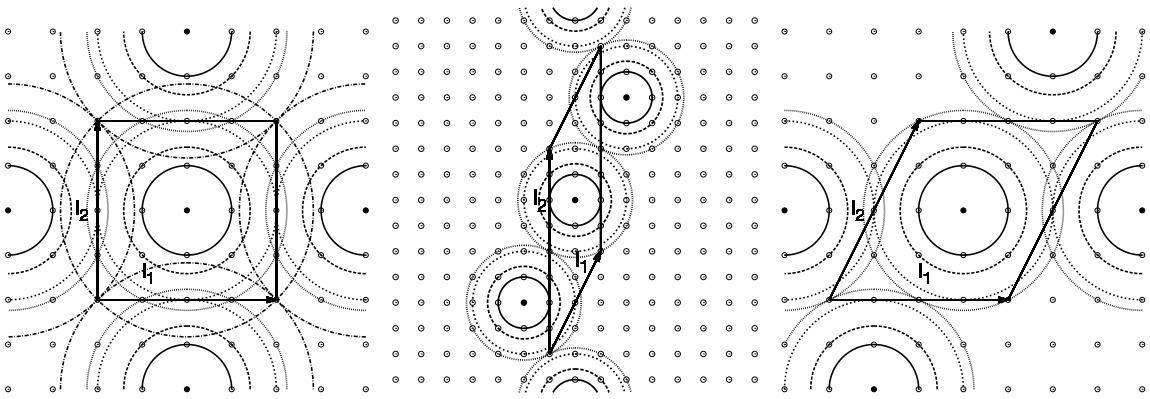
(a) Geometrical nearest neighbours.

(b) Topological nearest neighbours.

Figure 1.8.: Nearest neighbour descriptions. (a) the geometrical nearest neighbour rings and (b) the topological nearest neighbour boxes.

Level	Euclidean distance	Manhattan distance
1	4	4
2	4	8
3	4	12
4	8	16
5	4	20

Table 1.2.: Number of neighbours per level for the euclidean and manhattan distance.



(a) Square lattice, 16B

(b) Parallelogram lattice, 16A, version 1

(c) Parallelogram lattice, 16A, version 2

Figure 1.9.: Nearest neighbour rings (geometrical distance) for computing the imperfection using the distance to the nearest image of the origin for 3 lattices, (a) square lattice 16B, $\vec{l}_1 = (4,\, 0)^T$, $\vec{l}_2 = (0,\, 4)^T$, (b) square lattice 16Av1, $\vec{l}_1 = (4,\, 0)^T$, $\vec{l}_2 = (2,\, 4)^T$, (c) square lattice 16Av2, $\vec{l}_1 = (2,\, 4)^T$, $\vec{l}_2 = (0,\, 8)^T$

1.3. Boundary condition

In order to simulate infinite lattices using a finite cluster of sites, described by our lattice vectors $\vec{l}_i$ we have to define how we can compute the wave-function of the lattice using the wave-function of the cluster. This is done using boundary conditions. Our goal is to find boundary conditions that minimize the surface effects, such that the solution on the finite cluster is as close as possible to the one on the infinite lattice. In this chapter we will first provide definitions for the common boundary conditions: open, periodic, anti-periodic and complex boundary conditions. After the short definitions we will present the influence of the different boundary conditions on the solution on the infinite lattice.

Open Boundary Condition. In this case, we cut out the cluster and try to simulate the behavior of the system with only the reduced particles in the cluster, thus our wave function outside the cluster is set to zero:

$$\psi(\vec{r} + \vec{l}_i) = 0. \tag{1.10}$$

Periodic Boundary Condition. In this case, the lattice is covered with exact copies of the cluster. All information needed for the solution on the infinite lattice, is contained in one copy of the cluster. This information is then copied to all other copies according to the following equation:

$$\psi(\vec{r} + \vec{l}_i) = \psi(\vec{r}). \tag{1.11}$$

Anti Periodic Boundary Condition. In the case of anti-periodic boundary conditions, only one copy of the cluster does not contain all the information of the wave-function of

the lattice. However there is a relation that connects the information on one copy of the cluster to the information of the other copies. This way we still need only one cluster and extrapolate to all other copies as follows:

$$\psi(\vec{r}+\vec{l_i}) = -\psi(\vec{r}). \tag{1.12}$$

Complex Boundary Condition. After introducing the most common boundary conditions, we generalize and say that a number n of copies of the cluster contain all the information of the wave-function. This can be described by the eqation:

$$\psi(\vec{r}+\vec{l_i}) = e^{i\varphi}\psi(\vec{r}). \tag{1.13}$$

where $\varphi = \frac{2m\pi}{n}$. Using the above equation one can extrapolate again from the information contained in only one copy of the cluster the entire information of the wave-function.

A small example that illustrates different boundary conditions can be seen in Fig. 1.10. The solutions illustrated are for a 4 site, 1D chain. We see on the first row the solution of the open boundary problem. Outside the cluster the value of our wave-function is zero. The next line illustrates the periodic boundary condition, where the entire information of the wave function is contained in one copy of the cluster, this information is then copied at all other sites of the cluster. The next line describes the anti-periodic boundary conditions, where the information of two clusters are needed to reconstruct all the information of the lattice. The last line shows a more general case, where 3 copies of the cluster are required until the information is repeated.

1.3.1. Solution of the finite problem

In order to solve the finite cluster with boundary conditions one has to take into consideration the specific boundary condition used in the determination of the Hamiltonian. We will use again the the tight binding model that allows us to build a tridiagonal Hamiltonian. The Hamiltonian for the infinite system has been introduced in Eq. (1.3). We will take a look at the same system used in Fig. 1.10: 1D, 4 sites, $\vec{l} = 4$. We will first discuss the construction of the general Hamiltonian for the finite size system and take a look at the $\vec{k}$-values for different boundary conditions.

Open boundary condition. As described in Eq. (1.10) in this case we are only considering the cluster itself, cutting to zero anything outside of our system. If we start by the

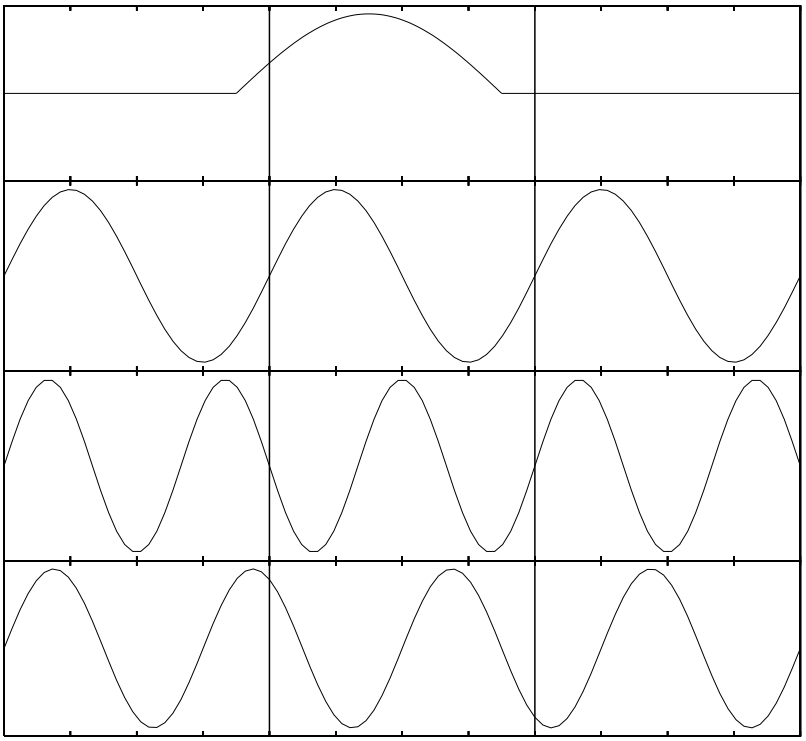


Figure 1.10.: Influence of boundary conditions on the solution of a 4 site problem (from top to bottom): open boundary condition (infinite lattice restricted to one cluster), periodic boundary condition(one copy of the cluster that repeats itself over the lattice), anti-periodic boundary condition (the periodicity comes after 2 copies of the cluster) and a more general case (the periodicity comes after 3 copies of the cluster).

infinite size Hamiltonian we get:

$$
\begin{aligned}
H\psi &= \begin{pmatrix}
\ddots & \ddots & \ddots & \ddots & \ddots & & & & \\
 & 0 & -t & \varepsilon & -t & 0 & & & \\
 & & 0 & -t & \varepsilon & -t & 0 & & \\
 & & & 0 & -t & \varepsilon & -t & 0 & \\
 & & & & 0 & -t & \varepsilon & -t & 0 \\
 & & & & & \ddots & \ddots & \ddots & \ddots & \ddots
\end{pmatrix} \cdot \begin{pmatrix}
\vdots \\ 0 \\ \psi_{i_0} \\ \psi_{i_0+1} \\ \psi_{i_0+2} \\ \psi_{i_0+3} \\ 0 \\ \vdots
\end{pmatrix} \\
&= \begin{pmatrix}
\vdots \\
\varepsilon\psi_{i_0} - t\psi_{i_0+1} \\
-t\psi_{i_0} + \varepsilon\psi_{i_0+1} - t\psi_{i_0+2} \\
-t\psi_{i_0+1} + \varepsilon\psi_{i_0+2} - t\psi_{i_0+3} \\
-t\psi_{i_0+2} + \varepsilon\psi_{i_0+3} \\
\vdots
\end{pmatrix} .
\end{aligned}
$$

In this case we can comprise the Hamiltonian and the wave function as:

$$H = \begin{pmatrix} \varepsilon & -t & 0 & 0 \\ -t & \varepsilon & -t & 0 \\ 0 & -t & \varepsilon & -t \\ 0 & 0 & -t & \varepsilon \end{pmatrix}, \psi(\vec{r}) = \begin{pmatrix} \psi_{i_0} \\ \psi_{i_0+1} \\ \psi_{i_0+2} \\ \psi_{i_0+3} \end{pmatrix}.$$

If we make our Bloch wave ansatz $\psi(\vec{r}) = c_1 e^{ik\vec{r}} + c_2 e^{-ik\vec{r}}$ we will get the conditions for the boundaries:

$$\begin{aligned} &\psi(x_0) = 0 \Leftrightarrow c_1 e^{ikx_0} + c_2 e^{-ikx_0} = 0 \Leftrightarrow c_1 + c_2 = 0 \Leftrightarrow c_1 = -c_2 \\ &\psi(x) = c\left(e^{ikx} - e^{-ikx}\right) = i\,c\,sin(kx) \\ &\psi(x_4) = i\,c\,sin(kx_4) = 0 \Leftrightarrow sin(kx_4) = 0 \\ &k_n = \frac{\pi n}{l}, n = 0, 1, 2, \ldots, l-1 \Leftrightarrow k \in \left\{\frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi\right\}. \end{aligned}$$

Finite size Hamiltonian with boundary conditions. We will look at complex boundary conditions $\psi(\vec{r}+\vec{l_i}) = e^{i\varphi}\psi(\vec{r})$ for 4 sites. The infinite size Hamiltonian as in Eq. (1.3) is multiplied with the wave function:

$$\begin{aligned} H\psi &= \begin{pmatrix} \ddots & \ddots & \ddots & \ddots & \ddots & & & & \\ & 0 & -t & \varepsilon & -t & 0 & & & \\ & & 0 & -t & \varepsilon & -t & 0 & & \\ & & & 0 & -t & \varepsilon & -t & 0 & \\ & & & & 0 & -t & \varepsilon & -t & 0 \\ & & & & & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix} \cdot \begin{pmatrix} \vdots \\ \psi_{i_0-1} \\ \psi_{i_0} \\ \psi_{i_0+1} \\ \psi_{i_0+2} \\ \psi_{i_0+3} \\ \psi_{i_0+4} \\ \vdots \end{pmatrix} \\ &= \begin{pmatrix} \vdots \\ -t\psi_{i_0-1} + \varepsilon\psi_{i_0} - t\psi_{i_0+1} \\ -t\psi_{i_0} + \varepsilon\psi_{i_0+1} - t\psi_{i_0+2} \\ -t\psi_{i_0+1} + \varepsilon\psi_{i_0+2} - t\psi_{i_0+3} \\ -t\psi_{i_0+2} + \varepsilon\psi_{i_0+3} - t\psi_{i_0+4} \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ -te^{-i\varphi}\psi_{i_0+3} + \varepsilon\psi_{i_0} - t\psi_{i_0+1} \\ -t\psi_{i_0} + \varepsilon\psi_{i_0+1} - t\psi_{i_0+2} \\ -t\psi_{i_0+1} + \varepsilon\psi_{i_0+2} - t\psi_{i_0+3} \\ -t\psi_{i_0+2} + \varepsilon\psi_{i_0+3} - te^{i\varphi}\psi_{i_0} \\ \vdots \end{pmatrix} \end{aligned}$$

The Hamiltonian can be mapped to a 4 by 4 matrix that contains all the interactions in one copy of the cluster. This copy can then be multiplied by the boundary condition and you can get every other combination of 4 sites of the infinite lattice:

$$H = \begin{pmatrix} \varepsilon & -t & 0 & -te^{-i\varphi} \\ -t & \varepsilon & -t & 0 \\ 0 & -t & \varepsilon & -t \\ -te^{i\varphi} & 0 & -t & \varepsilon \end{pmatrix}.$$

In order to solve this problem we make the Ansatz of a Bloch wave $\psi(\vec{r}) = e^{ik\vec{r}}$, the solution for the infinite lattice and we restrict this function to fulfill the boundary conditions:

$$\begin{aligned}&\psi(x_{i+4}) = e^{i\varphi}\psi(x_i) \Leftrightarrow e^{ik(x_{i+4})} = e^{i\varphi} \cdot e^{ikx_i} \Leftrightarrow e^{ikx_i} \cdot e^{ik\vec{l}} = e^{i\varphi} \cdot e^{ikx_i} \Leftrightarrow e^{ik\vec{l}} = e^{i\varphi} \\ &k = (2\pi n - \varphi)/\vec{l}, n \in Z.\end{aligned} \tag{1.14}$$

After solving the general equation we now take a look at more particular boundary conditions.

Periodic boundary condition. $\varphi = 0$**.** Using Eq. (1.11) we will construct the Hamiltonian for periodic boundary conditions, finite size problem:

$$H = \begin{pmatrix} \varepsilon & -t & 0 & -t \\ -t & \varepsilon & -t & 0 \\ 0 & -t & \varepsilon & -t \\ -t & 0 & -t & \varepsilon \end{pmatrix}.$$

This Hamiltonian will lead to the solutions as given in Eq. (1.14):

$$k_n = \frac{2\pi n}{l}, n = 0, 1, 2, \ldots, l-1 \Leftrightarrow k \in \left\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\right\}. \tag{1.15}$$

Anti-periodic boundary condition. $\varphi = \pi$**.** In this case the wave function changes its phase while stepping over the boundary, as described in Eq. (1.15), which leads to the Hamiltonian:

$$H = \begin{pmatrix} \varepsilon & -t & 0 & t \\ -t & \varepsilon & -t & 0 \\ 0 & -t & \varepsilon & -t \\ t & 0 & -t & \varepsilon \end{pmatrix}.$$

After solving the system we get the solutions:

$$k_n = \frac{(2\pi n - \pi)}{l}, n = 1, 2, \ldots, l \Leftrightarrow k \in \left\{\frac{\pi}{4}, \frac{3\pi}{4}, \frac{5\pi}{4}, \frac{7\pi}{4}\right\}.$$

Complex boundary condition. $\varphi = \frac{2\pi}{3}$**.** In this case the wave function changes its phase while stepping over the boundary in a more general way, which leads to the Hamiltonian:

$$H = \begin{pmatrix} \varepsilon & -t & 0 & -te^{-i\frac{2\pi}{3}} \\ -t & \varepsilon & -t & 0 \\ 0 & -t & \varepsilon & -t \\ -te^{i\frac{2\pi}{3}} & 0 & -t & \varepsilon \end{pmatrix}.$$

After selecting the bloch waves that fulfill the boundary condition we get:

$$k_n = \frac{(2\pi n - \frac{2\pi}{3})}{l}, n = 1, 2, \ldots, l \Leftrightarrow k \in \left\{ \frac{2\pi}{6}, \frac{5\pi}{6}, \frac{8\pi}{6}, \frac{11\pi}{6} \right\}.$$

Generalization 1D boundary condition. For the complex boundary condition with a given φ as boundary condition, we have the Hamiltonian:

$$H = \begin{pmatrix} \varepsilon & -t & 0 & -te^{-i\varphi} \\ -t & \varepsilon & -t & 0 \\ 0 & -t & \varepsilon & -t \\ -te^{i\varphi} & 0 & -t & \varepsilon \end{pmatrix}.$$

which leads to the $\vec{k}$-values given by:

$$k_n = \frac{(2\pi n - \varphi)}{l}, n = 1, 2, \ldots, l.$$

This leads to the conclusion that a boundary condition is just a shift in the $\vec{k}$ values of the periodic problem. In Fig. 1.11 we can see an illustration of this phase shift that occurs when using boundary conditions for a 1D problem.

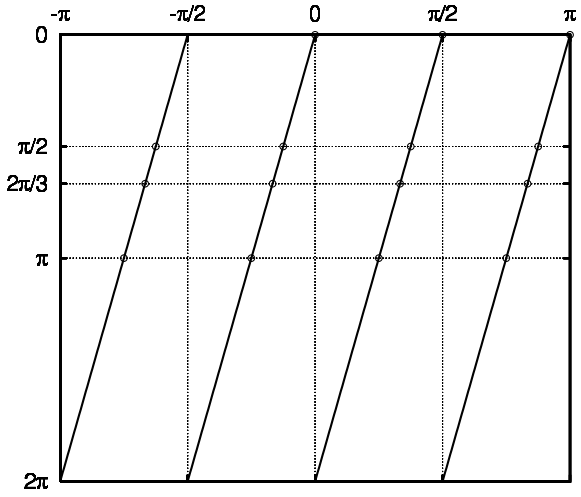


Figure 1.11.

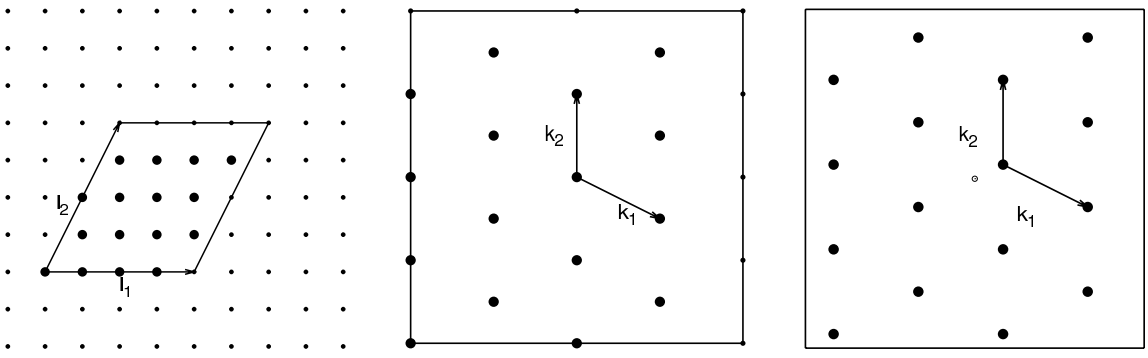
Generalization 2D boundary condition. For the 2D case we just have to consider both lattice vectors when we compute the $\vec{k}$-values. We will take the reciprocal vectors defined in Eq. (1.6), and apply an appropriate phase shift:

$$k_{nm\varphi} = \left(n - \frac{\varphi_1}{2\pi}\right) \cdot \vec{k}_1 + \left(m - \frac{\varphi_2}{2\pi}\right) \cdot \vec{k}_2,\ n, m \in Z.$$

An example of such a lattice can be seen in Fig. 1.12. It is the same cluster as seen earlier in Fig. 1.5. The boundary condition used is first periodic then a complex boundary condition

for a $\varphi = \frac{2\pi}{3}$ in each direction. We notice that the only difference to the periodic case is a shift of all the $\vec{k}$ values of our cluster.

Note that the eigenvectors and the eigenvalues have the same expression as in the periodic case, the only difference lies within the $\vec{k}$-values.



(a) Super-cell Points **(b)** K-Points **(c)** K-Points with boundary condition

Figure 1.12.: 16Av2 lattice: $\vec{l_1} = (4,0)^T$, $\vec{l_2} = (2,4)^T$ with a boundary phase difference of $\varphi = \frac{2\pi}{3}$. (a) the lattice points within the cell remain the same, (b) the k-points within the first BZ of the lattice for periodic boundary conditions and (c) the k-points that lie within the first BZ of the lattice for the boundary condition $\varphi = \frac{2\pi}{3}$ in each direction

	Fig.	$\vec{l}_1^T$	$\vec{l}_2^T$	σ	I_F^{top}	I_F^{geom}	I_B^{top}	I_B^{geom}	Bip.	Symm.
16B	1.9a	(4 0)	(0 4)	1	3	3	1	1	yes	D_4
16A v1	1.9b	(2 4)	(0 8)	1.14	1	1	0	0	yes	C_2
16A v2	1.9c	(2 4)	(4 0)	1.05	1	1	0	0	yes	C_2
2A		(1 1)	(−1 1)	1.00	0	0	0	0	yes	
4A		(2 0)	(0 2)	1.00	1	1	0	0	yes	
8A		(2 2)	(−2 2)	1.00	0	1	0	0	yes	
8C		(2 −1)	(0 4)	0.96	0	1			no	
9A		(3 0)	(0 3)	1.00	0	0			no	
9B		(1 3)	(−3 0)	1.03	0	0			no	
10A		(2 3)	(−2 2)	0.99	0	1			no	
10B		(1 3)	(−3 1)	1.00	1	1	0	0	yes	
11A		(1 3)	(−3 2)	1.01	0	0			no	
12A		(10 −2)	(−4 2)	1.02	2	15	0	4	yes	
12B		(2 3)	(−2 3)	1.04	0	0			no	
12D		(4 0)	(1 3)	1.01	2	2	0	0	yes	
13A		(3 2)	(−2 3)	1.00	0	0			no	
14A		(3 2)	(−3 2)	1.04	0	0			no	
14B		(4 2)	(−1 3)	0.98	2	2	0	0	yes	
15A		(3 2)	(−3 3)	1.00	0	0			no	
15B		(4 1)	(1 4)	1.06	0	0			no	
16A		(4 2)	(0 4)	1.05	1	1	0	0	yes	
16B		(4 0)	(0 4)	1.00	3	3	1	1	yes	
16C		(3 2)	(−2 4)	0.99	0	0			no	
16D		(4 1)	(0 4)	1.02	1	1			no	
17A		(1 4)	(−4 1)	1.00	0	0			no	
17B		(4 3)	(−3 2)	1.00	0	0			no	
18A		(3 3)	(−3 3)	1.00	0	5	0	1	yes	
18B		(3 3)	(−2 4)	1.03	0	0	0	0	yes	
18C		(4 2)	(−1 4)	1.01	0	0			no	
18D		(−1 3)	(−6 0)	0.92	4	6	2	3	yes	
19A		(4 1)	(−3 4)	1.03	0	0			no	
19B		(3 2)	(−5 3)	0.98	0	2			no	
20A		(4 2)	(−2 4)	1.00	1	9	0	3	yes	
20B		(3 4)	(−2 4)	1.05	0	0			no	
20C		(1 4)	(−5 0)	1.01	1	1			no	
20E		(5 1)	(0 4)	0.99	3	3	1	1	yes	
20F		(5 0)	(0 4)	0.99	3	3			no	

Table 1.3.: Properties of super-cells up to 20 sites.

	Fig.	$\vec{l}_1^T$	$\vec{l}_2^T$	σ	I_F^{top}	I_F^{geom}	I_B^{top}	I_B^{geom}	Bip.	Symm.
21A		(3 4)	(−3 3)	1.00	0	4			no	
21B		(1 5)	(−4 1)	0.99	2	2			no	
21C		(3 3)	(−2 5)	1.03	0	2			no	
22A		(2 4)	(−5 1)	1.01	2	1	0	0	yes	
22B		(4 2)	(−3 4)	1.00	0	3			no	
22D		(1 4)	(−5 2)	0.99	1	7			no	
23A		(3 4)	(−5 1)	1.05	0	0			no	
24A		(4 4)	(−4 2)	1.01	3	6	0	3	yes	
24B		(4 0)	(2 6)	0.97	5	10	1	5	yes	
24C		(4 0)	(−4 6)	0.98	7	12	2	6	yes	
24D		(5 1)	(1 5)	1.04	3	0	0	0	yes	
24E		(3 3)	(−2 6)	1.01	3	6	0	3	yes	
24F		(−3 3)	(−7 −1)	1.02	3	8	0	4	yes	
24G		(4 3)	(−4 3)	1.02	0	3			no	
25A		(4 3)	(−3 4)	1.00	0	4			no	
25B		(5 2)	(0 5)	1.04	2	0			no	
26A		(4 2)	(−3 5)	0.98	3	6	0	3	yes	
26B		(1 5)	(−5 1)	1.00	5	5	1	1	yes	
26C		(5 2)	(−3 4)	1.02	0	3			no	
27A		(5 3)	(−4 3)	1.03	0	2			no	
27B		(5 2)	(−1 5)	1.01	2	2			no	
28A		(2 4)	(−8 −2)	1.00	4	9	1	4	yes	
28B		(5 3)	(−1 5)	1.03	2	1	0	0	yes	
28C		(2 5)	(−4 4)	1.04	0	3			no	
28D		(4 3)	(−4 4)	1.00	0	5			no	
29A		(5 2)	(−2 5)	1.00	0	4			no	
29B		(5 3)	(−3 4)	1.00	0	4			no	
30A		(5 3)	(−5 3)	1.06	1	1	0	0	yes	
30B		(4 3)	(−2 6)	1.01	0	4			no	
30C		(2 5)	(−6 0)	1.03	1	3			no	
30D		(1 5)	(−6 0)	1.00	5	5	2	2	yes	
30G		(5 5)	(−2 4)	0.97	5	11	2	5	yes	
31A		(5 3)	(−2 5)	1.00	0	2			no	
31B		(4 5)	(−3 4)	1.00	0	4			no	
32A		(4 4)	(−4 4)	1.00	0	11	0	3	yes	
32B		(2 6)	(−5 2)	0.99	2	4			no	
32C		(3 5)	(−4 4)	1.02	0	2	0	1	yes	
32D		(1 6)	(−5 2)	1.01	1	3			no	
32G		(3 4)	(−5 4)	0.99	0	11			no	
36A		(6 0)	(0 6)	1.00	8	33	3	13	yes	
64A		(8 0)	(0 8)	1.00	18	69	7	27	yes	

Table 1.4.: Properties of super-cells with more then 20 sites.

CHAPTER 2

Lua

2.1. Why Lua

If our code runs computationally intensive operations, it is best to write them in a programming language that is compiled and optimized, however if the code should be made to be flexible and handle more paths of execution and if it also should respond dynamically to the requests of the user, then an interpreted language might be a better match. If you have both requirements, then it might be best to interface an interpreted language with a compiled one. In this project we do it this way: the Lanczos and the Lapack solver are written in C and the simple computations and the main of the program is written in Lua. Python is a very powerful interpreted language whose interpreter has a size of 824kb. Lua is a very nice and easy to use interpreted language. It possesses all the advantages of a programming language and the advantage of being interpreted at runtime. The major

advantage of using Lua would be because of the small size of its interpreter. If we compile the code without the files for the interface and without linking to Lua we get an executable file of the size 68664 bytes, while compiling it including Lua gives a total of 350112 bytes, leading to 281,448 Kbytes for the entire interface files and the interpreter. This is approximately 4 times smaller than the Pyhton interpreter, making it more suitable for usage in scientific computing, where memory is one of the biggest issues.

In this chapter we will present the basic features of the Lua programming language. We start from the usage as a simple input generator and move on to using it in combination with complex constructs like classes to generate a more complete set of input files. In the end we will integrate a dynamic interpreter in a C program allowing us to change the Lua part of the program at run time.

2.2. Lua as simple input

Lua can be used for generating a simple input script. It is very easy to declare variables and move the values to C for computation. As an example we consider a program to solve a Hamiltonian. We will start by declaring simple variables, the vectors defining the lattice on which the electrons sit, as seen in Listing 2.1. This resembles very much a script declaring the variables one after the other, however the order of the variables makes no difference to Lua. We will also show some basic features of the Lua programming language, for example, we can observe that the comments are preceded by minus signs `--`. Also notice the fact that we can group two lines of Lua code using a semicolon, generating a more compact input.

```
1 -- simple lua script defining variables
  ax = 1
3 ay = 0
  bx = 0;by = 1 -- semicolon separates two commands on the same line
```

Listing 2.1: inputVariables

After defining the values in Lua we need to transfer them to C. For this Lua uses a very simple and elegant method: a stack, [**?**]. This stack holds Lua *elements* of any kind: numbers, strings, even functions. Every element occupies one *slot*, regardless of its type. We can push, pop or manipulate elements on the stack pretty easily. For example, to read a number at a given position we call the C function `lua_tonumber(L, pos)`. The position descriptor, `pos`, can either be a positive number, the absolute position of the element starting at 1, or a negative number, the relative position from the top of the stack starting at -1.

In order to gain access to the elements defined in the Lua script, we have to populate the stack with the variables that we need. We begin by initializing Lua with `lua_open` and loading the Lua base libraries, for example the math library or the string library using `luaL_openlibs`. After that we read in the script. We can do this using `luaL_dofile`, which precompiles the script and puts the returned value, the executable code, on the

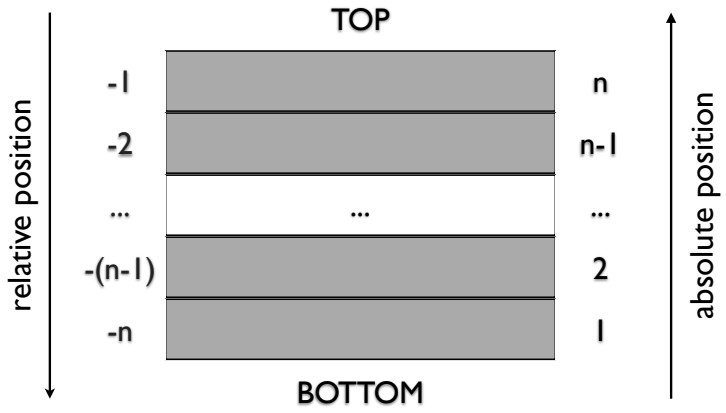


Figure 2.1.: Relative and absolute indexes used with the Lua Stack.

stack and then calls it up. After that we call the variables and place them on the top of the stack by name using `lua_getglobal(L,name)`. This puts the variable on top of the stack and then we can easily convert them to a C variable. The small C program reading in our input can be found in Listing 2.2.

```
// CIncludes
#include "stdio.h"

// LuaIncludes
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"

int main ( int argc, char *argv[] )
{
  // LuaInitializationPhase
  lua_State* L;                 // the Lua interpreter
  L = lua_open();               // initialization phase
  luaL_openlibs(L);             // load Lua base libraries

  // ExecuteLuaScript
  int ret = luaL_dofile(L, "inputVariables.lua");
  if (ret != 0) return ret;

  double a[2], b[2];

  // ReadInParameters - read "ax","ay"
  lua_getglobal(L,"ax");        // put variable on top of stack
  a[0] = lua_tonumber(L,-1);    // convert it to a number
  lua_pop(L,1);                 // remove elem. from stack
  lua_getglobal(L,"ay");
  a[1] = lua_tonumber(L,-1);
```

```
28    lua_pop(L,1);

30    // ReadInParameters - read "bx","by"
      lua_getglobal(L,"bx");
32    b[0] = lua_tonumber(L,-1);
      lua_pop(L,1);
34    lua_getglobal(L,"by");
      b[1] = lua_tonumber(L,-1);
36    lua_pop(L,1);

38    printf("ax,ay = %.2e,%.2e \t bx,by = %.2e,%.2e \n",a[0],a[1],b[0],b[1]);

40    // DoWork

42    // LuaFinalizationPhase
      lua_close(L);
44    return 0;
  }
```

Listing 2.2: mainVariables

At the beginning the stack can hold up to 20 elements. Every time we read a new Lua script, the stack is guaranteed to have at least 20 free slots. Beyond those 20 elements, expanding the stack is left to the user. We can do this using the C call `lua_checkstack(L, newSize)`. If the stack can hold more than `newSize` elements already, the function does nothing and returns, otherwise it expands the Lua stack to the new desired size.

Even if this small example could be useful, in scientific computing we often use vectors or matrices. We show how we can read a one dimensional (1D) array from Lua into C. We start with the same small example of reading vectors that define a lattice. In Lua arrays are implemented as tables. A table can hold any type of element, in our case we use it to hold integers. We show in Listing 2.3 two ways of declaring a table in Lua. The first one declares the table first and then fills it up with variables. This way we can generate also sparse tables by filling only the indices that contain values different from 0, the rest would then be filled with `nil`. The second declaration explicitly fills the table at its declaration point. This way the table will be dense. In contrast to C, indexing starts with 1 and goes all the way to `#table`, the length of the table.

```
1  -- simple lua script defining small tables
   -- define table a
3  a = {}
   -- fill table a
5  a[1] = 1
   a[2] = 0
7
   -- define and fill table b
9  b = {0,1} -- now b[1] = 0 and b[2] = 1
```

Listing 2.3: inputTable

Transferring the variables out of a table into C does not differ much from the reading of atomic variables. We need the index where we would find the table in the stack. We can

move the table for example to the top of the stack, using again `lua_getglobal(L,name)`. We also need the index in the table that we want to access. We push the index on top of the stack and then call `lua_gettable(L,pos)`, where `pos` is the position of the table. The index is taken from the top of the stack and the element from the table at the required index will be placed on the top instead.

```
//Includes
[...]

int main ( int argc, char *argv[] )
{
  // LuaInitializationPhase
  [...]

  // ExecuteLuaScript
  [...]

  double a[2], b[2];

  // ReadInParameters - read table "a"
  lua_getglobal(L,"a");           // put table on top on stack
  for(int i = 0; i < 2; i++) {
    lua_pushnumber(L,i+1);      // push wanted Lua index of stack
    lua_gettable(L,-2);         // push a[i+1] from table at pos -2
    a[i] = lua_tonumber(L,-1); // convert it to a number
    lua_pop(L,1);               // remove elem. from stack
  }
  lua_pop(L,1);                   // pop table

  //ReadInParameters - read table "b"
  [...]

  printf("ax,ay = %.2e, %.2e \t bx,by = %.2e, %.2e \n",a[0],a[1], b[0], b[1])
      ;

  // DoWork

  // LuaFinalizationPhase
  lua_close(L);
  return 0;
}
```

Listing 2.4: mainTable

Tables in Lua do not simply work as arrays. We can also store "key-value" pairs in them as we would in a dictionary, which we can then access by `myTable.myKey = myValue`.

```
> myTable = { 11, 22, x=1, y=2 }
> -- myTable[1] = 11; myTable[2] = 22; myTable.x = 1; myTable.y = 2
```

If we then want to gain access to the elements of our dictionary out of C, we push, instead of the array index, the key on the stack using `lua_pushstring(L,"x")`.

In this section we showed that access to variables defined in Lua is made easy through the implementation of a communication stack. The input file is not restricted to a specific format, the variables can be defined in whatever order one likes. The only restriction is that it has to respect the Lua language style and the variable names have to be known. We have shown how we can read in atomic variables and tables, but Lua is more than an input language, it is a programming language. So we could think about inserting more Lua code, e.g. compute the Hamiltonian we want to solve in Lua and pass the entire matrix to C for solving it.

2.3. Input table

Lua has a flexible format and can work with derived data types, making it a good candidate for an input language. After transferring the Lua input into C via the stack we can use the computing power of C to solve complex problems. But Lua is also a programming language. We can use Lua to generate a more complex input, keeping the compiled code as general as possible. To show this we build up the Hamiltonian for the Hubbard model in Lua and keep only a matrix solver in C. We require the user to give only simple parameters: the number of sites and the hopping element. Using these simple parameters we construct the single particle Hamiltonian in Lua and pass that to the C program for solving.

In Listing 2.5 we have the input generation for a 1D periodic chain. We first have to declare the matrix as a table, in which we will store a table for each line. We then fill it with the nearest neighbour hopping matrix elements.

```
-- more complicated input
-- input needed from user
nsites = 4
t = 1

-- construct hamiltonian for 1Dchain, pbc
hsgl = {}
for i = 1,nsites do
  -- initialize line
  hsgl[i] = {0,0,0,0}
  -- fill line with nearest neighbour hopping
  j = ((i-1)-1+nsites)%nsites+1
  hsgl[i][j] = -t
  j = ((i-1)+1+nsites)%nsites+1
  hsgl[i][j] = -t
end
```

Listing 2.5: ex2_inputTable

In order to read `hsgl` into C we define ourselves some re-usable functions that manipulate the stack. We need at least the following:

- `array_luaGetElementAt_plain` and `matrix_luaGetElementAt_plain` - get the specified element of a 1D/2D table and put it on the top of the stack. These functions work with any type of elements of the table, since they actually do not read the

elements, just place them on top of the stack. Note that we do not make any checks regarding the existence of the elements or of the table at the given position.

- `table_lua2c` - function returning a dense 2D table of numbers defined in Lua found at a given position in the stack to C.

We can improve on the code by inserting safety checks. We can for example check if at a specified position we actually have a table. This is done by checking the return value of the function `lua_type(L, pos)`. If it is equal to a predefined value `LUA_TTABLE` then we have a table at that position in the stack, however these tests would cost time, so we have to be considerate when using them.

```
//Includes
[...]

// Get from array at "pos" array["index"] and put it on stack
// index C style, starting at 0
void array_luaGetElementAt_plain(lua_State* L, int pos, int index){
  lua_rawgeti (L, pos, index+1); // converting to index Lua style, starting
      at 1
  return;
}

// Get from table at "pos" table["l"]["c"] and put it on stack
void table_luaGetElementAt_plain(lua_State* L, int pos, int l, int c){
  array_luaGetElementAt_plain(L, pos, l);     // put line l on stack
  array_luaGetElementAt_plain(L, -1, c);      // put column c on stack
  lua_remove(L,-2);                           // remove line l from stack
  return;
}

// Get dense 2D table, *nLines = #entries, *nCols = size of entry
// Warning not to use when table can contain nils
//   -- it will return wrong sizes and then a wrong table
double* table_lua2c(lua_State* L,int pos, int* nLines, int* nCols){
  // returning the number of elements in the table
  *nLines = lua_objlen(L, pos);

  array_luaGetElementAt_plain(L, pos, 0);
  // counting the columns with objlen
  *nCols = lua_objlen(L, -1);
  lua_pop(L,1);

  // allocate memory
  double *res;
  res = (double *) malloc(sizeof(double) * (*nLines) * (*nCols));
  int i, j;
  for(i = 0; i < *nLines; i ++){
    array_luaGetElementAt_plain(L,pos,i);
    for(j = 0; j < *nCols; j ++){
      array_luaGetElementAt_plain(L, pos, j);
      res[i*(*nCols)+j] =  lua_tonumber(L,-1);
      lua_pop(L,1);
    }
    lua_pop(L,1);
  }
```

```
44    return res;
  }
46
  int main ( int argc, char *argv[] )
48 {
    // LuaInitializationPhase
50  [...]

52  // ExecuteLuaScript
    [...]
54
    double *hsgl;
56  int lines, columns, i, j;

58  // ReadInParameters - get "hsgl"
    lua_getglobal(L,"hsgl");   // put table on stack
60  hsgl = table_lua2c(L,-1,&lines,&columns);

62  for(i = 0; i < lines; i ++){
      for(j = 0; j < columns; j++)
64      printf("%+.2e ",hsgl[i*columns+j]);
      printf("\n");
66  }

68  // DoWork

70  // LuaFinalizationPhase
    lua_close(L);
72  return 0;
  }
```

Listing 2.6: ex2_mainTable

We have moved some computations to Lua leaving less work to be done in C. The input file now reflects more closely what we would like to solve and there is no intermediate step in the C program that generates the actual matrix, however we would have to pay the price of a bigger input file. But Lua could do much more than compute the Hamiltonian for the simple 1D chain. It could be modified to calculate the Hamiltonian for different, more complicated systems as well and we can use the same compiled code to solve them. The script then contains the explicit description of the system and we do not need to recompile to run a new system.

2.4. Classes in Lua

We will show in this section how classes are implemented in Lua and how we can use them to our advantage. Let's start off by looking more closely at the tables in Lua. We start by having a look at the following code, (REF http://lua-users.org/wiki/LuaClassesWithMetatable):

```
1 > t = { 11, 22, 33, you='one', me='two' }
  > -- iterate over table t and apply function print on elements
3 > table.foreach(t,print)
```

```
1       11
2       22
3       33
me      two
you     one
>
> = t[2]
22
> = t.me
two
> = t.fred
nil
```

This example shows every feature of the table in Lua that we discussed earlier. The table can be indexed by numbers and by keys. We used here the `table.foreach` construct to print out every entry in our table. Notice that when we ask for a key that is not found in our table, we get a `nil`. We now explain what actually happens in the background.

Lua follows the prototype-based object oriented programming paradigm. It uses for each object 3 tables:

- objectTable - the actual object that can contain several elements
- functionTable - the functions and methods that our object should have. It is a table containing as keys the function names and as values the code that needs to be executed
- metaTable - the table that contains information about specific functions, e.g. `print`, `+`, etc. Here also information about the functionTable is kept.

When we call a key that is not found in the objectTable, the `__index` function is called from the metaTable. The metaTable is nothing more than a normal table that has as keys some specific function names and as values the code for those functions. For example the metaTable contains the `__index` function mentioned earlier or the `__gc` function that is called when the garbage collection cycle wants to release our variable [?]. If the `__index` value is again a table, the functionTable, this table is iterated instead.

To start defining a new metaTable for a variable we just have to define a table and store the functions needed in it. Let's start by changing the `__index` entry in the metaTable to the print function. After defining our new metaTable with the desired functionality, we also have to specify that the associated variable has this metaTable. This is done using the `setmetatable` function.

```
> t = { 11, 22, 33, you='one', me='two' }
> -- define a new metatable with a new __index function
> mt = {}
> mt.__index = print
> -- set the metatable to our variable
> setmetatable(t,mt)
> -- test the metatable entry
> = t.fred
table: 0x8075e80        fred
```

```
nil
```

To start writing a class in Lua we first define our prototype, the functionTable. This contains every method that will be used by objects of this class. We also build a constructor for our class, found in the functionTable. This constructor initializes the values of the objectTable and sets the metaTable entry `__index` to our functionTable. Then it returns the objectTable to the caller. This is exactly how we start in Listing 2.7. After that, we can add functions and methods to our functionTable. Functions and methods are basically the same. The only difference between a function and a method is that the method expects to be called from an instantiated object and has one implicit argument, the object itself, stored in the variable `self`.

We can now take a look at the definition of a rectangular lattice. We define functions that count the number of points inside the lattice cell, that define a list of all points and that can build a Hamiltonian given the on-site energy and the hopping element for periodic and antiperiodic boundary conditions.

```
nonSkewedLattice = {}
nonSkewedLattice.__index = nonSkewedLattice

-- conctructor for rectangular Lattice of size a1 x a2
function nonSkewedLattice.create(a1,a2)
  local sl = {}
  setmetatable(sl, nonSkewedLattice)
  sl.a1 = a1
  sl.a2 = a2
  sl.nsites = sl.a1*sl.a2
  sl.points = sl:buildPoints()
  return sl
end

-- stores all points that are considered in the super cell spanned by a1 and
    a2
function nonSkewedLattice:buildPoints()
  local latticePoints = {}
  for i = 0,self.a1-1 do
    for j = 0,self.a2-1 do
      table.insert(latticePoints,{i,j})
    end
  end
  return latticePoints
end

-- map point p to equivalent point in cell
function nonSkewedLattice:mapInCell(p)
  local coeff = {- math.floor(p[1]/self.a1),- math.floor(p[2]/self.a2)}
  local newp = { p[1] + coeff[1]*self.a1, p[2] + coeff[2]*self.a2}
  return newp,coeff
end

-- returns the index of a given point in the latticePoints table
function nonSkewedLattice:getIndex(p)
```

```
  for i = 1,self.nsites do
    if ((self.points[i][1] == p[1]) and (self.points[i][2] == p[2])) then
      return i
    end
  end
  return -1
end

-- build Hamiltonian given boundary conditions bc, hopping matrix element t
    and local energy eps0
-- NOTE - bc = [bc1, bc2] with 0 - periodic or 1 - antiperiodic, boundary
    condition vector
function nonSkewedLattice:buildH(eps0, t, bc)
  local H = {}
  -- initialize Hamiltonian with 0
  for i = 1,self.nsites do
    H[i] ={}
    for j = 1, self.nsites do
      H[i][j] = 0
    end
  end

  -- insert values for each lattice point
  for i = 1, self.nsites do
    H[i][i] = eps0  -- insert local energy
    local current_site = self.points[i]
    -- for each nearest neighbour
    -- first go up on y
    local next_site = {current_site[1], current_site[2]+1}
    local new_p, coeff = self:mapInCell(next_site)
    H[i][self:getIndex(new_p)] = H[i][self:getIndex(new_p)]-(-1)^(bc[1]*coeff
        [1])*(-1)^(bc[2]*coeff[2])*t
    -- go down on y
    next_site = {current_site[1], current_site[2]-1}
    new_p, coeff = self:mapInCell(next_site)
    H[i][self:getIndex(new_p)] = H[i][self:getIndex(new_p)]-(-1)^(bc[1]*coeff
        [1])*(-1)^(bc[2]*coeff[2])*t
    -- first go up on x
    next_site = {current_site[1]+1, current_site[2]}
    local new_p, coeff = self:mapInCell(next_site)
    H[i][self:getIndex(new_p)] = H[i][self:getIndex(new_p)]-(-1)^(bc[1]*coeff
        [1])*(-1)^(bc[2]*coeff[2])*t
    -- go down on x
    next_site = {current_site[1]-1, current_site[2]}
    new_p, coeff = self:mapInCell(next_site)
    H[i][self:getIndex(new_p)] = H[i][self:getIndex(new_p)]-(-1)^(bc[1]*coeff
        [1])*(-1)^(bc[2]*coeff[2])*t
  end
  return H
end

return nonSkewedLattice -- output for require function
```

Listing 2.7: ex3_nonSkewedLattice

Let's assume we also store the class in a separate file and we want to include this file in our script. To achieve this we use the function `require "filename"`, which actually comprises the actions of the file and stores the result of the execution in a variable. We then can access the functions or variables stored in the file. That is why our class definition ends with a return statement.

```
local nonSkewedLattice = require "nonSkewedLattice"

function initHsgl()
  myNSkewedLattice = nonSkewedLattice.create(6,1)
  -- build single particle Hamiltonian
  -- bc = {0,0} -- periodic BC
  bc = {1,1} -- antiperiodic BC
  t = 1      -- hopping element
  eps0 = 0   -- on site energy
  hsgl = myNSkewedLattice:buildH(eps0, t, bc)
end
```

Listing 2.8: ex3_script

Now that we have grouped the input regarding the generation of the Hamiltonian for rectangular lattices into a file, we can leave the C code even more general, solving for a matrix in the most general case, not restricting ourselves to a given setting. We will call in our small example the Lapack routine to solve a symmetric matrix. We will expand also our general Lua interface functions by adding a call from C to a function with a given name in Lua, `eval(L,funcName)`. This is useful if we want the Lua part to get the control of our program.

```
//Includes
[...]

// Lapackroutine called
extern void dsyev( char* jobz, char* uplo, int* n, double* a, int* lda,
    double* w, double* work, int* lwork, int* info );

void array_luaGetElementAt_plain(lua_State* L, int pos, int index){[...]}

void table_luaGetElementAt_plain(lua_State* L, int pos, int l, int c){[...]}

double* table_lua2c(lua_State* L,int pos, int* nLines, int* nCols){[...]}

// function calling a lua function
void eval(lua_State* L, char* func){
  assert (luaL_loadstring( L, func ) || lua_pcall( L, 0, LUA_MULTRET, 0 ) ==
      0);
}

// wrapper for Lapack call
void lapack_call_r(int n, double * a, double ** ee_p, double ** ev_p){
  // Locals
  int info, lwork;
  double wkopt;
  double* work;
  // Local arrays
```

```
  double w[n];
  // Query and allocate the optimal workspace
  lwork = -1;

  char cmd[] = "Vectors";
  char opt[] = "Upper";
  dsyev( cmd, opt, &n, a, &n, w, &wkopt, &lwork, &info );
  lwork = (int)wkopt;
  work = (double*)malloc( lwork*sizeof(double) );
  // Solve eigenproblem
  dsyev( cmd, opt, &n, a, &n, w, work, &lwork, &info );

  // Construct result vectors
  double * ee;
  double * ev;
  ee = (double * ) malloc (sizeof(double) * n);
  ev = (double * ) malloc (sizeof(double) * n * n);

  int i, j;
  for( i = 0; i < n; i++ ) {
    ee[i] = w[i];
    for( j = 0; j < n; j++ ) {
      ev[i*n+j] = a[i*n+j];
    }
  }

  *ee_p = ee;
  *ev_p = ev;

  // Free workspace
  free( (void*)work );
  return;
}

int main ( int argc, char *argv[] ) {
  // LuaInitializationPhase
  [...]

  // load the script - equivalent to luaL_dofile(L,"filename";
  if(luaL_loadfile(L, "script.lua")||lua_pcall(L, 0, LUA_MULTRET, 0)){
    printf("Error: %s \n",lua_tostring(L, -1));
    lua_pop(L,1);
    exit(1);
  }

  // call function to initialize Hamiltonian
  char str[] = "initHsgl()";
  eval(L,str);

  double *hsgl;
  int lines, columns;

  // ReadInParameters - get "hsgl"
  [...]

  // SolveWithLapack
```

```
    double *ee;
82  double *ev;

84  lapack_call_r(lines, hsgl, &ee, &ev);

86  // DoOtherWork

88  // LuaFinalizationPhase
    lua_close(L);
90  return 0;
}
```

Listing 2.9: ex3_mainLapack

After this section we can use Lua to generate a more general input file, moving some of the easy computation in the Lua part of our project. We can also group the different properties belonging together into a class, implemented in Lua using tables. We have shown also how one can call a given Lua function from the C program, knowing its name and the arguments. In this way we can design a project that has the main control in C and calls Lua functions for the easy input generation. We would then give the control from time to time to Lua, structuring our project as a ping-pong game between Lua and C. If we can design access from C to Lua, why not the other way around as well? We could then leave the entire control to Lua and call up different C functions for the number crunching.

2.5. Lua calling C

In the end we would like to have a project that runs entirely from Lua and that executes the computationally intensive parts in C. To achieve this we have to let Lua know about the functions that we defined in the C program. We will start this section by explaining the registration of functions to Lua and then show the way we could use it to call the solving routine seen in the previous example.

```
1 void lapack_call_r(int n, double * a, double ** ee_p, double ** ev_p);
```

We would like to call this function from Lua for example like

```
1 lee, lev = solve(hsgl)
```

The only thing that we will have to implement in the C program is the interface function that makes the binding between Lua and C. This function has the following signature:

```
1 static int solveC(lua_State *L);
```

Every call from Lua will translate into this kind of a C call. The arguments that are transferred from Lua are put on the stack `L`, in the order that they are used in the Lua part. We would then have at the first index the first argument, at the second index the second argument and so on. We could also index the stack with a relative (negative) index. We can check the number of arguments on the stack by calling the function `lua_gettop(L)` before starting the computations. For every call from Lua a different stack is created and given to C that can hold up to 20 elements. Here too the stack management is left up to the user. The return values are put on the same stack and the single return argument of the function is the number of arguments returned to Lua. We can pop the original calling arguments and leave the stack with only the return values on it, or we can leave all arguments on the stack. The stack will be freed after returning to Lua and getting the arguments back. In the Listing 2.10 we see the implementation of our little example calling the Lapack routine from Lua using the `solveC` interface function. We also expand our general Lua-C interface functions with functions that write values back to Lua, `array_c2lua` and `array_c2lua_plain`.

```
1  //Includes
   [...]
3
   // Lapackroutine called
5  [...]

7  void array_luaGetElementAt_plain(lua_State* L, int pos, int index){[...]}

9  void table_luaGetElementAt_plain(lua_State* L, int pos, int l, int c){[...]}

11 double* table_lua2c(lua_State* L,int pos, int* l, int* c){[...]}

13 // Writing a 1D table to lua and keep it on top of the stack
   int array_c2lua_plain(lua_State* L, double* vec, int size){
15   lua_createtable(L, size, 0);
     int newTable = lua_gettop(L);
17   int i;
     for( i = 0; i < size; i ++){
19     lua_pushnumber(L,vec[i]);        //push element on top of stack
       lua_rawseti (L, newTable, i+1); //push the element in line i of newTable
21   }
     return newTable;
23 }

25 // Writing a potentially sparse 1D table to lua and keep it on top of the
       stack
   int array_c2lua(lua_State* L, double* vec, int size){
27   int newTable = array_c2lua_plain(L, vec, size);
     lua_pushstring(L,"size");
29   lua_pushnumber(L,size);
     lua_rawset(L,newTable);
31   return newTable;
   }
33

35 void lapack_call_r(int n, double * a, double ** ee_p, double ** ev_p){[...]}
```

```
37 // interface function between Lua and C
   static int solveC(lua_State *L)
39 {
     int lines, columns;
41   double * matrix = table_lua2c(L, -1, &lines, &columns);

43   double *ee;
     double *ev;
45
     lapack_call_r(lines, matrix, &ee, &ev);
47
     array_c2lua(L, ee, lines);
49   array_c2lua(L, ev, lines*lines);

51   return 2;
   }
53
   int main() {
55   // LuaInitializationPhase
     [...]
57
     // RegisterFunctionsToLua
59   lua_register(L, "solve", solveC);

61   // ExecuteLuaScript
     [...]
63
     // LuaFinalizationPhase
65   lua_close(L);
     return 0;
67 }
```

Listing 2.10: ex4_mainLapack

On the Lua part, the changes are not that big, we only have to use our newly registered function `solve` with the proper arguments and get the result, as if it would be a simple Lua function.

```
1 local nonSkewedLattice = require "nonSkewedLattice"

3 function initHsgl()
    myNSkewedLattice = nonSkewedLattice.create(6,1)
5   -- build single particle Hamiltonian
    bc = {0,0} -- periodic BC
7   bc = {1,1} -- antiperiodic BC
    t = 1 -- hopping element
9   eps0 = 0 -- on site energy
    hsgl = myNSkewedLattice:buildH(eps0, t, bc)
11 end

13 initHsgl()
   -- call C solve function
15 l_ee, lev = solve(hsgl)

17 print("LUA: Eigenvalues")
   for i=1,#hsgl do
```

```
19    io.write (string.format("%.2f\t",l_ee[i]))
  end
21  print()
  print("LUA: Eigenvectors - linewise")
23  l_ev = {}
  for i=1,#hsgl do
25    l_ev[i] = {}
    for j=1,#hsgl do
27      io.write (string.format("%.2f\t", lev[(i-1)*#hsgl+j]))
      l_ev[i][j] = lev[(i-1)*#hsgl+j]
29    end
    print()
31  end
  print()
```

Listing 2.11: ex4_script

After this section we can design a program that can be steered by the Lua script. We can execute simple functions in Lua, we can register almost any C function to be called from Lua and get the result back into our script. Moreover we can, if necessary call Lua functions from C and compute things again. This functionality suffices for almost any scientific program, however we want to introduce some nicer features of the C++ programming language, namely classes in order to group the variables in C as well.

2.6. Lua and advanced C++ programming

It is much more convenient to group the variables and functions that work on typical structures together in a class, as we did in Lua. For our Lapack example we want to build a class holding our structure: the matrix, the eigenvalues and the eigenvectors. As functions we want to be able to set the matrix and call the Lapack solver. The header file will then look like in Listing 2.12. The code for the implementation of the header can be found in Listing 2.13 and is a purely C++ implementation with no relation to Lua.

```
  #ifndef LAPACK_SOLVER_HPP
2 #define LAPACK_SOLVER_HPP

4 class LapackSolver{
    private:
6
    public:
8     double *a;          // the matrix that we want to diagonalize
      int n;              // size of matrix a
10
      // the solutions given by a lapack solver
12    double *ee;         // eigenvalues
      double **ev;        // eigenvectors
14
      LapackSolver();                              // empty constructor needed for
          Lua
16    LapackSolver(double * a_new, int n_new);  // create new LapackSolver
```

```
18      void solve();
        void print_matrix(const char* desc, int m, int n, double* a, int lda );
20
        void freeSolver(); // clean up
22  };
    #endif
```

Listing 2.12: ex5_LapackSolverHpp

```
    #include <stdio.h>
2   #include <iostream>

4   extern "C"{
      void dsyev( char* jobz, char* uplo, int* n, double* a,
6           int* lda, double* w, double* work, int* lwork, int* info );
    }
8
    #include "LapackSolver.hpp"
10
    LapackSolver::LapackSolver(){
12    ee = NULL;
      ev = NULL;
14     a = NULL;
       n = 0;
16  }

18  LapackSolver::LapackSolver(double * a_new, int n_new){
      ee = NULL;
20    ev = NULL;
      n = n_new;
22    a = (double*) malloc( n*n*sizeof(double) );
      for( int i = 0; i < n*n; i++)
24      a[i] = a_new[i];
    }
26
    void lapack_call_r(int n, double * a, double ** ee_p, double ** ev_p){[...]}
28
    // call Lapack solver
30  void LapackSolver::solve(){
      if(a != NULL) lapack_call_r(n, a, &ee, &ev);
32    return;
    }
34

36  void LapackSolver::freeSolver(){
      if(ee) free(ee);
38    if(ev)
        for(int i = 0; i < n; i++)
40        if(ev[i]) free(ev[i]);
      if(a) free(a);
42    n = 0;
    }
```

Listing 2.13: ex5_LapackSolver

We want to use this C++ code in Lua under a special namespace, that groups our class, for example `LapackSolver`.

```
solver = LapackSolver.new(hsgl)
lee, lev = solver:solve()
```

To do this we have to define interface functions for every method that we want to access, define the namespace and register it to Lua. Therefore we implement in C some very simple functions:

- `lua pushSolver` - a function that creates a new instance of the solver. It calls the constructor of the LapackSolver class and sets also the matrix. The function returns a `userdata` to Lua, which can be seen as a pointer to the C++ class
- `lua_toSolver`,`lua_checkSolver` - it reads a solver from the stack and returns it as a pointer or as an instance to C++
- `lua_solveLapack` - the interface function that calls the solver routine in the class and returns the eigenvalues and eigenvectors to Lua

Last but not least we have to make sure that the Lua program knows about these functions and their names in Lua. Therefore we have to define the functionTable and add them to the `__index` entry of the metaTable. We can also define some other entries of the metatable, like the `__gc` or the `__print` functions. The last function of our interface file registers this new type to Lua.

```
#include "tb-lua.hpp"
#include "LapackSolver.hpp"

// LuaIncludes
[...]

// Create a Solver instance and return it to lua
static int lua_pushSolver( lua_State *L ){
  int lines, columns;
  double * matrix = table_lua2c(L, -1, &lines, &columns);

  LapackSolver solver(matrix, lines);

  LapackSolver* pSolver = (LapackSolver *)lua_newuserdata(L, sizeof(
      LapackSolver));
  *pSolver = solver;
  luaL_getmetatable( L, "LapackSolver");
  lua_setmetatable( L, -2 );

  return 1;
}

// Convert the item at "index" in the stack to a LapackSolver*
static LapackSolver* lua_toSolver (lua_State *L, int index){
  LapackSolver *pSolver = (LapackSolver *)lua_touserdata(L, index);
  if (pSolver == NULL) luaL_typerror(L, index, "LapackSolver");
  return pSolver;
```

```
}

// Convert the item at "index" in the stack to a LapackSolver
static LapackSolver lua_checkSolver (lua_State *L, int index){
  LapackSolver *pSolver, solver;
  luaL_checktype(L, index, LUA_TUSERDATA);
  pSolver = (LapackSolver*)luaL_checkudata(L, index, "LapackSolver");
  if (pSolver == NULL) luaL_typerror(L, index, "LapackSolver");
  solver = *pSolver;
  return solver;
}

// wrapper that calls the solve routine and returns the ee and ev to Lua
static int lua_solveLapack( lua_State *L ){
  LapackSolver solver = lua_checkSolver(L, -1);
  lua_pop(L,1);
  solver.solve();

  array_c2lua(L, solver.ee, solver.n);
  table_c2lua(L, solver.ev, solver.n);

  return 2;
}

// methods of LapackSolver Lua class
static const luaL_reg LapackSolver_methods[] = {
  {"new",       lua_pushSolver},
  {"solve",     lua_solveLapack},
  {0,0}
};

// function that will be called at garbage collection
static int Lapack_gc (lua_State *L){
  LapackSolver *pSolver = lua_toSolver(L,1);
  if (pSolver) pSolver->freeSolver();
  printf("goodbye Lapacksolver (%p)\n", lua_touserdata(L, 1));
  return 0;
}

// the metamethods that are registered
static const luaL_reg LapackSolver_meta[] = {
  {"__gc",       Lapack_gc},
  {0, 0}
};

// function that registers the methods and metamethods to the new class
int Lua_Lapack_register (lua_State *L){
  // add methods table to globals
  luaL_openlib(L, "LapackSolver", LapackSolver_methods, 0);
  // add Solver metatable to Lua registry
  luaL_newmetatable(L, "LapackSolver");
  luaL_openlib(L, 0, LapackSolver_meta, 0); // fill metatable
  lua_pushliteral(L, "__index");
  lua_pushvalue(L, -3);                     // dup methods table
  lua_rawset(L, -3);                        // metatable.__index = methods
  lua_pushliteral(L, "__metatable");
```

```
  lua_pushvalue(L, -3);                    // dup methods table
84  lua_rawset(L, -3);                     // hide metatable
  lua_pop(L, 1);                           // remove metatable
86  return 1;                              // return methods on the stack
}
```

Listing 2.14: ex5_LuaLapackSolverHpp.tex

At this point in time, grouping all general communication functions into a header file seems like a very good idea. We also expand the interface with a function that generates directly a Lua 2D table. Many other functions could then be added to this interface, however for our introduction we have reached the desired capabilities and this is the final version of the interface file.

```
1 //Includes
  [...]
3
  void array_luaGetElementAt_plain(lua_State* L, int pos, int index){[...]}
5
  void table_luaGetElementAt_plain(lua_State* L, int pos, int l, int c){[...]}
7
  double* table_lua2c(lua_State* L,int pos, int* l, int* c){[...]}
9
  int array_c2lua_plain(lua_State* L, double* vec, int size){[...]}
11
  int array_c2lua(lua_State* L, double* vec, int size){[...]}
13
  // Writing a 2D table to lua and keep it on top of the stack
15 // note this function calls array_c2lua_plain
  // note this function assumes a square table
17 int table_c2lua(lua_State* L, double** vec, int size){
    int lines = size;
19  int columns = size;
    // convert the vec pointer into a lua table
21  lua_createtable(L,lines, 1);
    int newTable = lua_gettop(L);
23  for(int i = 0; i < lines; i ++){
      array_c2lua_plain(L,vec[i],columns);
25    lua_rawseti (L, newTable, i+1);
    }
27  lua_pushstring(L,"size");
    lua_pushnumber(L,size);
29  lua_rawset(L,newTable);
    return newTable;
31 }
```

Listing 2.15: ex5_tb-luaHpp.tex

The only difference in the main program is that it has to call the registration function and give the control to the Lua script, executing a Lua file as before.

```
1 //Includes
  [...]
3
  #include "LapackSolver.hpp"
```

```
#include "LuaLapackSolver.hpp"

int main() {
  // LuaInitializationPhase
  [...]

  // RegisterFunctionsToLua
  Lua_Lapack_register(L);

  // ExecuteLuaScript
  [...]

  // LuaFinalizationPhase
  lua_close(L);
  return 0;
}
```

Listing 2.16: ex5_mainLapack

In the Lua part of the program we only have to use the new class defined. As for normal classes we call functions and methods for the instantiated objects. Note that there is no apparent difference in usage between the classes defined in Lua and the classes defined in C. The usage is the same way, however we have to bear in mind that the LapackSolver stores a pointer to the C structure and not a Lua table internally.

```
local nonSkewedLattice = require "nonSkewedLattice"

function initHsgl()
  myNSkewedLattice = nonSkewedLattice.create(6,1)
  -- build single particle Hamiltonian
  bc = {0,0} -- periodic BC
  bc = {1,1} -- antiperiodic BC
  t = 1      -- hopping element
  eps0 = 0   -- on site energy
  hsgl = myNSkewedLattice:buildH(eps0, t, bc)
end

initHsgl()

solver = LapackSolver.new(hsgl)
l_ee, l_ev = solver:solve()

print("LUA: Eigenvalues")
for i=1,#hsgl do
  io.write (string.format("%.2f\t",l_ee[i]))
end
print()
print("LUA: Eigenvectors - linewise")
for i=1,#hsgl do
  for j=1,#hsgl do
    io.write (string.format("%.2f\t", l_ev[i][j]))
  end
  print()
end
print()
```

Listing 2.17: ex5_script

In this section we have shown that it is very easy to get access to C++ classes as well, not only C functions. We can create new classes from C and then use them in Lua. The last step that we would like to do is to adapt the code dynamically, using an interpreter and not a predefined script, so that we could use several input files depending on our current needs.

2.7. The Lua interpreter and dynamic input

The Lua interpreter is written in C, we could use this to integrate the interpreter into our program and read lines from a prompt and react dynamically to the requests of the user. This way we could import Lua classes according to the current demands. We will show here how this can be done. The Lua part of the program does not change at this point. Now every line of the script will be requested from the keyboard. This is very good while debugging and testing new codes. On the C part, we have to define some functions that create our prompt and read command lines. This is done here in the main function in Listing 2.18. All we have to do then is to import some functions from the Lua implementation, namely the functions found in the header of Listing 2.19. These can be copied from the official lua.c code and help us to set a personalized prompt and parse and interpret lua commands.

```
// CIncludes
[...]

// LuaIncludes
#include "LuaHelper.hpp"
[...]

#include "LapackSolver.hpp"
#include "LuaLapackSolver.hpp"

int leval(lua_State* L, std::string str ){
  return (luaL_loadstring( L, str.c_str() ) || lua_pcall( L, 0, LUA_MULTRET,
      0 ));
}

int main() {
  // LuaInitializationPhase
  [...]

  // RegisterFunctionsToLua
  Lua_Lapack_register(L);

  // StartLuaInterpreter
  leval(L, "_PROMPT='iParLa> '\n");
  prompt(L);
```

```
26    // LuaFinalizationPhase
      lua_close(L);
28    return 0;
}
```

Listing 2.18: ex6_mainLapack

```
1  #ifndef __LUA_HELPER__
   #define __LUA_HELPER__
3
   extern "C"
5  {
   #include "lua.h"
7  #include "lualib.h"
   #include "lauxlib.h"
9  }

11 int incomplete( lua_State *L, int status );
   int pushline( lua_State *L, int firstline );
13 int loadline( lua_State *L );
   void prompt( lua_State *L );
15 void l_message( const char *pname, const char *msg );
   int report( lua_State *L, int status );
17 const char *get_prompt( lua_State *L, int firstline );
   int docall( lua_State *L, int narg, int clear );
19 #endif
```

Listing 2.19: ex6_LuaHelperHpp

At the end of this chapter we have a fully functional, dynamic program that could be extended for any scientific application. We have shown how to generate classes in Lua and how to transfer information found in those classes to C. Furthermore we have shown that we can gain access both to C functions and to C++ classes in Lua and run a dynamical interpreter. This way we can tune our program to do all the heavy computations in C++ and all the computations that might change from problem to problem in Lua. We can then differentiate using the `require` function between the needed classes in Lua and import only those variables that we need for the given simulation, while keeping the possibility open to expand the program for further usage. Due to the light-weight interpreter we can afford to keep the Lua interpreter integrated even in massively parallel calculations, without any major influence on the memory management.

CHAPTER 3

Lanczos Method

The Lanczos algorithm, [**?**, **?**, **?**] is a method based on the idea of steepest descent. For the energy expectation value the direction of steepest descent starting from a vector $|v_0\rangle$ is related to $H|v_0\rangle$. Instead of performing only one step in this direction, we apply the Hamiltonian again and search for a minimum in the new space. The space that we generate this way is the Krylov space

$$K(H, v_0) = \{|v_0\rangle, H|v_0\rangle, H^2|v_0\rangle, ...\}. \tag{3.1}$$

The idea is to construct the solution iteratively, until convergence is reached, or until the space does not change anymore when applying the Hamiltonian again. Since the energy functional that is minimized has no local minimum, the ground state information is computed correctly. A good approximation to the ground state is reached even after only a few iterations, which allows us to truncate the construction of the Krylov space. This then lets us solve also bigger systems. The Hamiltonian in the Lanczos basis is tridiagonal and can be solved without much computational effort.

We will first start by describing the theoretical background for the Lanczos method and then go on to computing the ground state energy and the ground state vector using this method. The last section of this chapter is dedicated to calculating the spectral function using the Lanczos solver.

3.1. Variational Principle

The variational method is a way of calculating the ground state information by minimizing the energy functional

$$E[\psi] = \frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle}. \tag{3.2}$$

If ψ_n is equal to an eigenfunction of the Hamiltonian, then $E[\psi_n] = E_n$ is the corresponding eigenvalue, meaning also that $\frac{\delta E[\psi]}{\delta\langle\psi_n|} = 0$ for every eigenstate. However, these are saddle points, except for the ground state which is the only minimum of the functional. This means that the steepest descent method leads up to the correct ground state.

Varying the energy functional we get the following gradient

$$\frac{\delta E[\psi]}{\delta\langle\psi|} = \frac{H|\psi\rangle - E[\psi]|\psi\rangle}{\langle\psi|\psi\rangle}. \tag{3.3}$$

Modifying ψ in the opposite direction of this functional gradient we get a new wave-function $|\psi'\rangle = |\psi\rangle - \alpha\frac{\delta E[\psi]}{\delta\psi}$ which has a smaller value of the energy functional $E[\psi']$, with α an unknown parameter. We can calculate the parameter α by searching for the minimum of the functional in the space spanned by $\{\psi, \psi'\}$, which in fact is equal to $span(|\psi\rangle, H|\psi\rangle)$. We can construct an orthonormal basis starting from our initial vector $|\psi\rangle$ as follows:

$$|v_0\rangle = \frac{|\psi\rangle}{\sqrt{\langle\psi|\psi\rangle}}. \tag{3.4}$$

After normalizing the first vector, we orthogonalize $H|\psi\rangle$ with respect to $|v_0\rangle$ and normalize it:

$$\begin{aligned} b_1|v_1\rangle &= |\tilde{v}_1\rangle = H|v_0\rangle - |v_0\rangle \underbrace{\langle v_0|H|v_0\rangle}_{a_0} \\ b_1 &= \sqrt{\langle\tilde{v}_1|\tilde{v}_1\rangle}. \end{aligned} \tag{3.5}$$

Multiplying Eq. (3.5) with $\langle v_1|$ we get the following relation:

$$\begin{aligned} b_1 \underbrace{\langle v_1|v_1\rangle}_{=1} &= \langle v_1|H|v_0\rangle - a_0 \underbrace{\langle v_1|v_0\rangle}_{=0} \\ b_1 &= \sqrt{\langle\tilde{v}_1|\tilde{v}_1\rangle} = \langle v_1|H|v_0\rangle. \end{aligned}$$

To compute the minimum, we now have to diagonalize the Hamiltonian in this subspace, which is given by

$$H_{K_2} = \begin{pmatrix} \langle v_0|H|v_0\rangle & \langle v_1|H|v_0\rangle \\ \langle v_1|H|v_0\rangle & \langle v_1|H|v_1\rangle \end{pmatrix} = \begin{pmatrix} a_0 & b_1 \\ b_1 & a_1 \end{pmatrix}. \tag{3.6}$$

After computing the state with the lowest energy in this basis, we can start over and apply the variational principle again. Instead of doing this, we can also extend the basis of our Krylov space by one more vector and compute the minimum in the new space. This is the idea behind the Lanczos algorithm.

3.2. Lanczos algorithm

After finding out how to compute the closest approximation to the ground state using two vectors, we want to extend it to use more orthonormal vectors and get a better result. We therefore compute new vectors of the Krylov space:

$$\begin{aligned} b_2|v_2\rangle &= H|v_1\rangle - |v_0\rangle \underbrace{\langle v_0|H|v_1\rangle}_{b_1} - |v_1\rangle \underbrace{\langle v_1|H|v_1\rangle}_{a_1} \\ b_3|v_3\rangle &= H|v_2\rangle - |v_0\rangle \underbrace{\langle v_0|H|v_2\rangle}_{=0} - |v_1\rangle \underbrace{\langle v_1|H|v_2\rangle}_{b_2} - |v_2\rangle \underbrace{\langle v_2|H|v_2\rangle}_{a_2}. \end{aligned} \tag{3.7}$$

In order to prove that $\langle v_0|H|v_2\rangle = 0$, we apply $\langle v_2|$ to Eq. (3.5) and get

$$\begin{aligned} b_1\langle v_2|v_1\rangle &= \langle v_2|H|v_0\rangle - a_0\langle v_2|v_0\rangle \\ \langle v_2|H|v_0\rangle &= 0. \end{aligned} \tag{3.8}$$

The Eq. (3.8) is also valid for every other Lanczos vector that we will construct from now on, since the basis is orthogonal

$$\begin{aligned} b_1\langle v_i|v_1\rangle &= \langle v_i|H|v_0\rangle - a_0\langle v_i|v_0\rangle \\ \langle v_i|H|v_0\rangle &= 0, \forall i \geq 2. \end{aligned} \tag{3.9}$$

Where we used the following notation $a_i = \langle v_i|H|v_i\rangle$. In general a vector v_{n+1} of the Lanczos basis is constructed as:

$$\begin{aligned} b_{n+1}|v_{n+1}\rangle &= H|v_n\rangle - \textstyle\sum_{i=0}^{n} |v_i\rangle\langle v_i|H|v_n\rangle \\ &= H|v_n\rangle - a_n|v_n\rangle - b_n|v_{n-1}\rangle - \textstyle\sum_{i=0}^{n-2} |v_i\rangle\langle v_i|H|v_n\rangle, \end{aligned} \tag{3.10}$$

where b_{n+1} contains the normalization of the vector and $\sum_{i=0}^{n} |v_i\rangle\langle v_i|H|v_n\rangle$ is the orthogonalization with respect to all previous basis vectors. If we apply a vector of the basis $\langle v_m|$ we get:

$$\begin{aligned} b_{n+1}\langle v_m|v_{n+1}\rangle &= \langle v_m|H|v_n\rangle - a_n\langle v_m|v_n\rangle - b_n\langle v_m|v_{n-1}\rangle - \textstyle\sum_{i=0}^{n-2}\langle v_m|v_i\rangle\langle v_i|H|v_n\rangle \\ b_{n+1}\delta_{m,n+1} &= \langle v_m|H|v_n\rangle - a_n\delta_{m,n} - b_n\delta_{m,n-1} - \textstyle\sum_{i=0}^{n-2} \delta_{m,i}\langle v_i|H|v_n\rangle. \end{aligned} \tag{3.11}$$

This provides us with three relevant cases:

$$\begin{aligned} m = n \quad & 0 &= \langle v_n|H|v_n\rangle - a_n \\ m = n+1 \quad & b_{n+1} &= \langle v_{n+1}|H|v_n\rangle \\ m > n+1 \quad & 0 &= \langle v_m|H|v_n\rangle \end{aligned} \tag{3.12}$$

and shows that the Hamiltonian in the Lanczos basis is tridiagonal, defined by:

$$H|v_{i-1}\rangle = b_i|v_i\rangle + a_{i-1}|v_{i-1}\rangle + b_{i-1}|v_{i-2}\rangle. \tag{3.13}$$

In this basis we can solve the Hamiltonian easily and get the information about the ground state:

$$H_{tri} = \begin{pmatrix} a_0 & b_1 & 0 & \cdots & 0 \\ b_1 & a_1 & b_2 & & 0 \\ 0 & b_2 & a_2 & b_3 & \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & & b_L & a_L \end{pmatrix}, \text{for } L \text{ steps.} \tag{3.14}$$

Building the matrix iteratively, we need to store only the previous two vectors v_{i-1}, v_i to compute another two elements of the matrix in the Lanczos basis, a_i and b_i. The Listing 3.1 illustrates the implementation of this algorithm. At each call of this function we need the previous values of the vectors v, w denoted by $\_v, \_w$ and we compute new values and also one set of new values for a, b. Using this function we compute the elements of the vectors a, b in the order given in Tab. 3.1.

iteration	1	2	...	n
Elements	a_0, b_1	a_1, b_2	...	a_{n-1}, b_n

Table 3.1.: The output of Lanczos iterations in the program

```
function lanczos.pass(H, start, maxiter)
  local w = start -- for storing the vector |v_i>
  local v = {}    -- for storing the vector |v_{i-1}>
  local aVec = {} -- storing a_0, a_1, a_{L-1}, a_i = aVec[i+1]
  local bVec = {} -- storing b_1, b_2, b_{L}
  for i = 0, #start do
    v[i] = 0
  end
  local b = 0
  for i = 1,#w do
    b = b + w[i] * w[i]
  end
  b = math.sqrt(b)
  -- here v = 0, w = b[0] |v_0>
  aVec[1], bVec[1], v, w = lanczos.doOneStep(H, v, w, b)
  -- here v = |v_0>, w = b[1] | v_1 >
  done = false
  local eps = 1e-13
  -- enter loop
  for i = 2, maxiter do
    if (math.abs(bVec[#bVec]) < eps) then
      print("Abs b got too small iter ",i, math.abs(bVec[#bVec]))
      return aVec, bVec
    end
    aVec[i], bVec[i], v, w = lanczos.doOnePass(H, v, w, bVec[i-1])
  end
  return aVec,bVec
end
```

Listing 3.1: One lanczos pass.

```
-- _w = b[i-1]|v_{i-1} >
-- _v = |v_{i-2} >
function lanczos.doOneStep(H, _v, _w, _b)
  for i = 1, #_w do
    _w[i] = _w[i] / _b     -- normalization of |v_{i-1} >
    _v[i] = _v[i] *(-_b)   -- -b[i-1] |v_{i-2} >
  end
  local v = _w
  local w = _v
  -- w = w + H*v
  local s = 0
  for i = 1,#w do
    s = 0
    for j = 1,#v do
      s = s + H[i][j] * v[j]
    end
    w[i] = s + w[i]
  end
  -- a = dot(v,w)
  local a = 0
  for i = 1,#v do
    a = a + v[i] * w[i]
  end
  -- axpy(-a,v,w): w = w - a * v
  for i = 1, #v do
    w[i] = w[i] - a * v[i]
  end
  -- b = norm2(w)
  local b = 0
  for i = 1,#v do
    b = b + w[i] * w[i]
  end
  b = math.sqrt(b)
  -- w = b[i] |v_i > , v = | v_{i-1} >
  return a,b,v,w
end
```

Listing 3.2: One step of the Lanczos method implemented in Lua.

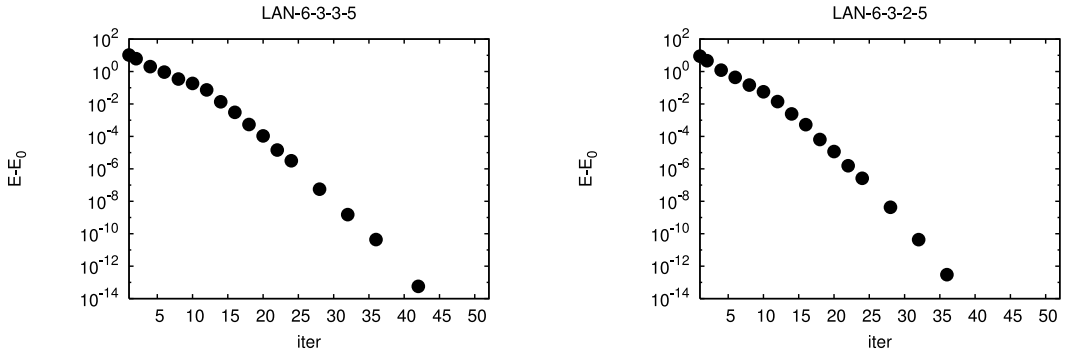
By applying the Hamiltonian H again we can get the last element of the a vector, $a_n = \langle w|\frac{H}{b_n^2}|w\rangle$.

As an improvement of this algorithm, we can check for convergence. This is useful when we only want to compute the ground state information. In the project implementation we make a difference between the runs till convergence and the runs with a given number of iterations.

3.3. Ground State Energy and Ground State Vector

By solving the truncated tridiagonal matrix, we get a good approximation of the ground state, in the Lanczos basis. In Fig. 3.1 we can see the convergence to the ground state

energy of the Lanczos method versus iteration numbers for a system with 6 sites, half filling 3.1a and for a system not at half filling in 3.1b, $t = 1$ and $U = 5$. For the half filled system, where the Hilbert space for a given number of sites has the largest dimension, we reach numerical accuracy in less than 50 iterations, a number much smaller than the dimension of the Hilbert space, which in this case is 400.



(a) Half filled system. $N_\uparrow = 3, N_\downarrow = 3$

(b) Not half filled system. $N_\uparrow = 3, N_\downarrow = 2$

Figure 3.1.: Energy convergence for a Hubbard chain with nsites = 6, t = 1, U = 5 and periodic boundary conditions for (a) half filling and (b) a system off half filling.

If we would like also to compute the ground state vector we need to perform a transformation back to the original basis. This transformation is given by the formula:

$$|\psi_0\rangle = \sum_{i=0}^{L} c_{0,i}|v_i\rangle, \tag{3.15}$$

where $c_{0,i}$ are the elements of the ground state vector in the Lanczos basis and $|v_i\rangle, i = 0 \dots L$ are the respective basis vectors. The coefficients $c_{0,i}$ are only known after solving the tridiagonal matrix gotten from the first Lanczos pass. In order to compute our ground state vector we could store the basis vectors as we compute them, which would lead to memory problems for larger systems. Our way is to perform a second Lanczos pass with the same starting vector and recompute the basis vectors to calculate the ground state vector as we go.

3.4. Spectral Function

In order to get as much information about our system as possible we want to also compute dynamical response functions. We would like to use the Lanczos method for that. The Green's function can be interpreted as the response of the system if at one point in time we

add/remove one particle and at some later time we remove/add it back. In Fourier space this is given by the equation:

$$G_{ii}(\omega) = \left\langle \psi_0 \left| c_i^\dagger \frac{1}{\omega - i\eta + (H^< - E_0)} c_i \right| \psi_0 \right\rangle + \left\langle \psi_0 \left| c_i \frac{1}{\omega - i\eta - (H^> - E_0)} c_i^\dagger \right| \psi_0 \right\rangle, \tag{3.16}$$

where $H^<, H^>$ are the finite Hamiltonians for the systems associated with $c_i|\psi_0\rangle$ and $c_i^\dagger|\psi_0\rangle$ respectively. The first term can be interpreted as the response of the system if we first remove one particle, like photoemission, when light releases an electron from the material. The second term can be seen as the inverse process of first adding one electron to the system. If we construct the tridiagonal matrices $H^<_{tri}, H^>_{tri}$ starting the Lanczos method on $c_i|\psi_0\rangle$ and $c_i^\dagger|\psi_0\rangle$ respectively, we can compute Eq. (3.16) as follows:

$$\begin{aligned} G_{ii}(\omega) &= \left\langle \psi_0 \left| c_i^\dagger \frac{1}{\omega - i\eta + \left(H^<_{tri} - E_0\right)} c_i \right| \psi_0 \right\rangle + \left\langle \psi_0 \left| c_i \frac{1}{\omega - i\eta - \left(H^>_{tri} - E_0\right)} c_i^\dagger \right| \psi_0 \right\rangle \\ &= \left[\omega - i\eta + H^<_{tri} - E_0\right]^{-1}_{00} + \left[\omega - i\eta - H^>_{tri} + E_0\right]^{-1}_{00}. \end{aligned} \tag{3.17}$$

Both parts of the Green function are given by elements of the form $\left[z \pm H^{\lessgtr}_{tri}\right]^{-1}_{00}$. This element of the resolvent can be easily calculated using the inversion by partitioning method, found in Appendix A:

$$z \pm H^{\lessgtr}_{tri} = \left(\begin{array}{c|cccc} z \pm a_0 & \pm b_1 & 0 & \cdots & 0 \\ \hline \pm b_1 & z \pm a_1 & \pm b_2 & & 0 \\ 0 & \pm b_2 & z \pm a_2 & \pm b_3 & \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & & & \end{array}\right). \tag{3.18}$$

The element we are interested in is thus given by

$$\left[z \pm \left(H^{\lessgtr}_{tri}\right)\right]^{-1}_{00} = \frac{1}{z \pm a_0 - \dfrac{b_1^2}{\left[z \pm \left(H^{\lessgtr}_{tri}\right)^{(1)}\right]_{00}}}, \tag{3.19}$$

where $\left[z \pm \left(H^{\lessgtr}_{tri}\right)^{(1)}\right]^{-1}_{00}$ is the resolvent of the lower part of the matrix:

$$z \pm \left(H^{\lessgtr}_{tri}\right)^{(1)} = \left(\begin{array}{c|cccc} z \pm a_1 & \pm b_2 & 0 & \cdots & 0 \\ \hline \pm b_2 & z \pm a_2 & \pm b_3 & & 0 \\ 0 & \pm b_3 & z \pm a_3 & \pm b_4 & \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & & & \end{array}\right). \tag{3.20}$$

This can again be solved using the same technique. In the end we get a continued fraction representation

$$\left[z \pm H_{tri}^{\lessgtr}\right]_{00}^{-1} = \cfrac{1}{z \pm a_0 - \cfrac{b_1^2}{z \pm a_1 - \cfrac{b_2^2}{\cdots}}}. \tag{3.21}$$

This continued fraction ends when the b coefficient becomes small enough. After that the following terms do not contribute any more. Using this method we can easily calculate the spectral function

$$\begin{aligned} A(\omega) &= -\frac{1}{\pi}\operatorname{Im}\left(G_{00}\left(\omega\right)\right) \\ &= -\frac{1}{\pi}\operatorname{Im}\left(\left(b_0^{<}\right)^2\left[\omega - i\eta + \left(H_{tri}^{<} - E_0\right)\right]_{00}^{-1} + \left(b_0^{>}\right)^2\left[\omega - i\eta - \left(H_{tri}^{>} - E_0\right)\right]_{00}^{-1}\right). \end{aligned} \tag{3.22}$$

as the sum of two continued fractions. We can construct an extended tridiagonal matrix for each side, which by inversion gives up the corresponding continued fraction of the Green function:

$$\hat{H}_{tri}^{<} = \begin{pmatrix} 0 & b_0^{<} & 0 & \cdots & & 0 \\ b_0^{<} & z - a_0^{<} & -b_1^{<} & & & \\ 0 & -b_1^{<} & z - a_1^{<} & -b_2^{<} & & \vdots \\ \vdots & & & \ddots & & \\ 0 & & \cdots & & -b_{L<}^{<} & z - a_{L<}^{<} \end{pmatrix},$$

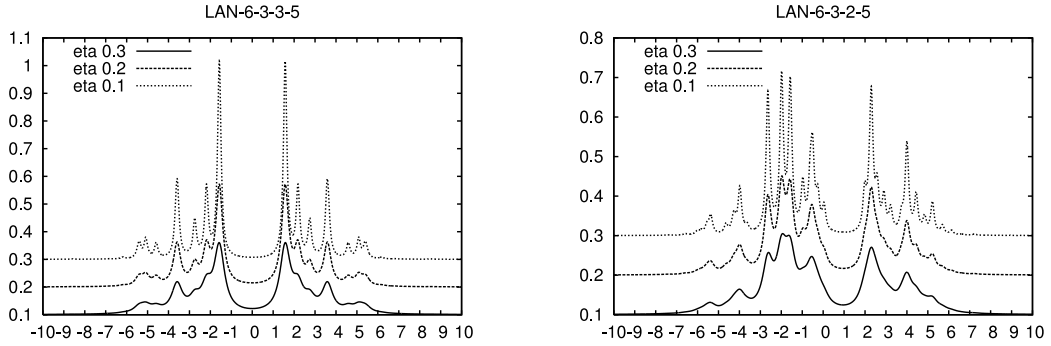
with $b_0^{<} = \sqrt{n_\alpha} = \sqrt{\langle\psi_0|cc^\dagger|\psi_0\rangle}$ the normalization factor. Analogously, we can construct the extended tridiagonal matrix for the inverse photoemission $\hat{H}_{tri}^{>}$, taking into account the sign changes and the different normalization. Using this notation the spectral function becomes

$$A(\omega) = -\frac{1}{\pi}\operatorname{Im}\left(\left[\left(\hat{H}_{tri}^{<}\right)\right]_{00}^{-1} + \left[\left(\hat{H}_{tri}^{>}\right)\right]_{00}^{-1}\right).$$

The parameter η has a small value and determines the width of the peaks. It can be interpreted as a Lorentzian broadening of the Green function. If the parameter is bigger, the peaks of the Green function become wider and overlap. This way the Green function becomes more smooth and more of the detailed features are lost, however the main appearance remains the same, as seen in Fig. 3.2.

To compute the spectral function we only have to construct the Hamiltonians and the starting vectors for the photoemission and the inverse photoemission part. After running the Lanczos solver we can calculate the value of the spectral function using the tridiagonal representation of the Hamiltonian and the continued fraction.

In Fig. 3.3 we can see the evolution of the spectral function with regard to the iterations done for the continued fraction, for the same system with 6 sites, half filling and not-half filling, $t = 1$ and $U = 5$. We can notice that we get a pretty good representation even after only 10 iterations.



(a) Half filled system. $N_\uparrow = 3, N_\downarrow = 3$

(b) Not half filled system. $N_\uparrow = 3, N_\downarrow = 2$

Figure 3.2.: Eta influence for a Hubbard chain with nsites = 6, t = 1, U = 5 and periodic boundary conditions for (a) half filling and (b) a system off half filling.

3.4.1. Moments of the spectral function

Knowing all the moments of the spectral function determines the exact and full knowledge of it. We look now at the moments of the spectral function in the Lehman representation:

$$\begin{aligned}\int_{-\infty}^{\infty} d\omega \omega^m A(\omega) &= \sum_{n=0}^{\infty} \left\langle v_0 | \psi_n^{N+1} \right\rangle \left\langle \psi_n^{N+1} | v_0 \right\rangle {E_n}^m + \sum_{n=0}^{\infty} \left\langle v_0' | \psi_n^{N-1} \right\rangle \left\langle \psi_n^{N-1} | v_0' \right\rangle {E_n}'^m \\ &\approx \left\langle v_0 \right| \left(H_{tri}^{<}\right)^m \left| v_0 \right\rangle + \left\langle v_0' \right| \left(H_{tri}^{>}\right)^m \left| v_0' \right\rangle ,\end{aligned} \tag{3.23}$$

where $|v_0\rangle$ and $|v_0'\rangle$ are the start vectors, $c_i|\psi_0\rangle$ or $c_i^\dagger|\psi_0\rangle$ for the photoemission or the inverse photoemission respectively. For simplicity from now on we will show only one equation.

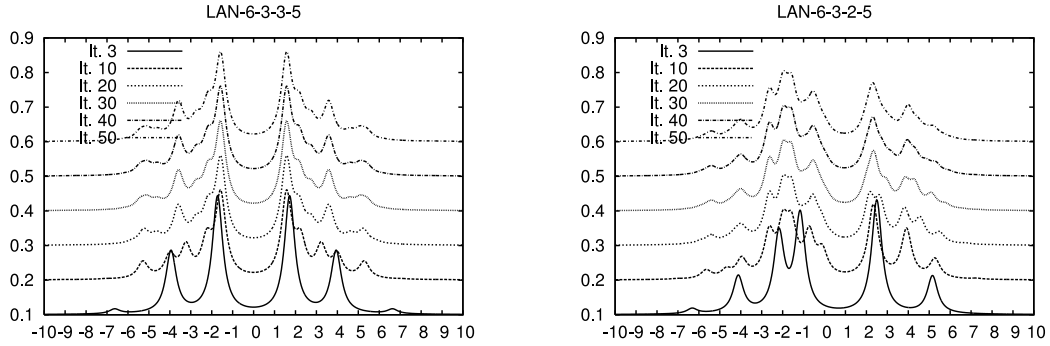
We want to figure out how many moments we can calculate exactly using a truncated Lanczos basis. We start by looking increasingly at the moments. If we want to calculate the zero moment, we only need the ground state vector. This means that even without making any Lanczos steps, for the PES or IPES Hamiltonian we get the zero moment correct if we have calculated the ground state information. For the first moment we have to calculate

$$\mu_1 = \langle v_0 | H | v_0 \rangle = a_0. \tag{3.24}$$

This is done in the first Lanczos step. For computing the second moment we use the definition of the first Lanczos vector and insert it into the moment:

$$\mu_2 = \langle v_0 | H^2 | v_0 \rangle = \langle v_0 | H(b_1 | v_1 \rangle + a_0 | v_0 \rangle) = b_1^2 + a_0^2. \tag{3.25}$$

This can be graphically shown using Fig. 3.4b. We multiply the weights of the paths connecting the left side with the right side. The lines stand for the non zero elements



(a) Half filled system. $N_\uparrow = 3, N_\downarrow = 3$

(b) Not half filled system. $N_\uparrow = 3, N_\downarrow = 2$

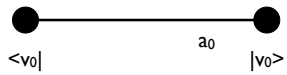
Figure 3.3.: Convergence of the spectral function for a Hubbard chain with nsites = 6, t = 1, U = 5 and periodic boundary conditions for (a) half filling and (b) a system off half filling.

of the Hamiltonian. The skewed lines represent the off-diagonal elements in the Lanczos Hamiltonian while the straight lines represent the diagonal part.

Using the graphical interpretation in Fig. 3.4 we can show that for L Lanczos steps, we get information about $2L + 1$ moments.

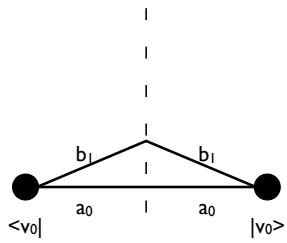
(a) μ_0

$$\mu_0 = \langle v_0|v_0\rangle = 1$$



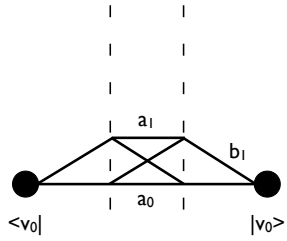
(b) μ_1

$$\mu_1 = \langle v_0|H|v_0\rangle = a_0$$



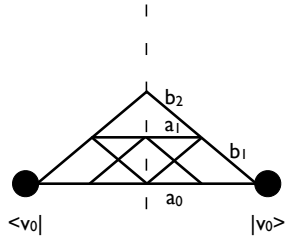
(c) μ_2

$$\begin{aligned} H|v_0\rangle &= b_1|v_1\rangle + a_0|v_0\rangle \\ \mu_2 &= \langle v_0|H^2|v_0\rangle = \langle v_0|H(b_1|v_1\rangle + a_0|v_0\rangle) = b_1^2 + a_0^2 \end{aligned}$$



(d) μ_3

$$\begin{aligned} H^2|v_0\rangle &= b_1\left(b_2|v_2\rangle + a_1|v_1\rangle + b_1|v_0\rangle\right) \\ &\quad + a_0\left(b_1|v_1\rangle + a_0|v_0\rangle\right) \\ &= b_1 b_2|v_2\rangle + b_1\left(a_0 + a_1\right)|v_1\rangle + \left(a_0^2 + b_1^2\right)|v_0\rangle \\ \mu_3 &= \langle v_0|H^3|v_0\rangle = a_0\left(a_0^2 + b_1^2\right) + b_1^2\left(a_1 + a_0\right) \end{aligned}$$



(e) μ_4

$$\mu_4 = \langle v_0|H^4|v_0\rangle = \left(b_1 b_2\right)^2 + b_1^2\left(a_0 + a_1\right)^2 + \left(a_0^2 + b_1^2\right)^2$$

Figure 3.4.: A graphical explanation linking the Lanczos coefficients to the moments.

CHAPTER 4

Dynamical Mean-field Theory

4.1. Static Mean-field Theory

To illustrate Static Mean-field Theory (SMFT) for a lattice model we consider the Ising model, which looks at spins situated on a discrete lattice with an external magnetic field. The energy of the system in a specific state, s_i is given by

$$H\{s_i\} = -J \sum_{\langle i,j \rangle} S_i S_j - B \sum_i S_i, \tag{4.1}$$

where $J > 0$ is the ferromagnetic coupling and B is the external field. For simplification we considered a system, where the ferromagnetic coupling J between neighbours is the same for every site. As an approximation we replace the interaction term with its mean value:

$$H_{mf} = -\sum_i \left(J \sum_j \langle S_j \rangle \right) S_i - B \sum_i S_i. \tag{4.2}$$

In the limit of infinitely many neighbours, by the central limit theorem, the sum over the spins becomes equal to the connectivity times the average spin. However, in this limit the

interaction energy would diverge, hence it is required to rescale $J \to J/Z$. By using this mean field, the spins become independent from each other. Due to the invariance of the lattice, we can reduce the $J\sum_{\langle ij\rangle}\langle S_j\rangle = JZm$, where Z is the coordination number and m is the magnetization of the lattice. Using statistical mechanics we can derive an equation for the magnetization, [**?**]:

$$m = \tanh\left(\beta B + \beta JZm\right). \tag{4.3}$$

This gives us a method for solving the system iteratively until we reach a self consistent magnetization. We could try to use this idea in solving our Hubbard Hamiltonian as well, by mapping the lattice onto a single site problem embedded in a dynamical bath. This way the dynamical properties of the system are not completely ignored, while the bath provides a simpler way of solving the Hamiltonian.

4.2. Self consistency loop for lattices of Fermions

In our case, we do not have localized spins, as in the Ising model, but we have the Hubbard Hamiltonian:

$$H = -t\sum_{\langle ij\rangle,\sigma} c^{\dagger}_{i\sigma}c_{j\sigma} + U\sum_{i} n_{i\uparrow}n_{i\downarrow}, \tag{4.4}$$

which also contains a kinetic term, describing the hopping of electrons between sites and U the Coulomb repulsion. We introduce the Dyson equation,

$$G_0\left(k,\omega\right)^{-1} - G_{latt}\left(k,\omega\right)^{-1} = \Sigma\left(k,\omega\right), \tag{4.5}$$

which gives the definition of the self energy, $\Sigma\left(k,\omega\right)$, as the difference between the interacting Green function $G_{latt}\left(k,\omega\right)^{-1}$ and the non-interacting Green function $G_0\left(k,\omega\right)^{-1}$. In the atomic limit, $U/t \gg 1$ the charges are localized and the kinetic term can be neglected. The system can then be reduced to independent sites, which results in a local Σ, i.e. k-independent. In the other extreme case, $U/t \ll 1$ the kinetic term takes precedence. Setting $U = 0$ we get the non-interacting solution of the system, which means that the interacting and the non-interacting Green function have the same value, leading to $\Sigma = 0$, which also is k-independent. After solving the single electron case we can use Slater determinants to get the solution of our problem.

What we are interested in is the case where the kinetic energy and the potential energy both play an important role. In the case of infinite dimensions $d \to \infty$ we can assume a k-independent Σ and the Green function becomes:

$$G_{latt}\left(k,\omega\right) = \frac{1}{\omega + \mu - \epsilon_k - \Sigma\left(k,\omega\right)} \overset{d\to\infty}{\approx} \frac{1}{\omega + \mu - \epsilon_k - \Sigma\left(\omega\right)}. \tag{4.6}$$

If we integrate over the Brillouin zone we get the local Green function:

$$G_{loc}\left(\omega\right) = \int d^3k G_{latt}\left(k,\omega\right) \approx \int d^3k \frac{1}{\omega + \mu - \epsilon_k - \Sigma\left(\omega\right)}. \tag{4.7}$$

At this point we introduce the density of states, a quantity that reflects how many states our system has for a given energy value:

$$D(\epsilon) = \int \frac{d^d k}{V_{BZ}} \delta(\epsilon - \epsilon_k), \tag{4.8}$$

where V_{BZ} is the volume of the Brillouin zone. We can then rewrite the Green function as follows:

$$G_{loc}(\omega) = \int d^3k \frac{1}{\omega + \mu - \epsilon_k - \Sigma(\omega)} = \int d\epsilon \frac{D(\epsilon)}{\omega + \mu - \epsilon - \Sigma(\omega)}. \tag{4.9}$$

In terms of the Hilbert transform:

$$\tilde{D}(\xi) \quad = \quad \int d\epsilon \frac{D(\epsilon)}{\xi - \epsilon}, \tag{4.10}$$

the Green function becomes

$$G_{loc}(\omega) = \tilde{D}(\omega + \mu - \epsilon - \Sigma(\omega)). \tag{4.11}$$

The question is how to find the right approximation for the self energy term $\Sigma(\omega)$. The idea is to solve a self consistency loop. To this end we first map the Hubbard Hamiltonian onto a non-interacting bath. The single site Anderson impurity model (SIAM), given by:

$$H_{SIAM} = \sum_{\sigma} \mu n^f_{\sigma} + U n^f_{\uparrow} n^f_{\downarrow} + \sum_{j\sigma} \epsilon_j n^c_{j\sigma} + \sum_{j\sigma} V_j \left(f^{\dagger}_{\sigma} c_{j\sigma} + c^{\dagger}_{j\sigma} f_{\sigma} \right), \tag{4.12}$$

where the first two terms describe the impurity with the on-site energy and the Hubbard U. The second term describes the on-site energies of the bath and the last term of the sum is the interaction of the bath with the impurity.

Using this SIAM Hamiltonian we can calculate the Green function on the impurity. The idea is to adjust the hybridization parameters, V_j, such that the bath can simulate the physics of the original lattice.

The method starts with an estimate for the self energy and calculates the local Green function of the lattice. We use then the Dyson equation to compute an estimate of the non-interacting bath Green function

$${G_b}^{-1}(\omega) = \Sigma(\omega) + {G_{loc}}^{-1}(\omega). \tag{4.13}$$

Using this value of the bath Green function we want to determine the hybridization parameters V_j, which form the single particle part of the SIAM Hamiltonian, as seen from Fig. 4.3a:

$$H_{SIAM} = \begin{pmatrix} \mu & V_1 & V_2 & \dots \\ V_1 & \epsilon_1 & 0 & \\ V_2 & 0 & \epsilon_2 & \\ \vdots & & & \ddots \end{pmatrix} \tag{4.14}$$

We can do this by fitting the bath Green function:

$$G_b{}^{-1}(\omega) = \omega + \mu - \sum \frac{|V_j|^2}{\omega - \epsilon_j}, \tag{4.15}$$

to the computed value in Eq. (4.13). Since the hybridization parameters enter the Green function as absolute value, we can choose them to be real values. Since we cannot treat infinite baths, we restrict ourselves to a finite number of bath sites, which in general means only an approximation of the bath Green function can be made.

After having determined the parameters we can compute the impurity Green function G_{imp}, e.g. using Lanczos and use the Dyson equation again to compute a new self energy:

$$\Sigma(\omega) = G_b{}^{-1}(\omega) - G_{imp}{}^{-1}(\omega). \tag{4.16}$$

Since we have used a finite number of bath parameters, we restart the calculation with the new computed self-energy, until we reach self consistency in the bath parameters.

Using the Hilbert transform, (4.10) and its inverse we can directly compute the bath Green function using the impurity Green function:

$$G_b{}^{-1}(\omega) = \omega + \mu + G_{imp}{}^{-1}(\omega) - R\left[G_{imp}{}^{-1}(\omega)\right], \tag{4.17}$$

which becomes important when the inverse Hilbert transform is known.

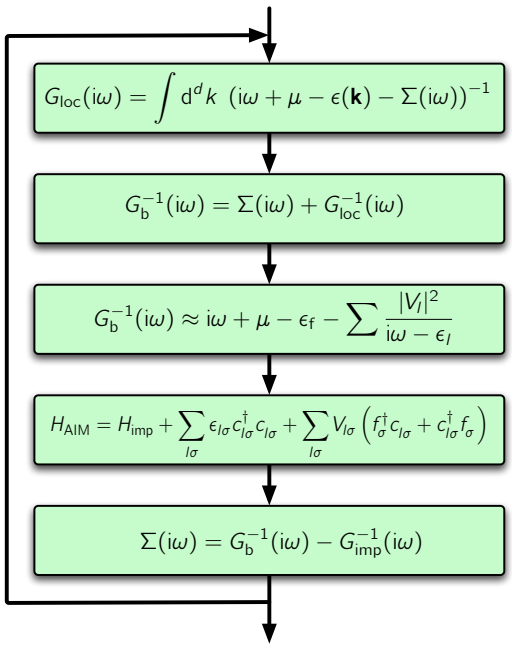


Figure 4.1.: DMFT self consistency loop, [**?**].

4.3. Bethe lattice

A special kind of lattice can be used, which can simplify the model, namely the Bethe lattice. This is a cycle free lattice with connectivity Z for each node. In this case we can compute the density of states in infinite connectivity:

$$D(\epsilon) = \frac{\sqrt{4(Z-1)\,t^2/Z - \epsilon^2}}{2\pi\,(t^2 - \epsilon/Z)} \overset{Z\to\infty}{\approx} \frac{\sqrt{4t^2 - \epsilon^2}}{2\pi t^2}. \tag{4.18}$$

The Hilbert transform of the density of states and the Green function then yield:

$$\begin{aligned} \tilde{D}(\xi) &= \frac{\xi - \sqrt{\xi^2 - 4t^2}}{2t^2}, \\ G_{loc}(\omega) &= \tilde{D}(\omega + \mu - \Sigma(\omega)) \end{aligned}$$

The reciprocal Hilbert transformation gives us the relation:

$$R\left[\tilde{D}(\xi)\right] = \xi \Leftrightarrow R[G(\omega)] = t^2 G(\omega) + G^{-1}(\omega). \tag{4.19}$$

Inserting this equation in Eq. (4.17), we get:

$$G_b^{-1}(\omega) = \omega + \mu + G_{imp}^{-1}(\omega) - R\left[G_{imp}^{-1}(\omega)\right] = \omega + \mu - t^2 G_{imp}(\omega). \tag{4.20}$$

This simplifies the self consistency cycle for the Bethe lattice, connecting the impurity Green function directly to the bath Green function.

4.3.1. Parameter choice

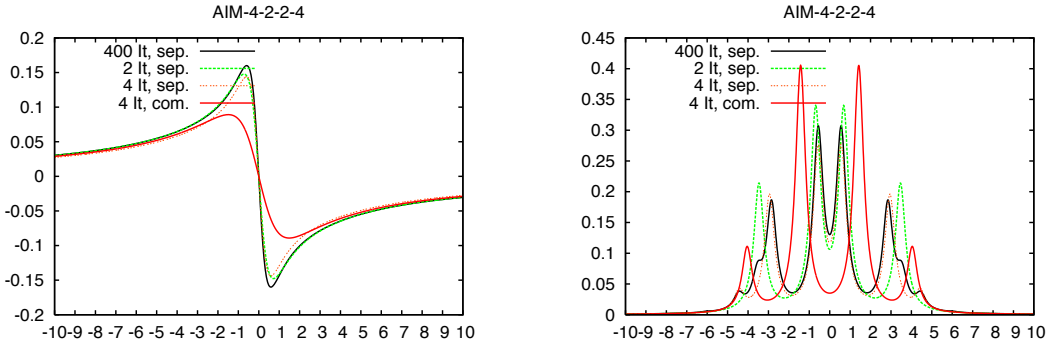
We have to find a way to calculate the bath parameters using the impurity Green function. We will first consider the method proposed by Michael Caffarel and Werner Krauth, [**?**]. There we use the Green function for the Anderson impurity model found in Eq. (4.15) and set it equal to the formula for the Bethe lattice in Eq. (4.20):

$$G_b^{-1}(\omega) = \omega + \mu - t^2 G_{imp}(\omega) \overset{\text{fit}}{\approx} \omega + \mu - \sum \frac{|V_j|^2}{\omega - \epsilon_j}. \tag{4.21}$$

The best approximation just fits $\sum \frac{|V_j|^2}{\omega - \epsilon_j}$ to $t^2 G_{imp}(\omega)$. Note that this is done for imaginary frequencies $\omega \in \mathbb{C}$, since the Green function on the imaginary axis is smoother and can be fitted easier as seen in Fig. 4.2.

Another method developed by Si et.al, [**?**] uses the continued fraction representation of the Green function, as in Eq. (3.21). To this end we write the impurity Green function out in its photoemission and inverse photoemission part as follows:

$$G_b^{-1}(\omega) = \omega + \mu - t^2 \left(G_{imp}^{<}(\omega) + G_{imp}^{>}(\omega)\right). \tag{4.22}$$



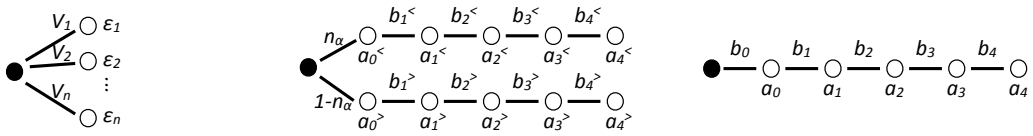
(a) $A(i\omega)$ **(b)** $A(\omega - i\eta)$

Figure 4.2.: Spectral function for self-consistency parameters using a Bethe lattice with 4 sites, $N_\uparrow = N_\downarrow = 2$, $t = 1$ and $U = 4$ on(a) the imaginary axis and (b) on the real axis, with $\eta = 0.3$.

Using the Lanczos method we can cut off the continued fraction to get a finite number of bath sites.

$$\begin{aligned} \left(G^{<}_{imp}\right)(\omega) &= \cfrac{n_\alpha}{\omega + a_0^{<} - \cfrac{b_1^{<2}}{\omega + a_1^{<} - \cdots}} \\ \left(G^{>}_{imp}\right)(\omega) &= \cfrac{1 - n_\alpha}{\omega - a_0^{>} - \cfrac{b_1^{>2}}{\omega - a_1^{>} - \cdots}}. \end{aligned} \tag{4.23}$$

We can observe that the impurity Green function represents actually the Green function of a non-interacting SIAM Hamiltonian, as seen in Fig 4.3b. The Equations (4.23) can become an exact approximation in the case of infinite number of bath sites, because then we would fit all moments of the Green function. However we restrict ourselves to a fixed number of parameters $a_i^{</>}, b_i^{</>}$. These are the parameters we use in this case. They can be interpreted as the on site energies of the bath site i, a_i and the hopping element, b_i, between site $i-1$ and site i. The number of parameters $L^{>}, L^{<}$ are in direct correlation with the number of moments that we fitted. However this moment fitting only gives us back the first $2 \cdot min(L^{<}, L^{>}) + 1$ moments for the entire Green function G_{imp}. A better idea would be to first combine the two continued fractions and fit all bath parameters at once.



(a) Parameter fitting

(b) Moment fitting, two separate continued fractions

(c) Moment fitting, combined continued fractions

Figure 4.3.: Different types for interpreting the self consistency loop for the Bethe lattice: (a) the Anderson impurity model, used with parameter fitting (b) moment fitting done via continued fractions (c) ideal combination of two continued fractions into one bath.

CHAPTER 5

Combined Continued Fractions

5.1. Problem Statement

In oder to compute the parameters of a DMFT calculation we are looking at the moments of the spectral function. Out of statistical physics we know that a function is entirely characterized if all its moments are known. The limitation that we have is the number of parameters that describe our impurity model. Therefore we want to use this given number to get as much information out of the spectral function as possible, using as few parameters as possible.

5.2. Approach

We look back to the Lanczos algorithm. We notice that at every step, using this steepest descent method, we apply the Hamiltonian to our vector, gaining information about 2 extra moments. If we now look at the photoemission and the inverse photoemission part of the Green function, even if we would calculate them to any given precision, the number of parameters we can feed back into our self-consistency loop is limited. If we assume that we compute the same number of steps on each side n, we would approximate only $2n+1$ moments correctly in the combination.

For the photoemission and the inverse photoemission part of the Green function we start a Lanczos pass with the vectors $c\ |\psi_0\rangle$ and $c^\dagger|\psi_0\rangle$ respectively. We could combine the two vectors beforehand and compute the total Green function in one go. We will use the vector $(c^\dagger + c)|\psi_0\rangle$ as starting point. We first show that the vector is normalized, by showing that the normalization reduces to the commutation relation:

$$\begin{aligned}(c+c^\dagger)(c^\dagger+c) &= (cc + c^\dagger c + c^\dagger c + c^\dagger c^\dagger) = \\ &= (c^\dagger c + c^\dagger c) = \{c, c^\dagger\} = 1.\end{aligned}$$

To show that the solution given by this new start vector is the Green function, we split the vector $(c+c^\dagger)|\psi_0\rangle = c|\psi_0\rangle + c^\dagger|\psi_0\rangle$. Since the two pieces live in different Hilbert spaces, they are orthogonal and the two parts of the equation do not interfere. The solution of the Lanczos pass with this new starting vector is thus equivalent to the Green function. Solving it for n steps we get the same Green function back, as we would using the photoemission and the inverse photoemission continued fractions for n steps. The Hamiltonian associated with this starting vector is composed of the photoemission and the inverse photoemission part of the Hamiltonian, as follows:

$$H^c = \left(\begin{array}{c|c} H^<_{tri} & 0 \\ \hline 0 & -H^>_{tri} \end{array}\right).$$

If we extend the Hamiltonian with the appropriate normalization we get the exact Green function by starting the Lanczos pass to compute the continued fraction of the following matrix:

$$\hat{H}^c = \left(\begin{array}{c|ccc|ccc} 0 & \sqrt{n_\alpha} & 0 & \cdots & \sqrt{1-n_\alpha} & 0 & \cdots \\ \hline \sqrt{n_\alpha} & & & & & & \\ 0 & & & & & & \\ \vdots & & z + H^<_{tri} - E_0 & & & 0 & \\ 0 & & & & & & \\ \hline \sqrt{1-n_\alpha} & & & & & & \\ 0 & & & & & & \\ \vdots & & 0 & & & z - (H^>_{tri} - E_0) & \\ 0 & & & & & & \end{array}\right).$$

Using this combined Hamiltonian we hope to preserve more moments of the Green function when re-iterating the DMFT loop even with a small number of steps. For example, if we

limit our bath to n sites, we have a total of $2n$ parameters that we can adjust. If we use the moment fitting procedure, we gain information about only n moments in total. Using the combined continued fraction procedure we get access to higher moments, up to a maximum of $2n$ moments.

5.3. Implementation

For implementing the DMFT loop for the Bethe lattice using combined continued fractions, we need a couple of simple functions. A function computing one set of combined coefficients is described in Listing 5.1. This function assumes the previous computation of the a, b values of the tridiagonal Hamiltonian up to the number of elements we expect to have in the combined Hamiltonian.

As an example, if we were to use a 6 site bath at half filling, in the case of the moment fitting we would use 3 sites for each side of the Green function, with 2 parameters per site, cf. Fig. 4.3b. To this end we need to compute only the a, values of the Hamiltonians. If we use the combined continued fraction method, we compute first 6 parameters for each side of the Green function, combine them to an H^c Hamiltonian and compute the parameters for all 6 sites from the continued fraction of the combined Hamiltonian, cf. Fig. 4.3c.

```
function DMFT_pass(nsites, nup, ndn, Uvec, tvec, eps, largeIter, smallIter,
    pbc, sysType)

  -- ground state information needed
  -- apes, bpes, aipes, bipes  needed
  [...]

  -- calculate diagonal (a1, a2)  and off diagonal (b1, b2) of H_c
  a1={}; a2={}
  b1={npes}; b2={nipes}
  for i=1,#apes do a1[i]=apes[i] end
  for i=1,#aipes do a2[i]=aipes[i] end
  for i=1,#bpes do b1[i+1]=bpes[i] end
  for i=1,#bipes do b2[i+1]=bipes[i] end

  maxiter = 2*smallIter+1
  aVec, bVec = combineGF(a1,b1,a2,b2,maxiter)

  str = sysType.."_"..base.."_"..Uvec[1].."_"..2*smallIter.."com"
  fd = io.open(str..".out","w+")
  fd:write("GFGF ",gse," ", bVec[1],"\n")
  fd:write("GF ",0," ",aVec[#aVec]," ",0," ",#aVec,"\n")
  for i = 1,#aVec-2 do
    k = #aVec-i
    fd:write("GF ",i," ",aVec[k]," ",bVec[k]," ",k,"\n")
  end
  fd:flush()

  fd = io.open(str..".dat","w+")
  spectralFunction.continuedFractionDMFT(str..".out", xmin, xmax, eta, fd)
  fd:flush()
```

```
32    local tvec_new = {}
      local eps_new = {mu}
34    for i = 2,#aVec-1 do
        tvec_new[i-1] = bVec[i-1]
36      eps_new[i] = aVec[i]
      end
38    [...]
  end
```

Listing 5.1: One DMFT iteration.

5.4. Combined Hamiltonian Checks

5.4.1. Symmetric Case

Let us look at the electron-hole symmetric case first. For such a system, the spectral function is symmetric about the chemical potential, μ. Assuming we have found an Eigenvector ψ_E corresponding to the eigenenergy E. We construct the eigenvector for the energy level $-E$ by alternating the signs of the coefficients in the vector ψ_E as follows:

$$\psi_E = \begin{pmatrix} \vdots \\ c_j \\ \vdots \end{pmatrix} \quad \psi_{-E} = \begin{pmatrix} \vdots \\ (-1)^j c_j \\ \vdots \end{pmatrix}.$$

We want to see what condition we have on our coefficients in the tridiagonal representation. We start by looking at the general case, for index i. The tridiagonal representation of the matrix is the following

$$\hat{H}^c_{tri} = \begin{pmatrix} \alpha_{-1} & \beta_0 & 0 & & \cdots & & 0 \\ \beta_0 & \alpha_0 & \beta_1 & & & & \\ 0 & \beta_1 & \alpha_1 & \beta_2 & & & \\ \vdots & & & \ddots & & & \vdots \\ & & & \beta_i & \alpha_i & \beta_{i+1} & \\ & & & & & \ddots & 0 \\ 0 & & & & \cdots & \beta_n & \alpha_n \end{pmatrix}.$$

For the two eigenvectors the line index $i+1$ we have the following equations for $(H \cdot \psi_{-e})_i$ and $(H \cdot \psi_{\ e})_i$ respectively:

$$\begin{array}{rrrl} (-1)^i \beta_i \cdot c_i + (-1)^{i+1} \alpha_{i+1} \cdot c_{i+1} + & (-1)^{i+2} \beta_{i+2} \cdot c_{i+2} = & -e\,(-1)^{i+1} \alpha_{i+1} \cdot c_{i+1} & \\ \beta_i \cdot c_i + & \alpha_{i+1} \cdot c_{i+1} + & \beta_{i+2} \cdot c_{i+2} = & e\,\alpha_{i+1} \cdot c_{i+1} \\ \hline & 2 \cdot \alpha_{i+1} \cdot c_{i+1} & = & 0 \end{array} \tag{5.1}$$

In Eq. (5.1) we have inserted no restrictions on the index i. This is valid also for the limiting cases, the first and the last line. This means that for a symmetric system the tridiagonal composed matrix has all $\alpha_i = 0$.

5.4.2. Numerical Checks

There are some interesting checks we can do to our combined Hamiltonian. We will start by doing a couple of Lanczos iterations by hand to illustrate them. Without loss of generality we consider a simpler starting matrix. These steps can be extended to an arbitrary size for the combined Hamiltonian.

$$
\begin{aligned}
\hat{H}^c &= \left(\begin{array}{c|cc|cc} 0 & b_0^< & 0 & b_0^> & 0 \\ \hline b_0^< & a_0^< & b_1^< & 0 & 0 \\ 0 & b_1^< & a_1^< & 0 & 0 \\ \hline b_0^> & 0 & 0 & -a_0^> & b_1^> \\ 0 & 0 & 0 & b_1^> & a_1^> \end{array}\right) & |v_0\rangle &= \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\
b_0^< &= \sqrt{n_\alpha} & b_0^> &= \sqrt{1-n_\alpha} \\
\hat{H}^c|v_0\rangle &= \begin{pmatrix} 0 \\ b_0^< \\ 0 \\ b_0^> \\ 0 \end{pmatrix} & \langle\tilde{v}_1|\tilde{v}_1\rangle &= n_\alpha + 1 - n_\alpha = 1 \\
\alpha_{-1} &= \langle v_0|\hat{H}^c|v_0\rangle = 0 & \beta_0 &= \langle v_1|\hat{H}^c|v_0\rangle = 1 \\
|v_1\rangle &= \begin{pmatrix} 0 \\ b_0^< \\ 0 \\ b_0^> \\ 0 \end{pmatrix} & \hat{H}^c|v_1\rangle &= \begin{pmatrix} & 1 \\ a_0^< & b_0^< \\ b_1^< & b_0^< \\ -a_0^< & b_0^> \\ b_1^> & b_0^> \end{pmatrix}
\end{aligned}
\tag{5.2}
$$

From Eq. (5.2) we conclude that for any combined Hamiltonian the parameters α_{-1} and β_0 have the values 0 and 1 respectively. This can be used as a check for the results given by the code. In a more general case where we also use the chemical potential μ we get that $\alpha_{-1} = \mu$. The parameter β_0 represents the sum rule for the spectral function.

5.4.3. Qualitative Checks

We can check the quality of our computations by comparing spectral functions. For example, we take a half filled Hubbard chain with periodic boundary conditions and calculate the Green function using n Lanczos iterations for each separate part. If we were to perform $2n$ steps for the combined Hamiltonian we are supposed to get the same result. This can be seen in Fig. 5.1., where we have a 6 site Hubbard chain, half-filling with $U = 4$. We have run the separate calculation for 3 and 6 steps and the combination for 12 steps.

As a comparison, we look at Fig. 5.2, where we have the same system, but using the combined Hamiltonian we only calculated 6 parameters. Notice that the spectral functions now differ between the 6 separate iterations and the 12 combined ones. However, the moments

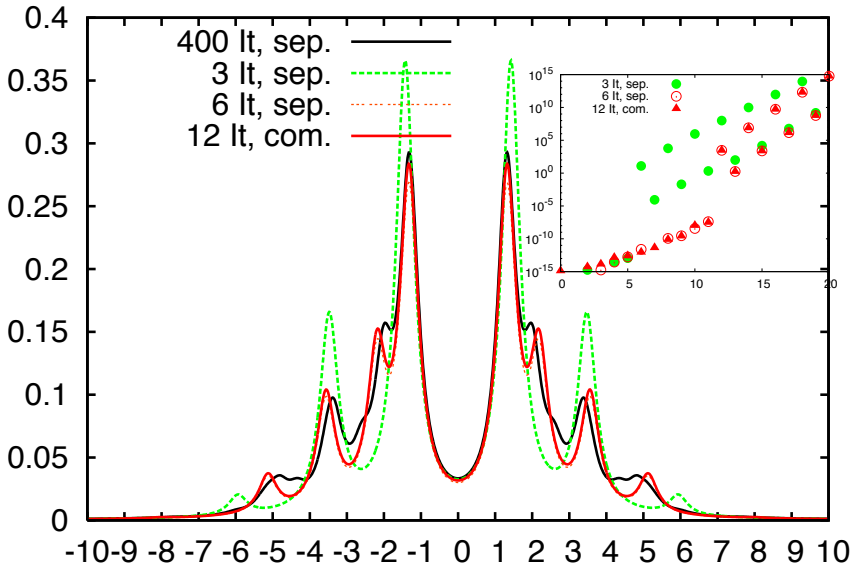


Figure 5.1.: Checking the quality of the combined Hamiltonian: the spectral function calculated with separate sums and with the combined Hamiltonian using only 6 parameters for each part and the moments of the two methods (separate sums and combined Hamiltonian). Hubbard chain with 6 sites, half filling, U = 4, t = 1 and periodic boundary conditions.

are better preserved than the spectral function with the same number of parameters, the 3 separate iterations simulation.

5.4.4. Stability Analysis

DMFT self-consistency means that the parameters for the bath do not change anymore from iteration to iteration. We investigate here what the precision is to which the parameters have to coincide between iterations. We take therefore a given set of parameters and add noise to them. Using the noisy output we compute the spectral function and compare it to the original one.

In Fig. 5.3 we can see the output of this function. We can observe that the errors up to 10^{-3} do not alter the output visibly. In the code we therefore use a stopping criterion of 10^{-5}, which should give sufficiently good results.

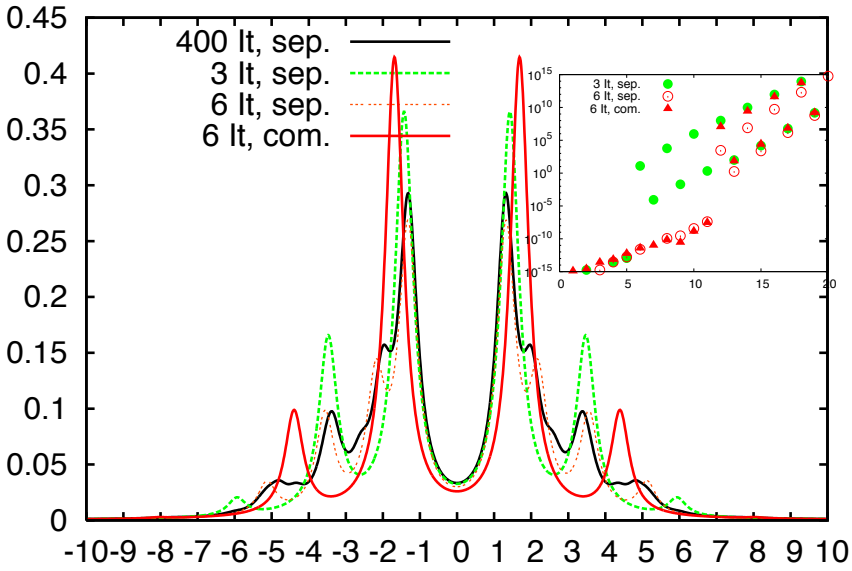
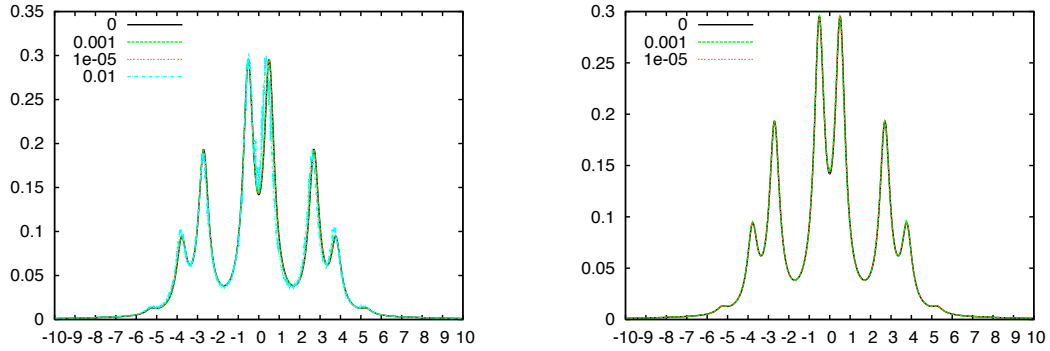


Figure 5.2.: Checking the quality of the combined Hamiltonian with fewer Lanczos steps: the spectral function calculated with separate sums and with the combined Hamiltonian and the moments of the two methods (separate sums and combined Hamiltonian). Hubbard chain with 6 sites, half filling, U = 4, t = 1 and periodic boundary conditions.

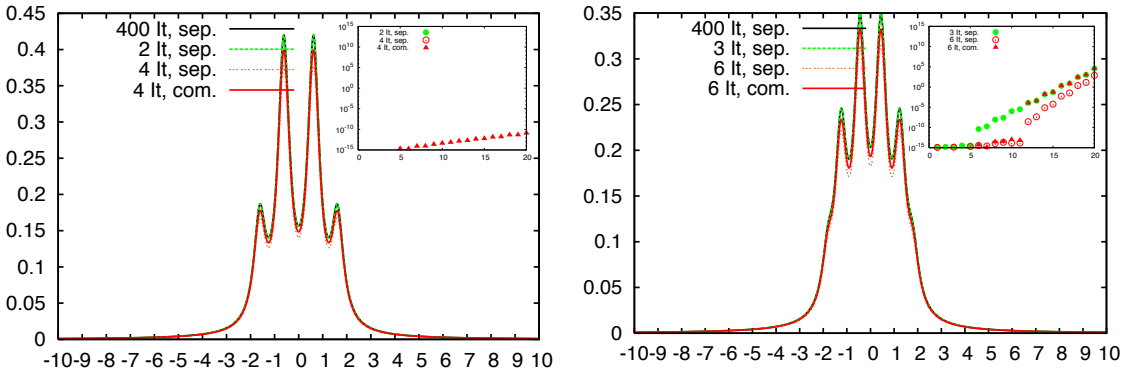
5.5. Parameter convergence

The combined continued fraction method is used on a bath with a half filled system and different number of bath sites (4 and 6 sites) and different values for U, $U = 0, 2, 8$. Figures 5.4 ($U = 0$), 5.5 ($U = 2$) and 5.6 ($U = 8$) show the spectral functions for increasing number of bath sites for a given value of U. We could expect that the number of bath sites determines how many moments of the DMFT spectral function we can calculate reliably. To check this, we compare the Green function parameters α,β for the runs with 4 and 6 bath sites. Remember that $\alpha = 0$ for half-filling. We find that the self consistent parameters for β agree well up to $n = 4$, while they start to differ for larger n, as seen in Fig. 5.7. This suggests that increasing the bath we could improve the DMFT Green function moment-by-moment.



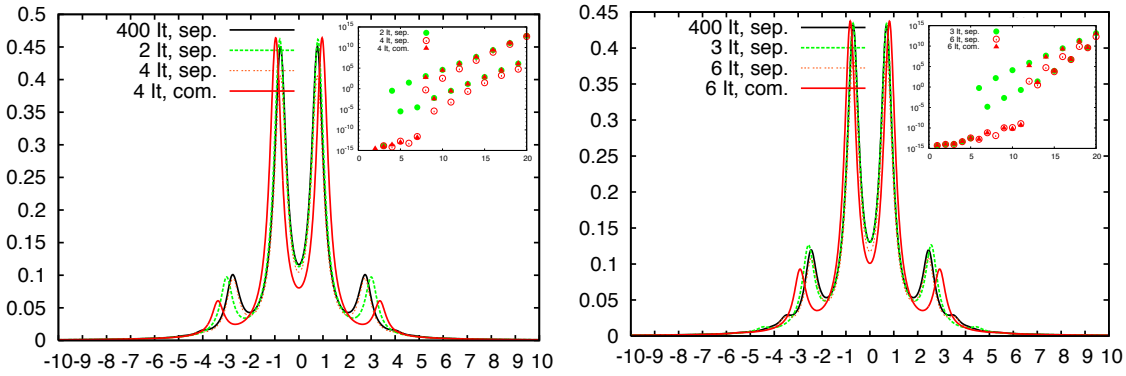
(a) $\epsilon = 0, 0.01, 0.001, 0.00001$ **(b)** $\epsilon = 0, 0.001, 0.00001$

Figure 5.3.: Stability analysis adding random error to the parameters, for different sets of values for the amplitude of the noise.



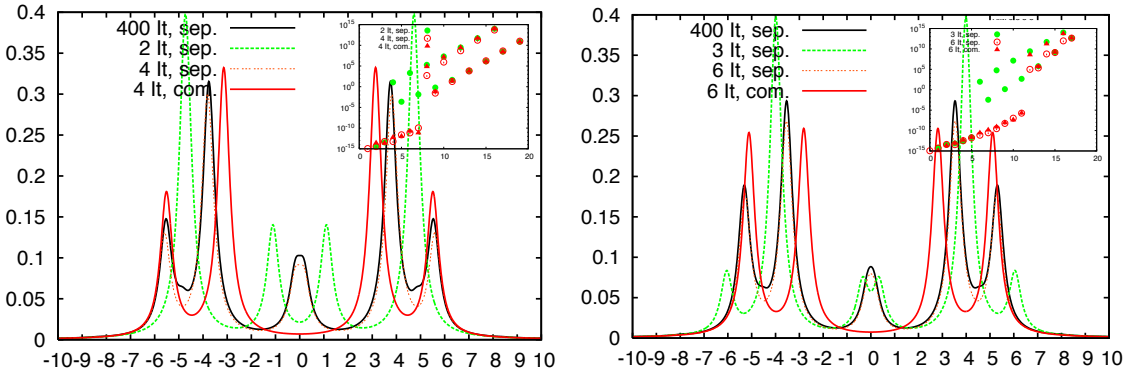
(a) $sites = 4, N_\uparrow = 2, N_\downarrow = 2$ **(b)** $sites = 6, N_\uparrow = 3, N_\downarrow = 3$

Figure 5.4.: Spectral function and moment difference to a large simulation at self consistency for a Bethe lattice with half filled bath of (a) 4 bath sites and (b) 6 bath sites, for $U = 0$.



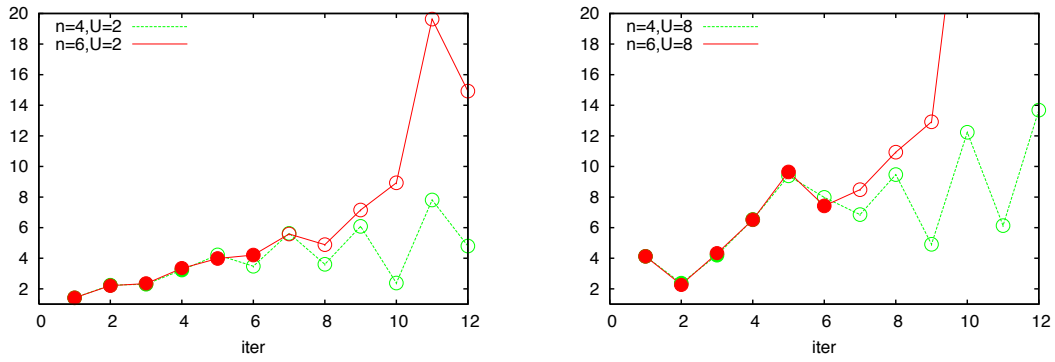
(a) $sites = 4, N_\uparrow = 2, N_\downarrow = 2$ **(b)** $sites = 6, N_\uparrow = 3, N_\downarrow = 3$

Figure 5.5.: Spectral function and moment difference to a large simulation at self consistency for a Bethe lattice with half filled bath of (a) 4 bath sites and (b) 6 bath sites, for $U = 2$.



(a) $sites = 4, N_\uparrow = 2, N_\downarrow = 2$ **(b)** $sites = 6, N_\uparrow = 3, N_\downarrow = 3$

Figure 5.6.: Spectral function and moment difference to a large simulation at self consistency for a Bethe lattice with half filled bath of (a) 4 bath sites and (b) 6 bath sites, for $U = 8$.



(a) $U = 2, \beta$ parameter

(b) $U = 8, \beta$ parameter

Figure 5.7.: β self-consistent parameters for a Bethe lattice with a half-filled bath with 4 and 6 sites for (a) $U = 2$ and (b) $U = 8$. The filled symbols are the self consistently determined parameters, the empty symbols are parameters determined with the self consistent bath.

CHAPTER 6

Conclusions

In this thesis we have looked at two ways of reducing the infinite Hubbard Hamiltonian to a manageable size. The first method is to cut out a finite lattice and use the periodicity of the solid structure we are looking at, while the second one projects the many body Hamiltonian onto a single site connected to a dynamical bath.

By cutting the lattice to a finite size, we introduce unwanted finite-size effects. To find a way to characterize the lattices used, we introduce a set of properties. A d-dimensional cluster is defined by d vectors. We have shown that the same lattice can be spanned by different combinations of vectors. To describe a cluster we have investigated properties like: the squareness, the imperfection and the bipartiteness. The squareness is susceptible to the changes of these vectors. It is a dimensionless measure of length that tells us how "close" the cell spanned by those vectors is to an actual square. Using unitary transformations we can find the combination of vectors that has the smallest squareness. This is the combination that we will use to compute all other properties of the lattice. Other important properties are the imperfection, which gives us a measure of the missing nearest-neighbours, the symmetry group the lattice belongs to and if the lattice is bipartite or not. All this can be done using a small Lua script, having no need for high performance computing, only the diagonalization of of the Hamiltonian is computationally intensive. This is where our easy to use interface between Lua and C comes in handy.

Another method for reducing the size of the Hamiltonian is using the dynamical mean field theory (DMFT), which maps the system onto a single impurity site embedded in a dynamical bath. Using this method we have to solve a self consistency loop to get the appropriate parameters for the bath. We need to calculate the local Green function for our lattice and connect it via the Dyson equation to the Green function of the bath, assuming a local self-energy Σ. The two traditional methods of computing the parameters of the bath is either by fitting or by preserving the moments with continued fractions. In this thesis

we introduce a different approach by combining the two parts of the Green function and preserving twice as many moments. The next step of development is to devise a method for calculating a continued fraction representation of the self-energy and use it to solve systems other than the Bethe lattice and compare the results with existing calculations.

Both methods require solving a many body system, which is here done using the Lanczos method. We can use Lua scripts to make the program more flexible, the only part that needs to be very efficient is the solver. We choose to use a C++ implementation of the Lanczos method. Larger systems require more memory since the dimension of the Hilbert space increases exponentially. The way to overcome this obstacle is to use a massively parallel implementation of the code and compute larger problems on distributed systems. The part of the implementation that needs to be parallelized is the C++ implementation where the many body Hamiltonian resides, meaning that the Lua scripts will still work connecting them through the interface to other C++ functions.

APPENDIX A

Inversion by partitioning

The method of inversion by partitioning can be used in case of computing the Green function very efficiently, however it is not restricted to such use, [**?**]. We describe here the method shortly. We start with an $n \times n$ matrix A that can be partitioned as follows:

$$A = \begin{pmatrix} P & Q \\ R & S \end{pmatrix}, \tag{A.1}$$

with P a $p \times p$ matrix and S a $s \times s$ matrix, $p + s = n$. The inverse of A is also partitioned as follows:

$$A^{-1} = \begin{pmatrix} \tilde{P} & \tilde{Q} \\ \tilde{R} & \tilde{S} \end{pmatrix}, \tag{A.2}$$

with $\tilde{P}$ a $p \times p$ matrix and $\tilde{S}$ a $s \times s$ matrix, $p + s = n$. The partition matrices of the inverse can be computed via the formula:

$$\begin{aligned} \tilde{P} &= \left(P - Q \cdot S^{-1} \cdot R\right)^{-1} \\ \tilde{Q} &= -\left(P - Q \cdot S^{-1} \cdot R\right)^{-1} \cdot \left(Q \cdot S^{-1}\right) \\ \tilde{R} &= \left(S^{-1} \cdot R\right) \cdot \left(P - Q \cdot S^{-1} \cdot R\right)^{-1} \\ \tilde{Q} &= S^{-1} + \left(S^{-1} \cdot R\right) \cdot \left(P - Q \cdot S^{-1} \cdot R\right)^{-1} \cdot \left(Q \cdot S^{-1}\right) \end{aligned} \tag{A.3}$$

APPENDIX B

Equations notation

B.1. Hamiltonians

B.1.1. Photoemission (PES) and Inverse Photoemission (IPES) Hamiltonians

Output Lanczos on PES Hamiltonian:

$$H^{<}_{tri} = \begin{pmatrix} a^{<}_0 & b^{<}_1 & 0 & \cdots & 0 \\ b^{<}_1 & a^{<}_1 & b^{<}_2 & & \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & b^{<}_{L^<} & a^{<}_{L^<} \end{pmatrix} =$$

$$= \begin{pmatrix} apes[1] & bpes[1] & 0 & \cdots & 0 \\ bpes[1] & apes[2] & bpes[2] & & \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & bpes[L] & apes[L+1] \end{pmatrix}$$

Extended tridiagonal representation of PES Hamiltonian:

$$\hat{H}^{<}_{tri} = \begin{pmatrix} 0 & \sqrt{n_\alpha} & 0 & \cdots & 0 \\ \sqrt{n_\alpha} & z + a^{<}_0 - E_0 & b^{<}_1 & & \\ 0 & b^{<}_1 & z + a^{<}_1 - E_0 & b^{<}_2 & \vdots \\ \vdots & & & \ddots & \\ 0 & & \cdots & b^{<}_{L^<} & z + a^{<}_{L^<} - E_0 \end{pmatrix}$$

Output Lanczos on IPES Hamiltonian:

$$H^{>}_{tri} = \begin{pmatrix} a^{>}_0 & b^{>}_1 & 0 & \cdots & 0 \\ b^{>}_1 & a^{>}_1 & b^{>}_2 & & \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & b^{>}_{L^>} & a^{>}_{L^>} \end{pmatrix} =$$

$$= \begin{pmatrix} aipes[1] & bipes[1] & 0 & \cdots & 0 \\ bipes[1] & aipes[2] & bipes[2] & & \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & bipes[L] & aipes[L+1] \end{pmatrix}$$

Extended tridiagonal representation of IPES Hamiltonian:

$$\hat{H}^{>}_{tri} = \begin{pmatrix} 0 & \sqrt{1-n_\alpha} & 0 & \cdots & 0 \\ \sqrt{1-n_\alpha} & z - (a^{>}_0 - E_0) & b^{>}_1 & & \\ 0 & b^{>}_1 & z - (a^{>}_1 - E_0) & b^{>}_2 & \vdots \\ \vdots & & & \ddots & \\ 0 & & \cdots & b^{>}_{L^>} & z - \left(a^{>}_{L^>} - E_0\right) \end{pmatrix}$$

Lanczos iter	1	2	...	n
PES	$a^{<}_0, b^{<}_1$	$a^{<}_1, b^{<}_2$	...	$a^{<}_{n-1}, b^{<}_n$
IPES	$a^{>}_0, b^{>}_1$	$a^{>}_1, b^{>}_2$	...	$a^{>}_{n-1}, b^{>}_n$

Table B.1.: The output of Lanczos iterations

B.1.2. Combined Hamiltonians

Input to Lanczos for combined Hamiltonians:

$$\hat{H}^c = \left(\begin{array}{c|ccc|ccc} 0 & \sqrt{n_\alpha} & 0 & \cdots & \sqrt{1-n_\alpha} & 0 & \cdots \\ \hline \sqrt{n_\alpha} & & & & & & \\ 0 & & z + H^<_{tri} - E_0 & & & 0 & \\ \vdots & & & & & & \\ 0 & & & & & & \\ \hline \sqrt{1-n_\alpha} & & & & & & \\ 0 & & 0 & & & z - (H^>_{tri} - E_0) & \\ \vdots & & & & & & \\ 0 & & & & & & \end{array}\right)$$

Output Lanczos on combined Hamiltonian:

$$\hat{H}^c_{tri} = \left(\begin{array}{ccccc} \alpha_{-1} & \beta_0 & 0 & \cdots & 0 \\ \beta_0 & \alpha_0 & \beta_1 & & \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & \beta_{Lc} & \alpha_{Lc} \end{array}\right) \Longrightarrow$$

$$H^c_{tri} = \left(\begin{array}{ccccc} \alpha_0 & \beta_1 & 0 & \cdots & 0 \\ \beta_1 & \alpha_1 & \beta_2 & & \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & \beta_{Lc} & \alpha_{Lc} \end{array}\right)$$

B.2. Continued Fractions and Spectral Functions

Given a Hamiltonian in tridiagonal form:

$$H_{tri} = \left(\begin{array}{ccccc} a_0 & b_1 & 0 & \cdots & 0 \\ b_1 & a_1 & b_2 & & \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & b_L & a_L \end{array}\right)$$

The contribution to the Green Function is given by the continued fraction:

$$CF(z - H_{tri}) = (z - H_{tri})^{-1}_{00} = \cfrac{1}{z - a_0 - \cfrac{b_1^2}{z - a_1 - \cfrac{b_2^2}{\cdots - \cfrac{b_L^2}{z - a_L}}}}$$

Green function using continued fractions of IPES and PES Hamiltonians:

$$\begin{aligned} G_{00}(z) &= n_\alpha \cdot CF(z + H^{<}_{tri}) + (1 - n_\alpha) \cdot CF(z - H^{>}_{tri}) \\ &= (b^{<}_0)^2 \cdot CF(z + H^{<}_{tri}) + (b^{>}_0)^2 \cdot CF(z - H^{>}_{tri}) \\ &= CF^{-1}(z - \hat{H}^{>}_{tri}) - CF^{-1}(z + \hat{H}^{<}_{tri}) \end{aligned}$$

Green function using continued fraction of combined Hamiltonian:

$$\begin{aligned} G_{00}(z) &= \cfrac{\beta_0^2}{z - \alpha_0 - \cfrac{\beta_1^2}{z - \alpha_1 - \cfrac{\beta_2^2}{\cdots - \cfrac{\beta_{Lc}^2}{z - \alpha_{Lc}}}}} \\ &= \beta_0^2 \cdot CF(z - H^c_{tri}) \\ &= CF^{-1}(z - \hat{H}^c_{tri}) - z + \alpha_{-1} \end{aligned}$$

APPENDIX C

Lorentzian

C.1. Residue Theorem

If $f(z)$ has an isolated singularity at z_0, then there exists an R such that the Laurent expansion $f(z) = \sum_{k=-\infty}^{\infty} a_k \, (z - z_0)^k$ converges for $0 < |z - z_0| < R$. The residue of $f(z)$ at z_0 is the coefficient of $1/(z - z_0)$ in the expansion and is given by

$$a_{-1} = \frac{1}{2\pi i} \int_{|z-z_0|=R/2} f(z)dz,$$

where the integration is in the positive sense.

C.2. Integral calculation with the residuum theorem

Example:

$$I = \int_{-\infty}^{\infty} \frac{1}{x^2+1} dx = \lim_{R\to\infty} \int_{\gamma_R} \frac{1}{z^2+1} dz$$

where $\gamma_R(t) = t, -R < t < R$. We can extend this path by a semicircle α_R making it a closed path. Since the integral $\lim_{R\to\infty} \int_{\alpha_R} \frac{1}{z^2+1} dz \to 0$. since the integral has only one pole in the upper part, we only need to calculate the Residuum for this pole. The residuum

of a function $f(z) = \frac{g(z)}{h(z)}$ at the simple pole z_0 is given by the formula $Res_{z_0} = \frac{g(z_0)}{h'(z_0)}$. Using this, our integral becomes:

$$I = \int_{-\infty}^{\infty} \frac{1}{x^2+1} dx = \lim_{R\to\infty} \oint_{\gamma_R+\alpha_R} \frac{1}{z^2+1} dz = 2\pi i Res_i \frac{1}{1+z^2}.$$

Note If in the semicircle we have more than one pole we simply add the residuums of the poles in question.

C.3. Lorentzian peak

If we were to represent a lorentzian peak $\frac{1}{\pi} \cdot Im(\frac{1}{x-i\eta})$, we have to choose our discretization parameter carefully. In the next figure we show 3 choices, all relative to η: η, $\eta/3$ and $\eta/4$. Also, the start point of the evaluation is important. If we would shift the grid by a factor of $2dx$, $4dx$ and dx respectively we would get a better or worse shape of the peak.

We conclude that for the numerical representation it suffices that $dx = \eta/4$ and the normalization factor for the integral of the Lorentzian peak is $\frac{1}{\pi}$.

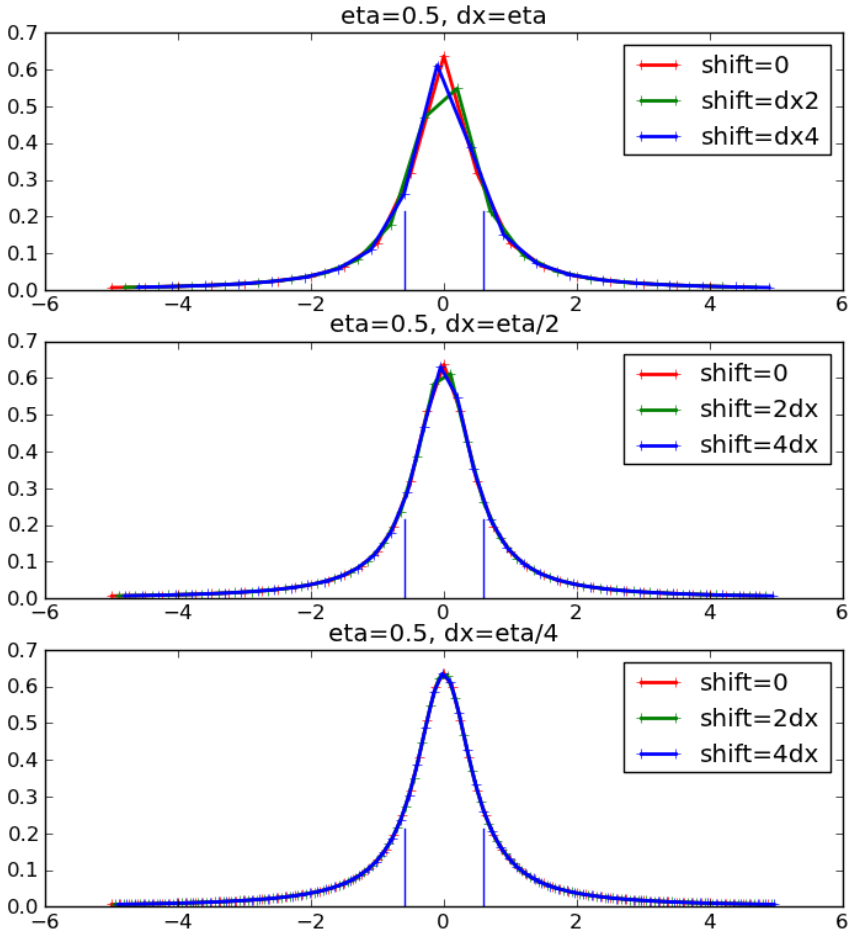


Figure C.1.: Influence of the discretization size and position of the points on the Lorentzian peak.

Bibliography

[1] D. Betts, S. Masui, N. Vats and G. Stewart, Canadian Journal of Physics **74**, 54 (1996).

[2] D. Betts, H. Lin and J. Flynn, Canadian Journal of Physics **77**, 353 (1999).

[3] R. Ierusalimschy, *Programming In Lua*, 2 ed. (PUC-Rio, 2006).

[4] R. Ierusalimschy, L. H. De Figueiredo and W. Celes, *Lua Reference Manual* (, 2006).

[5] E. Pavarini, E. Koch, D. Vollhardt and A. Lichtenstein, *LDA+DMFT Approach to strongly correlated materials* (Forschungszentrum Julich, 2011).

[6] C. C. Paige, *The Computation of Eigenvalues and Eigenvectors of Very Large Sparse Ma- trices*, PhD thesis, London University, 1971.

[7] C. Lanczos, Journal of Research of the National Bureau of Standards **45**, 255 (1950).

[8] H. Kerson, *Statistical Mechanics* (John Wiley, 1987).

[9] M. Caffarel and W. Karuth, Physical Review Letters **72**, 1545 (1994).

[10] Q. Si, M. J. Rozenberg, G. Kotliar and A. E. Ruckenstein, Physical Review Letters **72**, 2761 (1994).

[11] H. W. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, *Numerical Recipies The Art of Scientific Computing*, 3 ed. (Cambridge University Press, 2007).

Acknowledgements

It is a pleasure to thank those people who made this thesis possible. First I would like to thank my supervisor, Prof. Dr. Erik Koch, for his patience during my learning process. I know I can sometimes be a handful.

I would like to also thank Dr. German Ulm and Dr. Fabio Baruffa for proofreading my thesis and the entire group for the help and support they showed me throughout the entire project. I would especially like to mention Dr. Andreas Dolfen who opened my eyes to this field and helped me during the first months.

Many thanks go also to the DAAD, which funded my stay in Germany, without them achieving a Master degree here would have been more difficult.

Last but not least, I would also like to thank my friends and family which have given me the strength to see the thesis through.