User Manual

LSQuantumED

Code for Large Scale Exact Diagonalization for Frustrated Quantum Magnets (and more)

Author:

Alejandra María Foggia

Supervisors:

Ivan Girotto,

Marcello Dalmonte

Contents

Chapter 1

Introduction

This code is the result of a master thesis under the Master in High Performance Computer jointly held by Scuola Internazionale di Studi Superiore e Avanzati (SISSA) and The Abdus Salam Centre for Theoretical Physics (ICTP).

The code is thought to be used to model systems of spin 1/2 on 1-dimension (1D) and 2-dimension (2D) lattices with nearest and next-nearest neighbours interactions, with the possibility of considering anisotropy and disorder along the z-direction. However, the code can be extensible in many ways in order to have different lattices geometries, including three dimensions, and different kind of interactions between spins, which may include many-spins interactions and diagonal and off-diagonal ones.

Initially, the code was thought to be used to compute the dynamical structure factor of the system, a physical quantity that is related, actually proportional, to the intensity of the scattered neutrons of an inelastic neutron scattering experiment. This is way it is already implemented in the code, as well as some other physical quantities, as two-spin correlation functions and the magnetization of the system. As well as before, the code is conceived to be extended also in this aspect, one should be able to add the computation of any other quantity is desired.

This manual is structured in the following way: a general overview of the components of the package, its dependencies, how to install it and a small example on how to use it. After this introduction, comes a detailed explanation of each of its fundamental parts (classes) and how the user can extend them for its personal, particular use. Finally, some comments on miscellaneous functions of the code: how to time the code, parse input command line arguments, etc.

Chapter 2

Getting started

As previously mentioned, this library is primarily thought to (partially) solve the energy spectrum of the Hamiltonian of a spin-1/2 system with some particular nearest and next-nearest neighbours interactions. It has been tested for system sizes up to 38 spins but, in principle, it could be used for any system size up to 64 spins. Because of this, the code requires the use of distributed memory and the Message Passing Interface (MPI) in order to reach such high system sizes, as the memory requirement is such that it does not fit in one machine alone. In view of this, the code is built on top of two very powerful libraries: PETSc and SLEPc.

PETSc is the *Portable, Extensive Toolkit for Scientific Computation* developed at Argonne National Laboratory, for the numerical solution of partial differential equations and related problems. SLEPc is another package built on top of the previous one, it stands for *Scalable Library for Eigenvalue Problem Computations*, and it is developed at Universitat Politecnica de Valencia. These two libraries provided the basis for the development of this code, as they handle the parallel data distribution and the algorithms to diagonalize the matrix, both things done in a highly optimized way, allowing us to concentrate on the particularities of the physics we are dealing with.

2.1 Prerequisites

In order to use this code one needs to have some libraries already installed in the machine:

1. **PETSc** This library has many configuration options. We need the REAL, DOUBLE and with 64-BIT INDICES version of the library. These are fundamental features, the rest of them can be modified by the user. We recommend a NO-DEBUGGING build, for better performance. This package requires a *Blas* and *Lapack* implementation, like, for example, *MKL* or *OpenBlas*. It also requires a *MPI* implementation, like *MPICH* or *OpenMPI*.

2. **SLEPc** As this is built on top of PETSc, its installation is pretty straightforward and it can even be downloaded and installed with PETSc.

3. **Meson** is a build system tool, like CMake for example. It was decided to be used instead of CMake, basically, because it offers a more user friendly syntax. Its installation is also extremely easy. For this tool to work, it requires Python2 or Python3 and the Ninja built system tool also.

4. **Boost** is a group of libraries that can be used in addition with the C++ Standard Library. In the code not every library is used, just the bitwise operations and some constants are used.

5. **Doxygen** is a tool to generate documentation from annotated source code. Its installation and use is simple as well. This dependency is not mandatory, in the sense that everything will work correctly without it, the only thing is that documentation will not be created.

6. Git is a distributed version control system that allows one to handle many versions and revisions of a code keeping track of the history of changes.

For more information on how to install these packages, please refer to their web pages, where one can find very detail and well-explained instructions.

2.1.1 Building blocks of the code

The first two libraries mentioned before provide in their manuals diagrams of all the objects available. In here, for the sake of visually understanding which and how those objects are used, the diagram present in SLEPc's manual is shown in Figure 2.1. PETSc's VECTORS and MATRICES are essential objects, and are the two building blocks that are directly used in the code, the others may be used by SLEPc internally. Through these two, parallel data is managed. From SLEPc, the LINEAR EIGENSOLVERS are the objects directly used. If one should make a diagram of the objects in this package it would be like the shown in Figure 2.2. Later, in 3, a more detailed explanation of the objects will be given.

2.2 Installation

This package can be easily installed. Apart from having the previous libraries already on the machine, there are other important things to set:

- The environment variable `PKG_CONFIG_PATH` has to have the PETSc and SLEPc paths to their pkg-config files. The ".pc" files are usually under the paths `$PETSC_ARCH/lib/pkgconfig` and `$PETSC_ARCH/lib/pkgconfig`.

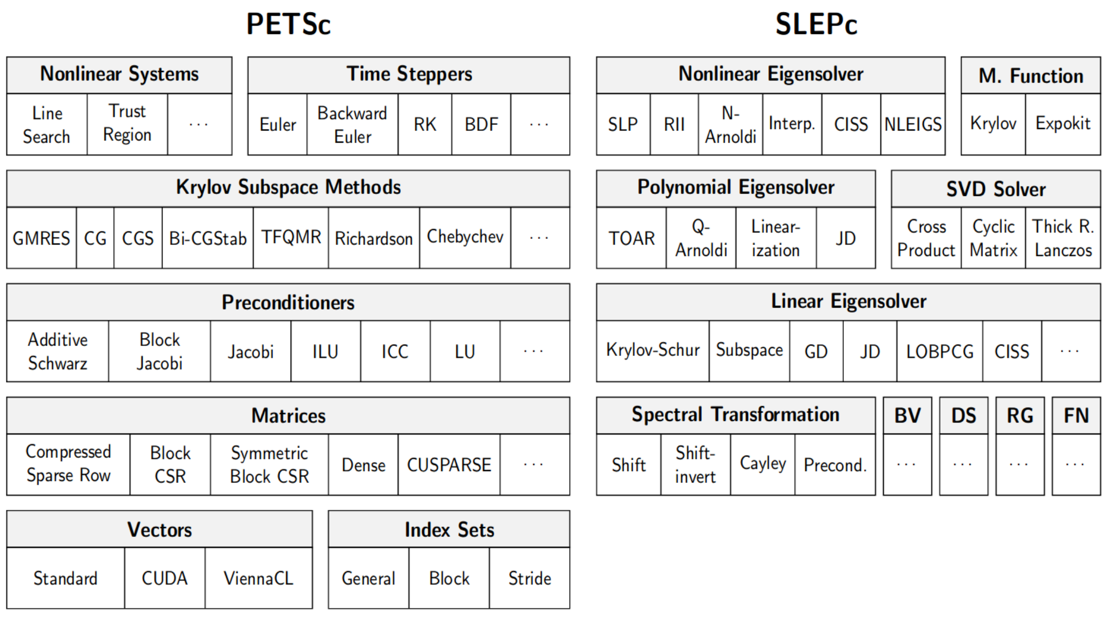

Figure 2.1: PETSc and SLEPc diagrams

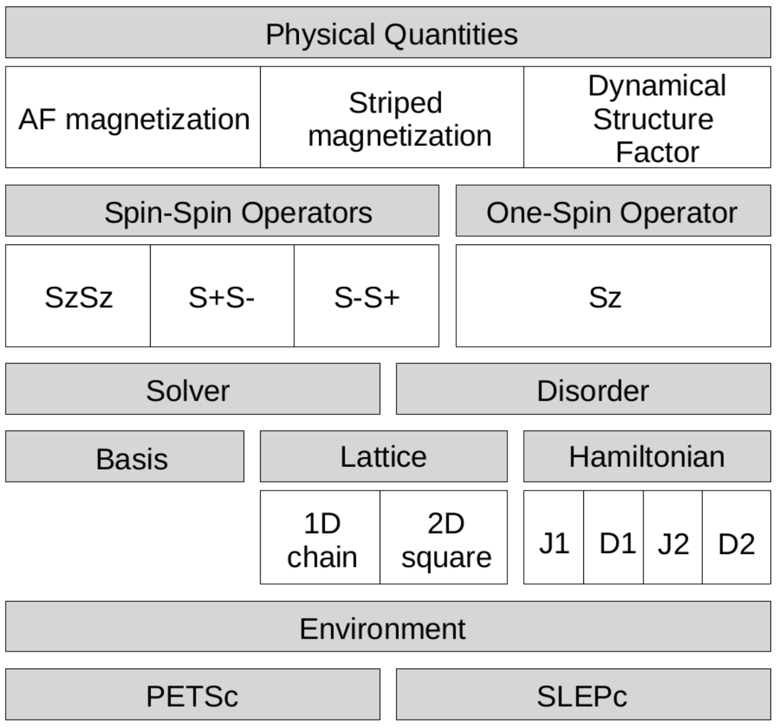

Figure 2.2: Diagram of the objects in the code

- The environment variable `LD_LIBRARY_PATH` has to have the PETSc and SLEPc paths to where the libraries are installed. These are usually under the paths `$PETSC_ARCH/lib` and `$PETSC_ARCH/lib`.
- The environment variable `PATH` has to have the path to the executable `meson.py`.

Once one has set up the environment properly, the steps to follow are:

1. Clone the repository from Github:

   ```
   $ git clone https://github.com/amfoggia/LSQuantumED.git
   ```

2. Configure the installation. Many things can be tuned with Meson, here just some of them, the rest can be seen with `$meson setup --help`.

   ```
   $ meson setup --buildtype=release
                 --prefix=<installation/path>
                 --warnlevel=3
                 --optimization=2
                 -DTIME_CODE=true
                 -Dboostdir=<path/to/boost/include/files>
                 <path/to/build/dir>
   ```

   The options with `-D` are the ones specific to the code and can be checked (with the rest of the configuration opcions), once the build is done, with `$meson configure`.

3. Once the configuration is done one can still reconfigure before installation:

   ```
   $ cd <path/to/build/dir>
   $ meson configure     ------> To check configuration options
   $ meson configure -DTIME_CODE=false
   ```

4. Now, to install the package just type:

   ```
   $ meson install
   ```

5. If one configures the installation with `-DBUILD_TEST=true` then it is possible to test the installation and the correct functioning of the code.

6. The installation generates a pkgconfig file for the code. It is found under the path `$prefix/lib/pkgconfig`, and one has to make sure that it is included in the `PKG_CONFIG_PATH` environment variable. The same way, one has to include the path for the library `$prefix/lib` on the environment variable `LD_LIBRARY_PATH`.

2.3 Running a code

Inside the installation folder there are a few examples one can run as trial. These are just very simple main functions that demonstrate how to construct a Hamiltonian and to diagonalize it, also how to compute the dynamical structure factor and how to add disorder to the Hamiltonian.

In order to run them one just has to do:

```
$ cd /path/to/installation/dir
$ mpiexec -np 1 ./examples/main.x
```

However, if one wants to write a main function on its own and then compile it in order to obtain an executable, the steps are:

```
$ mpicxx -c my_main.cpp `pkg-config --cflags LSQuantumED` -o my_main.o
$ mpicxx my_main.o `pkg-config --libs LSQuantumED`-o my_main.x
```

2.4 Command line options

In order to run a main function, these are the options one has to/can give to the executable:

- `-n` <6> for the number of spins in the system
- `-ltype <chain1D>` for the type of lattice, choose one of `chain1D square2D`
- `-j1` <1.0> for the nearest neighbours interaction
- `-d1` <1.0> for the nearest neighbours anisotropy
- `-j2` <0.0> for the next-nearest neighbours interaction
- `-d2` <0.0> for the next-nearest neighbours anisotropy
- `-disorder <false>` to allow running with disorder or not. In case one sets it to `true`, the following options are also allowed:
  - `-min_dis` <-0.3> for the minimum value for the random disorder variable
  - `-max_dis` <0.3> for the maximum value for the random disorder variable
  - `-rep` <1> for the number of disorder repetitions
  - `-reprod <true>` to allow or not the reproducibility of the random numbers

Apart from these ones, one can also use every command line option from PETSc and SLEPc libraries. In particular, to visualize them, as well as the ones specific to this code, one can do use `-help`.

Chapter 3

Implementation

This chapter explains the implementation of the code, in a detailed-but-not-so-detailed way, in order to allow any other user to add new lattices or new Hamiltonians.

The code is written in C++ language, in an object oriented fashion. It is composed by building blocks that mirror the organization of the underlying mathematical description of the physical problem: the basis of the system, the lattice (distribution of spins), the Hamiltonian, the solver (diagonalization of the matrix) and spin-spin operators S^zS^z. Each of these blocks is represented by a C++ class and they are all related, in the sense that some are needed for the construction of others. The work flow of a simple `main` function is shown in Figure **??**.

3.1 Environment

The `Environment` is an object that takes care of the homogenization of some variables among all the parts of the code. It also takes care of correctly initializing and finalizing the SLEPc environment, which handles the initialization and finalization of the PETSc environment, which, in turn, performs the MPI calls to set the parallel environment. One other parameter that is shared among every object is the number of spins in the system, and it is held by the `Environment` object. This number has to be positive and even.

There are two possible ways of instantiating an object of this type: one can pass the number of spins in the system, which means that it will be hard-coded on the `main` function; or, one can initialize it without this quantity and later give a value to the variable that holds the system size. This way, the user is able to change the number of spins at runtime by passing it as an argument of the executable.

Apart from this variable, one has to give the value of `argc` and `argv` (the same arguments of the `main` function). The signatures of the constructors are given in the Listing 3.1. The `help` argument is a sentence that contains information about the program one is running, and is displayed when using the `-help` command line option.

Listing 3.1: Environment constructors

```
Environment(int argc,
        char ** argv,
        char help[] = nullptr);

Environment(int argc,
        char ** argv,
        PetscInt m_nspins,
        char help[] = nullptr);
```

It is important to highlight that if one uses the first constructor, but does not initialize the `m_nspins` variable, the default value is 0. This will raise an error at runtime. However, if one later uses the `parse_args()` macro, the default value for `m_nspins` is 6.

Another thing that happens when instantiating an object of this type, if the `TIME_CODE` configuration option is true, is the creation of a `TimeMonitor` object. This is the responsible for collecting information on the execution time for every function.

The `Environment` object is the first thing to be instantiated when writing a `main` function, and is an input argument for every other object and function in the code.

3.2 Lattice

The ground state of the system and the correlation functions depend on the relative position of the spins. The `Lattice` class reflects this: it creates a list of the neighbours of each spin depending on the configuration. This list takes into account the *periodic boundary conditions* used in the problem treated here. One can choose between a `1D chain` lattice or a `2D square` lattice. This list is a `std::vector<PetscInt>` and it is replicated in the memory of every MPI process.

There is a base class, called `Lattice`, that displays the general structure of any kind of lattice one can have. Every other class is *derived* from this one, and specific member functions' implementations are given in the derived classes. The base class `Lattice` is shown in Listing 3.2. It has one member `nspins` and two important member functions: `get_nn()` and `get_nnn()`. These are the ones that construct the nearest neighbours and next-nearest neighbours vectors. No implementation is given in this base class, and it is marked (by the "`= 0`") such that every derived class has to implement them on their own.

Listing 3.2: Lattice base class.

```
class Lattice {
protected:
  PetscInt nspins;
  std::vector<PetscInt> nn;
  std::vector<PetscInt> nnn;
  std::vector<PetscInt> nnXsite;
  std::vector<PetscInt> nnnXsite;
public:
  Lattice() = default;
  Lattice(Environment& env)
    : nspins{env.nspins} {}
  virtual lattice_type get_type() const = 0;
  virtual PetscInt num_nn() = 0;
  virtual PetscInt num_nnn() = 0;
  virtual PetscInt num_nnXsite(PetscInt lat_site) = 0;
  virtual PetscInt num_nnnXsite(PetscInt lat_site) =
   0;
  virtual PetscInt num_spins() {return nspins;};
  virtual ~Lattice() {}
};
```

There are a few other functions that retrieve information on the lattice, like the type, the number of neighbours and the number of spins.

3.2.1 How are neighbours' list constructed?

Taking a 1D chain as an example, it will be shown how the neighbours' list is constructed. In Figure 3.1 an 8-spins chain with periodic boundary conditions is visible, spins 0 and 7 are duplicated in order to show this. Each of the spins has two nearest neighbours and two next-nearest neighbours. Take, for example, spins 1 and 2; its neighbours are {0,2} and {1,3}, respectively. However, there is no need to take both neighbours into account, because one can take only one and then count the interaction twice, as H_{12} and H_{21} is the same. This is the reason why, in this code, only one of the neighbours, the one on the right, is computed. For the example in the Figure 3.1, the vectors looks like:

```
nn = {0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,0}
nnn = {0,2,1,3,2,4,3,5,4,6,5,7,6,0,7,1}
```

For each spin, there are two elements, the first one contains the spin being considered, the second one, its neighbour. So, the functions `get_nn()` and `get_nnn()` construct these lists.

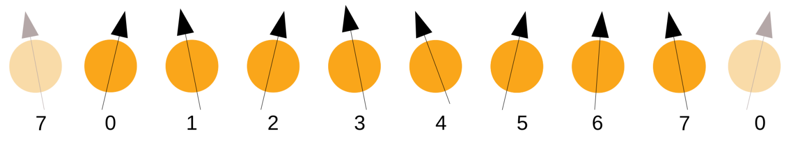

Figure 3.1: A one-dimensional chain lattice of 8 spins.

Now, the example of the 2D square lattice, a 16-spins system, shown in Figure 3.2. In this case, the number of neighbours is four, but only half of them are considered. One important thing to take into account is that the numbering of the spins grows first on the y direction and then on the x direction, and this "rule" is valid also for the order in which the neighbours are considered. The lists of neighbours are

```
nn = {0,1,4,                    nnn = {0,5,7,
      1,2,5,                           1,6,4,
      2,3,6,                           2,7,5,
      3,0,7,                           3,4,6,
      4,5,8,                           4,9,11,
      5,6,9,                           5,10,8,
      6,7,10,                          6,11,9,
      7,4,11,                          7,8,10,
      8,9,12,                          8,13,15,
      9,10,13,                         9,14,12,
      10,11,14,                        10,15,13,
      11,8,15,                         11,12,14,
      12,13,0,                         12,1,3,
      13,14,1,                         13,2,0,
      14,15,2,                         14,3,1,
      15,12,3}                         15,0,2}
```

Given that one wants to compute next-nearest neighbours, lattices with two spins in one direction do not make sense; and if one considers the striped ground state of a Hamiltonian, lattices with odd number of spins in one direction do not satisfy periodic boundary conditions, so they should not be allowed either. In summary, the number of spins in any direction should be even and larger than three.

As an example on how to derive a particular class from the `Lattice` base class, the case of `chain1D` is presented on Listing 3.3. There is one main difference with the `Lattice` base class: in here, there are two more members, `nn` and `nnn`, the neighbours' vectors. In Listing 3.3 it is also shown the constructor, where one can see how the function `get_nn()` and `get_nnn()` are used to populate the `nn` and `nnn` vectors.

Listing 3.3: chain1D derived class.

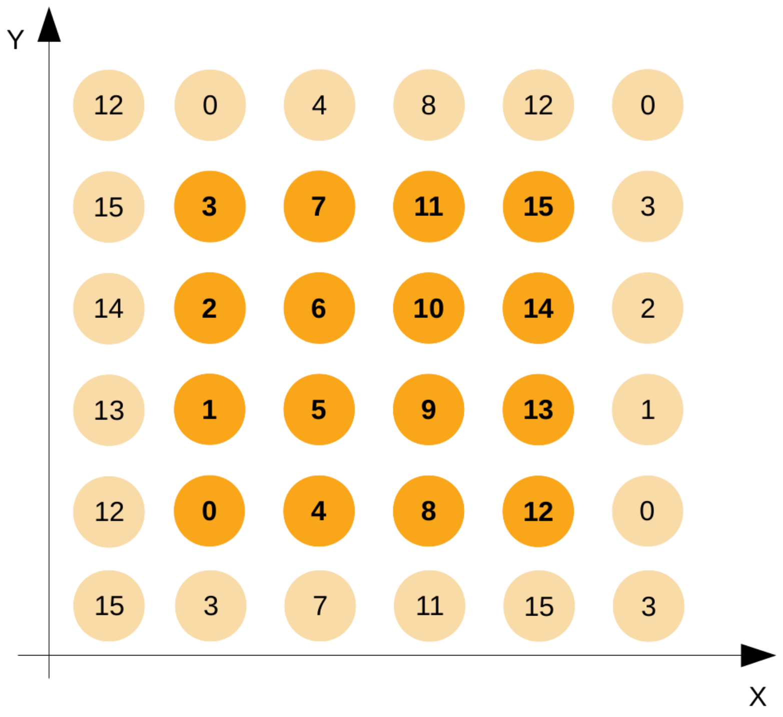

Figure 3.2: A two-dimensional square lattice of 16 spins.

```
class chain1D : public Lattice {
private:
  lattice_type type;
  std::vector<PetscInt>& get_nn();
  std::vector<PetscInt>& get_nnn();
public:
  std::vector<PetscInt> nn;
  std::vector<PetscInt> nnn;
  std::vector<PetscInt> nnXsite;
  std::vector<PetscInt> nnnXsite;
  chain1D(Environment& env);
  lattice_type get_type() const {return type;}
  PetscInt num_nn() {return 1;}
  PetscInt num_nnn() {return 1;}
  PetscInt num_nnXsite(PetscInt lat_site) {return
   nnXsite[lat_site];}
  PetscInt num_nnnXsite(PetscInt lat_site) {return
   nnnXsite[lat_site];}
// --------------------------------------------
// Constructor
chain1D(Environment& env)
  : Lattice{env},
    type{lattice_type::chain1D}
{ nn = get_nn();
```

```
    nn.shrink_to_fit();
    nnn = get_nnn();
    nnn.shrink_to_fit();
}
```

3.3 Basis

Another building block of the mathematical description of the problem is the basis of the system, represented by the `Basis` class. The purpose of this class is to

- compute the number of spins up and spins down given the total number of spins (`nspins`) and the total magnetization (`total_mag`), which is conserved;
- construct and hold the vector with the basis elements of the system `std::vector<PetscInt>`;
- construct and hold the vector with the basis elements in bit representation `std::vector<boost::dynamic_bitset<>>`, that is used during developing and testing stages only.

As total magnetization conservation is imposed, the number of elements in the basis vector is given by

$$\mathsf{N}_{\mathcal{H}_{\mathsf{red}}} = \frac{\texttt{nspins!}}{N_\uparrow! N_\downarrow!}, \tag{3.1}$$

where $N_\uparrow$ and $N_\downarrow$ are the number of spins pointing up and down, correspondingly. It is important to keep in mind that all these elements are going to be constructed and stored in the machine's memory.

3.3.1 Basis' machine representation

How can the system basis be represented? In Quantum Mechanics, once the quantization direction is established, in this case z-direction, the state of a 1/2 spin can be written as a combination of two basis states, $|\uparrow\rangle$ and $|\downarrow\rangle$ or, what is the same, $|+\frac{1}{2}\rangle$ and $|-\frac{1}{2}\rangle$. This 1/2-spin basis states satisfy the following, when applied to spin operators:

$$\begin{aligned} S^z|\uparrow\rangle &= 1/2\ \hbar\ |\uparrow\rangle, & S^z|\downarrow\rangle &= -1/2\ \hbar\ |\downarrow\rangle, \\ S^+|\uparrow\rangle &= 0, & S^+|\downarrow\rangle &= \hbar\ |\uparrow\rangle, \\ S^-|\uparrow\rangle &= \hbar\ |\downarrow\rangle, & S^-|\downarrow\rangle &= 0. \end{aligned} \tag{3.2}$$

In this code $\hbar$ is taken to be 1. In order to understand how the matrix gets constructed, it is important to remark that these are elements of a basis, therefore they satisfy $\langle a|b\rangle = 0$ and $\langle a|a\rangle = 1$.

When more than one spin is involved, it is possible to represent their state as a direct product of the state of each spin. This means that if the individual state of each of the N spins is represented as $|0\rangle$, $|1\rangle$, ..., $|N-1\rangle$, the system state is given by

$$|\Psi\rangle = |N-1\rangle \otimes \cdots \otimes |1\rangle \otimes |0\rangle = |(N-1)\ \ldots\ 1\ 0\rangle^1. \tag{3.3}$$

In order to be able to use these states to compute anything, the "up" state, $|\uparrow\rangle$, is represented by as a 1 and the "down" state $|\downarrow\rangle$ as a 0. Let us make an example to render things clearer. Think of a system of 6 spins in which half of them are pointing up, meaning they are in the state $|\uparrow\rangle$, and the other half are pointing down, starting with an "up" spin. Let's assume also that the spins in the lattice are numbered and contiguous spins point in opposite directions. The lattice in itself is not something that plays a role in the basis definition, it is only important to know that spins are numbered, in some way, from 0 to $\texttt{nspins}-1$. After all the setting, the system state is given by $|\,010101\rangle$. By using 1's and 0's for the states they can be treated as bits, and the state of the whole system can be seen as the bit representation of a positive integer number. In this example, the state of the system is represented by the number 21.

In order to create the basis, one has to know how many spins "up" and "down" there are, and for that two quantities are needed: total number of spins in the system and the total magnetization M_T. Then it is possible to obtain the number of spins "up", $N_\uparrow$, as

$$N_\uparrow = \frac{2M_T + \texttt{nspins}}{2}; \tag{3.4}$$

the number of spins "down" is just $N_\downarrow = \texttt{nspins} - N_\uparrow$. Once this is done, the basis elements are the integer numbers whose bit representation are all the possible combinations of $N_\uparrow$ 1's and $N_\downarrow$ 0's. For example, considering 6 spins with $M_T = 1$ the basis has 15 elements (see Table 3.1).

There exists an algorithm, called Gosper's hack [1], that uses bitwise operations and gives the integer numbers in an ascending order, and it is depicted in Algorithm 3.4.

Listing 3.4: Gosper's Hack.

```
input: int x (one element of the basis)
output: int y (next element of the basis)

unsigned long u = x & -x;
```

[1]The spins in the system state are displayed in descending order, because it is related to how the computer indexes the bits of a number when using its bit representation.

```
unsigned long v = u + x;
x = v +(((v^x)/u)>>2);
```

index	bits	integer
0	001111	15
1	010111	23
2	011011	27
3	011101	29
4	011110	30
5	100111	39
6	101011	43
7	101101	45
8	101110	46
9	110011	51
10	110101	53
11	110110	54
12	111001	57
13	111010	58
14	111100	60

Table 3.1: Basis elements, bit and integer representation, for a 6 spins system with $M_T = 1$.

Now, let us take a look at the `class Basis`. In Listing 3.5 there is the declaration of the class.

Listing 3.5: Basis class.

```
class Basis {
private:
  PetscInt max_mag;
  PetscInt compute_size();
  std::vector<PetscInt>& generate_int_basis();
  std::vector<boost::dynamic_bitset<>>&
   generate_bit_basis();

public:
  PetscInt nspins;
  PetscInt total_mag;
  PetscInt nspins_up;
  PetscInt size;
  PetscMPIInt mpi_size;
  PetscMPIInt mpi_rank;
  PetscInt local_size;
```

```
    PetscInt global_start_index;
    std::vector<PetscInt> int_basis;
    std::vector<boost::dynamic_bitset<>> bit_basis;

    // Constructors
    Basis();
    Basis(Environment& env,PetscInt total_mag);
    Basis(const Basis& rhs);
    Basis(Basis&& rhs) noexcept;

    // Assignment operators
    Basis& operator=(const Basis& rhs);
    Basis& operator=(Basis&& rhs) noexcept;
  };
```

There are two `std::vector<PetscInt>`, `int_basis` and `bit_basis`, that contain the elements of the basis in both representations; there are also two (private) functions that construct those vectors, which are called inside the constructor.

3.4 Hamiltonian

Next, one has to build the Hamiltonian of the system. The Hamiltonian that can be used in the code right now is the one shown in Equation 3.5. We can see three lines in this equation, the first one corresponding to nearest neighbours interactions, the second one to next-nearest neighbours interactions, and the third one to the disorder part. The parameters J_1, Δ_1, J_2 and Δ_2 can be modified at runtime, and it also can be specified if disorder is desired or not. We will talk about the `parse_args()` function in the next chapter.

$$\begin{aligned}\mathcal{H} = {} & J_1 \sum_{\langle i,j \rangle} \{\frac{1}{2}(S_i^+ S_j^- + S_i^- S_j^+) + \Delta_1 S_i^z S_j^z\} \\ & + J_2 \sum_{\langle\langle i,j \rangle\rangle} \{\frac{1}{2}(S_i^+ S_j^- + S_i^- S_j^+) + \Delta_2 S_i^z S_j^z\} \\ & + \sum_{i=0}^{N-1} h_i S_i^z\end{aligned} \tag{3.5}$$

This mathematical (and physical) object is conceived as a class also, the `Hamiltonian` class. It requires a `Lattice` and a `Basis` objects to be constructed, and its main purpose is to build the matrix that represents the Hamiltonian of Equation 3.5. The matrix is stored in a `Mat` PETSc's object, specifically it is a

MATMPIAIJ kind of matrix, which means a parallel sparse matrix. As the matrix is sparse, the total number of elements is not $\mathsf{N}_{\mathcal{H}_{\mathsf{red}}} \times \mathsf{N}_{\mathcal{H}_{\mathsf{red}}}$ (see Equation 3.1), still how sparse the matrix is depends on the number of spins in the system and the lattice configuration. In Listing 3.6 we can see the Hamiltonian class.

When instantiating a `Hamiltonian` object, the `MatInit()` function is called, which calls some PETSc functions that set the matrix type and size, and the `prealloc()` function that reserves the necessary memory for the matrix (for more detail see Section 3.4.1). We will discuss this last function in detail shortly. Later, the user has to call the "building functions" that wants to use, according to the kind of Hamiltonian it wants to create, meaning the kind of interactions. In the next section, we will explain how the Hamiltonian elements are computed, in order to allow users to create their own Hamiltonian.

Listing 3.6: Hamiltonian class.

```
template<typename L>
class Hamiltonian {

private:
  Basis basis;
  L& lattice;
  PetscInt size;
  PetscReal J1;
  PetscReal Delta1;
  PetscReal J2;
  PetscReal Delta2;
  PetscErrorCode MatInit(Environment& env);
  PetscErrorCode MatClean();

public:
  PetscMPIInt mpi_size;
  PetscMPIInt mpi_rank;
  Mat hamilt; // <-- Object that has the matrix

  // Constructor
  Hamiltonian(Environment& env,
        Basis& m_basis,
        L& m_lattice,
        PetscReal m_J1,
        PetscReal m_Delta1,
        PetscReal m_J2,
        PetscReal m_Delta2);
  ~Hamiltonian() { MatClean(); }
```

```
    // Building functions
    PetscErrorCode build_disorder(Environment& env,
                                  PetscReal* hi);
    PetscErrorCode build_diag(Environment& env);
    PetscErrorCode build_off_diag(Environment& env);
    PetscErrorCode build_hamilt(Environment& env);

    // Retrieving-parameters functions
    PetscInt hamilt_size() const { return size; }
    PetscInt hamilt_dim() const { return size*size; }
    PetscReal hamilt_J1() const { return J1; }
    PetscReal hamilt_J2() const { return J2; }
    PetscReal hamilt_D1() const { return Delta1; }
    PetscReal hamilt_D2() const { return Delta2; }
    lattice_type hamilt_type() const { return lattice.
     get_type(); }

    // Preallocation
    PetscErrorCode prealloc(Environment& env);
};
```

3.4.1 Hamiltonian elements

The Hamiltonian in Equation 3.5 has the symbol $\sum_{\langle i,j\rangle}$ (and $\sum_{\langle\langle i,j\rangle\rangle}$), this means that one has to sum the quantity that follows over all the pairs of spins i,j that are nearest (and next-nearest) neighbours. Now it is clear that the lattice becomes an important element as one needs to know who is neighbour of whom. Each of the elements of the matrix corresponds to the operation

$$\mathcal{H}_{(m,n)} = \langle\Psi_m|\mathcal{H}|\Psi_n\rangle = \Psi_m^\dagger * \mathcal{H} * \Psi_n, \tag{3.6}$$

that is equivalent to the multiplication of the matrix by the basis element vector on both sides. The said Hamiltonian has two distinct parts: a diagonal and an off-diagonal part. Without paying attention to the constants, the parts are given by $S_i^z S_j^z$ and $S_i^+ S_j^- + S_i^- S_j^+$, respectively.

Diagonal part

Suppose a system of 6 spins with total magnetization $M_T = 1$, the basis elements are shown in Table 3.1. Take element 8, $101110 \equiv 46$, and a `1D chain` lattice. The nearest neighbours diagonal part is computed as:

$$\begin{aligned}\mathcal{H}_{diag}|101110\rangle = S_0^z S_1^z|101110\rangle + S_1^z S_2^z|101110\rangle + S_2^z S_3^z|101110\rangle + \\ + S_3^z S_4^z|101110\rangle + S_4^z S_5^z|101110\rangle + S_5^z S_0^z|101110\rangle.\end{aligned} \tag{3.7}$$

Each term gives the following result:

$$\begin{aligned}
S_0^z S_1^z|101110\rangle &= (-1/2) * 1/2\ |101110\rangle, & S_1^z S_2^z|101110\rangle &= 1/2 * 1/2\ |101110\rangle,\\
S_2^z S_3^z|101110\rangle &= 1/2 * 1/2\ |101110\rangle, & S_3^z S_4^z|101110\rangle &= 1/2 * (-1/2)\ |101110\rangle,\\
S_4^z S_5^z|101110\rangle &= (-1/2) * 1/2\ |101110\rangle, & S_5^z S_0^z|101110\rangle &= 1/2 * (-1/2)\ |101110\rangle,
\end{aligned}$$

therefore, Equation 3.7 gives for the element $(8, 8)$ of the Hamiltonian matrix

$$\mathcal{H}_{diag}|101110\rangle = -\frac{1}{2}\ |101110\rangle = \mathcal{H}_{(8,8)}. \tag{3.8}$$

As the spin operator S_i^z does not change the basis element that is applied to it, the diagonal part only connects basis elements with themselves. To sum up, when two neighbour spins are in the same state, one has to sum 1 to the diagonal value of the basis element it is being considered, and when they are in opposite states one has to subtract one to the diagonal value. At the end, the result has to be multiplied by the constant $J\Delta/4$.

Generally speaking, to construct each part of the Hamiltonian there are two functions: one that computes the elements, like `get_coup_elems()`, `get_elem_diag()` and `get_disorder_elem()`, and one that sets the elements on the matrix, like `build_diag()`, `build_off_diag()` and `build_disorder()`, respectively.

The procedure to compute and construct the diagonal elements is shown in Listing 3.7. First, there is the function that computes the elements. We can see that it is very simple: in `bitA` and `bitB` we collect the bits corresponding to the neighbouring spins (`i` and `spin`), and then we compare them with a bitwise XOR (`^`) operation. If they are different, which corresponds to something like

$$\langle\uparrow\downarrow|S_1^z S_2^z|\uparrow\downarrow\rangle = \hbar^2 \times \frac{1}{2} \times -\frac{1}{2} = -1 \times \frac{1}{4}$$

if we consider $\hbar = 1$, the diagonal element is -1; else, it is $+1$.

Listing 3.7: Diagonal elements.

```
void HamiltHelper::get_elem_diag(PetscInt basis_elem,
                                 PetscInt i,
                                 PetscInt spin,
                                 PetscInt& diag_elem) {
  PetscInt bitA, bitB;

  // Get bits to compare
  bitA = (basis_elem >> i) & 0x1;
  bitB = (basis_elem >> spin) & 0x1;

  if (bitA ^ bitB)
    diag_elem -= 1; // If bits are different
  else
    diag_elem += 1; // If bits are equal
}

// ---------------------------------------------------------

template<typename L>
PetscErrorCode Hamiltonian<L>::build_diag(Environment&
     env) {

  PetscErrorCode ierr = 0;
  PetscInt Istart, Iend, tmp_elem, local_index;
  PetscReal elem;

  ierr = MatGetOwnershipRange(hamilt, &Istart, &Iend);
    CHKERRQ(ierr);

  // Do not construct diagonal
  if (std::fabs(Delta1) < 1e-14 && std::fabs(Delta2) <
    1e-14) {
    return(ierr);
  }
  // Construct nearest neighbours diagonal
  if (std::fabs(Delta1) >= 1e-14) {
    for (PetscInt i = Istart; i < Iend; ++i) {
      local_index = i - basis.global_start_index;
      tmp_elem = HamiltHelper::get_elem_diag(
   neigh_type::nn,basis.int_basis[local_index], basis.
   nspins, lattice);

      if (tmp_elem == 0)
          continue;

      elem = Delta1*0.25 * tmp_elem;
      ierr = MatSetValue(hamilt,
             i,
             i,
             elem,
             ADD_VALUES);
      CHKERRQ(ierr);
      } // -- If (Delta1)

  // Construct next-nearest neighbours diagonal
  if (std::fabs(Delta2) >= 1e-14) {
    for (PetscInt i = Istart; i < Iend; ++i) {
```

```
53        local_index = i - basis.global_start_index;
54
55        tmp_elem = HamiltHelper::get_elem_diag(neigh_type::nnn,basis.int_basis[local_index], basis.nspins, lattice);
56
57        if (tmp_elem == 0)
58            continue;
59
60        elem = Delta2*0.25 * tmp_elem;
61        ierr = MatSetValue(hamilt,
62                           i,
63                           i,
64                           elem,
65                           ADD_VALUES);
66        CHKERRQ(ierr);
67    } // -- If (Delta2)
68
69    return ierr;
70}
```

The $\frac{1}{4}$ factor is multiplied in the function that sets the values on the matrix, as is seen in line 41 and 60 of Listing 3.7. Regarding this function, it is important to mention some PETSc functions:

- `MatGetOwnershipRange(hamilt,&Istart,&Iend)` In `Istart` and `Iend` we store the global index of the first and last local rows of the matrix owned by each process. This way, the loop then goes over just these elements, meaning: each process sets its own values, there is no communication in this step.
- `MatSetValue(hamilt,i,i,elem,ADD_VALUES)` With this function, one sets the value `elem` in row `i` and column `i` (as this is the diagonal part), and adds it to whatever is in there already.

So, basically in the loop one first computes the local index corresponding to a global one (lines 35 and 53), then computes the value of the corresponding element with the `get_elem_diag()` function (lines 36 and 55), and the one sets the values with `MatSetValue()` (lines 42 and 61).

Off-diagonal part

The off-diagonal part is constructed in the same way, with only a few small changes, but the main idea is to construct the matrix row by row. Regarding the physics, the off-diagonal part has S_i^+ and S_i^- spin operators which change the state of the system when applied to them. Following the same example as before one has

$$\begin{aligned}\mathcal{H}_{off}|101110\rangle = {} & \big(S_0^+S_1^-|101110\rangle + S_0^-S_1^+|101110\rangle\big) + \\ & + \big(S_1^+S_2^-|101110\rangle + S_1^-S_2^+|101110\rangle\big) + \\ & + \big(S_2^+S_3^-|101110\rangle + S_2^-S_3^+|101110\rangle\big) + \\ & + \big(S_3^+S_4^-|101110\rangle + S_3^-S_4^+|101110\rangle\big) + \\ & + \big(S_4^+S_5^-|101110\rangle + S_4^-S_5^+|101110\rangle\big) + \\ & + \big(S_5^+S_0^-|101110\rangle + S_5^-S_0^+|101110\rangle\big)\,,\end{aligned} \tag{3.9}$$

$$\begin{aligned}\mathcal{H}_{off}|101110\rangle = {} & (|101101\rangle + 0) + (0\ + 0) + (0\ + 0) + \\ & + (0\ + |110110\rangle) + (|011110\rangle + 0) + (0\ + |001111\rangle)\,.\end{aligned}$$

The off-diagonal part connects basis element 8 with basis elements $7 \equiv 101101$, $11 \equiv 110110$, $4 \equiv 011110$ and $0 \equiv 001111$. In summary, in order to get the off-diagonal part of a row associated to a basis element, one needs to check if two neighbouring spins are in opposite states, swap them and see which basis element matches the state obtained after the swap.

Putting both parts together one obtains the eighth row of the matrix:

$$\mathcal{H}_{(8,:)} = \left(\frac{J_1}{2}, 0, 0, 0, \frac{J_1}{2}, 0, 0, \frac{J_1}{2}, \frac{J_1\Delta}{4}, 0, 0, \frac{J_1}{2}, 0, 0, 0\right). \tag{3.10}$$

The way the previous operations are carried out in the code is through bitwise operations (see Listing 3.8). As for the diagonal part, one gets the bits (in `bitA` and `bitB`) corresponding to the spins one wants to compare, `i` and `spin`. If the bits are different, one swaps them, searches for the index corresponding to that new basis element and stores it in the `coup_elems` vector (line 25).

The function that sets the element in the matrix is, again, very similar to the one in the diagonal case, with two main differences: we call `get_coup_elems()` to compute the elements of the row; and `MatSetValues()` to set a full row instead of element by element. At the end, one either "flushes" the values or does a "final assembly", depending on the presence of disorder or not.

Listing 3.8: Off-diagonal elements.

```
1 void HamiltHelper::get_coup_elems(Basis& basis,
2                     PetscInt elem,
3                     PetscInt i,
4                     PetscInt spin,
5                     PetscInt& ncol,
6                     PetscInt* coup_elems) {
7
```

```
  PetscInt bitA, bitB;
  PetscInt basis_elem, swapped_basis_elem;
  basis_elem = basis.int_basis[elem];
  std::vector<PetscInt> swapped_elems;

  // Get the bits to compare
  bitA = (basis_elem >> i) & 0x1;
  bitB = (basis_elem >> spin) & 0x1;

  // Compare bits
  if (bitA ^ bitB) {

    // Swap bits
    swapped_basis_elem = basis_elem ^ (1ull << i);
    swapped_basis_elem ^= (1ull << spin);

    // Search for the element
    coup_elems[ncol] = search_index(swapped_basis_elem, basis.nspins);
    ncol += 1;
  }
}
// ---------------------------------------------------
template<typename L>
PetscErrorCode Hamiltonian<L>::build_off_diag(Environment& env) {

  PetscErrorCode ierr = 0;
  PetscInt ncol, Istart, Iend, local_index;
  PetscScalar * Jcoup;
  PetscInt * coup_elems;

  PetscCalloc1(basis.nspins*3, &Jcoup);
  PetscCalloc1(basis.nspins*3, &coup_elems);

  // Construct nearest neighbours off-diagonal
  if (std::fabs(J1) >= 1e-14) {

    for (PetscInt i = 0; i < basis.nspins*3; ++i)
      Jcoup[i] = J1*0.5;

    ierr = MatGetOwnershipRange(hamilt, &Istart, &Iend); CHKERRQ(ierr);
    for (PetscInt elem = Istart; elem < Iend; ++elem) {
      local_index = elem - basis.global_start_index;

    HamiltHelper::get_coup_elems(neigh_type::nn, basis, local_index, lattice, ncol, coup_elems);
    ierr = MatSetValues(hamilt,
          1, /* number of rows */
          &elem, /* row(s) */
          ncol, /* number of columns */
          coup_elems, /* columns */
          Jcoup, /* values to fill them with */
          ADD_VALUES);
    CHKERRQ(ierr);
    }
  } // -- if J1

  // Construct next-nearest neighbours off-diagonal
  if (std::fabs(J2) >= 1e-14) {

    for (PetscInt i = 0; i < basis.nspins*3; ++i)
      Jcoup[i] = J2*0.5;

    ierr = MatGetOwnershipRange(hamilt, &Istart, &Iend); CHKERRQ(ierr);
    for (PetscInt elem = Istart; elem < Iend; ++elem) {
      local_index = elem - basis.global_start_index;

    HamiltHelper::get_coup_elems(neigh_type::nnn, basis, local_index, lattice, ncol, coup_elems);
    ierr = MatSetValues(hamilt,
          1, /* number of rows */
          &elem, /* row(s) */
          ncol, /* number of columns */
          coup_elems, /* columns */
          Jcoup, /* values to fill them with */
          ADD_VALUES);
    CHKERRQ(ierr);
    }
  } // -- if J2

  ierr = PetscFree(Jcoup);
  ierr = PetscFree(coup_elems);

  if (env.disorder_flg == false) {
    ierr = MatAssemblyBegin(hamilt,MAT_FINAL_ASSEMBLY)
```

```
; CHKERRQ(ierr);
    ierr = MatAssemblyEnd(hamilt,MAT_FINAL_ASSEMBLY);
   CHKERRQ(ierr);
  } else {
    ierr = MatAssemblyBegin(hamilt,MAT_FLUSH_ASSEMBLY)
   ; CHKERRQ(ierr);
    ierr = MatAssemblyEnd(hamilt,MAT_FLUSH_ASSEMBLY);
   CHKERRQ(ierr);
  }
  return ierr;
}
```

The search function

If one wants to implement another basis, where the magnetization is not conserved, for example, it is important to understand how the "search functions" in this code work. There are two functions, one that goes from `index` to `element` and vice-versa. In both, we make use of the fact that the number of "ones" is constant.

The "easiest" to understand is the second one, as it has an "expression", given in Equation 3.11. The index I of a basis element is given by

$$I = \sum_{i=1}^{N_\uparrow} \binom{p_i}{i}, \tag{3.11}$$

where p_i is the position of the i-th "up" spin and $\binom{p_i}{i} = 0$ when $p_i < i$. Using again the same setting as before, 6-spins system with total magnetization 1 ($N_\uparrow = 4$), take the elements 4, 7 and 11 as examples; results of the use of the hashing function are shown in Table 3.2.

integer	bits	p_1	p_2	p_3	p_4	index
30	011110	1	2	3	4	4
45	101101	0	2	3	5	7
54	110110	1	2	4	5	11

Table 3.2: An example of the use of the search function for a 6-spins system with total magnetization 1.

The other way round, there is no expression for the function but an algorithm. It is better if we explain it with an example. Suppose we take 6-spins system with total magnetization 1 (as before), say we want to know the element corresponding to index $I^* = 2$ (see Table 3.1) and let us use the notation from Equation 3.11. All the information we have is that there are four "ones" and two "zeros".

- **Step 0:** Assume the fourth “one” is in the highest position possible, this means
$$i = 4, pi = 5 \Rightarrow \binom{5}{4} = 5 > I^* = 2 \Rightarrow \text{discard}$$
- **Step 1:** Assume the fourth “one” is in the next highest position possible, this means
$$i = 4, pi = 4 \Rightarrow \binom{4}{4} = 1 < I^* = 2 \Rightarrow \mathbf{p_4 = 4}, I^* = |1 - 2| = 1$$
- **Step 2:** Assume the third “one” is in the highest position possible, this means
$$i = 3, pi = 3 \Rightarrow \binom{3}{3} = 1 = I^* = 1 \Rightarrow \mathbf{p_3 = 3}, I^* = |1 - 1| = 0$$
- **Step 3:** Assume the second “one” is in the highest position possible, this means
$$i = 2, pi = 2 \Rightarrow \binom{2}{2} = 1 > I^* = 0 \Rightarrow \text{discard}$$
- **Step 4:** Assume the second “one” is in the next highest position possible, this means
$$i = 2, pi = 1 \Rightarrow \binom{1}{2} = 0 = I^* = 0 \Rightarrow \mathbf{p_2 = 1}, I^* = |0 - 0| = 0$$
- **Step 5:** Assume the first “one” is in the highest position possible, this means
$$i = 1, pi = 0 \Rightarrow \binom{0}{1} = 0 = I^* = 0 \Rightarrow \mathbf{p_1 = 0}$$

This way we obtained the position of the four “ones” in the number, and index 2 corresponds to element $011011 \equiv 27$, and you can check Table 3.1 and see that we obtained a correct result.

Preallocation

One very important detail about the PETSc library when creating `Mat` objects is the *preallocation*. Creation of a `Mat` object has three required steps: first, a call to the function `MatSetSizes(Mat,...)`; second, the elements specification, that includes one or many calls to the function `MatSetValues(Mat,...)`, where one assigns values to the non zero elements of the matrix; and finally, the assembly, a call to the pair of functions `MatAssemblyBegin(Mat,MAT_FINAL_ASSEMBLY)` and `MatAssemblyEnd(Mat,MAT_FINAL_ASSEMBLY)` that leave the `Mat` object in the correct state for it to be used. When dealing with sparse matrices, not all the memory stated in the creation is used, so in principle, it is not allocated at the beginning. This means that one is using that memory dynamically, which generates an overhead in the calls to the `MatSetValues(Mat,...)` function. For this matter, preallocation is needed. If one is able to provide the sparsity pattern, the place where the non zero elements of the matrix are, before filling the matrix, the allocation is done at the beginning and memory is no longer treated dynamically, and this speeds up the code considerably. The code does this already, and if one wants to add new terms to the Hamiltonian has to keep it in mind. Some advice could be that it is always better to "over preallocate" memory rather than "under preallocating" (which might cause a runtime error) or no preallocating at all. Once one calls `MatAssembly` functions, the extra memory is freed.

The procedure to preallocate memory is very similar to the way we set the values of the matrix: split the elements among the processes, call the function that computes the element(s) and call the PETSc function that sets the element(s) in the matrix. In here, instead of calling `MatSetValues()` we call `MatMPIAIJSetPreallocation()`. Another difference is that after calling the functions that compute the element(s) we call the `prealloc_info()` function, that keeps the count on the number of non-zero elements.

Just in case you were wondering how much preallocating improves the execution time, in Table 3.3 you can see some results. It is visible that, while the creation of the Hamiltonian is only less than two times slower when preallocating, the building of the off-diagonal part is 80 and 135 times faster, for the case of 26 and 28 spins, respectively.

# spins	hamilt (s)			offdiag (s)		
	no	yes	ratio	no	yes	ratio
26	1.103	2.274	0.48	138.7	1.709	81.15
28	3.261	8.003	0.40	817.8	6.031	135.6

Table 3.3: Execution time (in seconds) of the creation of the Hamiltonian (`hamilt`) and the building of the off-diagonal part (`offdiag`) with and without preallocation, for 26 and 28 spins, run with 16 nodes.

3.5 Solver

After the matrix is assembled everything is ready for its diagonalization. The solver is initialized and the eigenpairs are obtained. This part is handled by the `Solver` class. For the matrix diagonalization, SLEPc handles all the work, providing that the user correctly initializes and set the solver parameters.

The following is the normal setup for a SLEPc program. The user is able to tweak the `solver` object during runtime because of the presence of the call to the `EPSSetFromOptions` function.

```
EPS solver;
```

```
PetscInt nconv;

EPSCreate(PETSC_COMM_WORLD, &solver);
EPSSetOperators(solver,matrix,NULL);
EPSSetProblemType(solver, EPS_HEP);
EPSSetWhichEigenpairs(solver, EPS_SMALLEST_REAL);
EPSSetFromOptions(solver);

EPSSolve(solver);
EPSGetConverged(solver,&nconv);
EPSGetEigenpair(solver,...);
EPSDestroy(&solver);
```

3.6 Spin operators

After one constructs the matrix and obtains the desired eigenstates, one compute some quantities with those states. For example, two-spin correlation functions, magnetization or the dynamical structure factor. The file `include/correlation.hpp` contains the declarations of a few classes. As it happens with the `Lattice` class, there is a base class, named `Correlation`, and from that, one can derive different kinds of operators. Let us take a look at the base class (in Listing 3.9) in order to understand how this works.

Listing 3.9: Correlation base class.

```
class Correlation {
public:

  virtual void OpOnBasisElems(Environment& env,
                              PetscInt spin_i,
                              PetscInt spin_j,
                              PetscInt basis_elem) =
   0;
  virtual PetscErrorCode OpCreate(Environment& env,
                                  PetscInt spin_i,
                                  PetscInt spin_j,
                                  Mat& OpIJ) = 0;
  virtual PetscErrorCode OpOnStateVector(Environment&
   env, PetscInt spin_i, PetscInt spin_j, Vec& state,
   Vec& rhs);
  virtual PetscScalar ExpectVal(Environment& env,
                                PetscInt spin_i,
                                PetscInt spin_j,
                                Vec& state);
  virtual ~Correlation() {}
};
```

When one computes the correlation between two spins for the operator S^z, for example, one has to do

$$C_{ij}^{zz} = \langle\Psi_0|S_i^z S_j^z|\Psi_0\rangle \tag{3.12}$$
$$= \langle\sum_m c_m^* \phi_m|S_i^z S_j^z|\sum_n c_n\phi_n\rangle \tag{3.13}$$
$$= \sum_{m,n} c_m^* c_n \langle\phi_m|S_i^z S_j^z|\phi_n\rangle, \tag{3.14}$$

where Φ_0 is the ground state and ϕ_m are the basis elements. This means that one has to know how to apply the operator, which in this case is $S_i^z S_j^z$, to the basis elements, and for that one needs to know how to construct said operator. All the steps shown in Equation 3.12 (and the ones that follow) are mimicked in the class: `OpCreate` creates the operator $S_i^z S_j^z$, `OpOnBasisElems` computes $S_i^z S_j^z|\phi_n\rangle$, `OpOnStateVector` computes the vector $\sum_n c_n(S_i^z S_j^z|\Psi_n\rangle)$, and, finally, `ExpectVal` does the dot product between the vector obtained from the previous function and the vector $\langle\Phi_0|$.

Derived classes, like `SzSz`, `SpSm` and `SiSj` just implement the functions that are specific for each case, as `OpCreate` and `OpOnBasisElems`. If you want add a new operator, just need to provide those functions, like, for example, in the case of `SpSm` (you can check the implementation on the files `include/correlation.hpp` and `src/correlation.cpp`).

3.7 Physical quantities

There are some quantities, including the correlation functions, that can be computed with this code: the order parameters, m_{AF}^2 and m_{STR}^2, given in Equations 3.15 and 3.16, and the dynamical structure factor (DSF), given in Equations 3.17 and 3.18.

$$m_{AF}^2 = \frac{8}{N(N+4)}\left[\sum_\alpha \sum_{i,j\in\alpha} \langle \mathbf{S}_i \cdot \mathbf{S}_j\rangle\right]_J \tag{3.15}$$
$$m_{STR}^2 = \frac{4}{N(N+4)}\left[\sum_\nu \sum_{\alpha_\nu} \sum_{i,j\in\alpha} \langle \mathbf{S}_i \cdot \mathbf{S}_j\rangle\right]_J \tag{3.16}$$

$$S_{\vec{q}}(w) = \sum_{n=1}^{nev} \left| \left\langle \Psi_n^X \middle| \hat{S}_{\vec{q}}^\alpha \middle| \Psi_0 \right\rangle \right|^2 \delta(\omega - (E_n^X - E_0)) \tag{3.17}$$

$$S_{\vec{q}}(w) = \sum_{n=1}^{nev} \left| \left\langle \Psi_n^X \middle| \hat{S}_{\vec{q}}^\alpha \middle| \Psi_0 \right\rangle \right|^2 \frac{1}{\pi} \frac{\epsilon^2}{(\omega - (E_n^X - E_0))^2 + \epsilon^2}$$

where

$$\hat{S}_{\vec{q}}^\alpha = \frac{1}{\sqrt{L}} \sum_{i=0}^{L-1} e^{i\vec{q}\cdot\vec{r}_i} \hat{S}_i^\alpha. \tag{3.18}$$

In the case of the order parameters, these expressions correspond to a 2D square lattice, as for other types of lattices the pre-factor changes according to the number of neighbours that a spin has in the lattice. For the DSF, we can write the operand explicitly inside the equation to obtain the version that is more similar to the one coded.

$$S_{\vec{q}}(w) = \sum_{n=1}^{nev} \left| \left\langle \Psi_n^X \middle| \frac{1}{\sqrt{L}} \sum_{i=0}^{L-1} e^{i\vec{q}\cdot\vec{r}_i} \hat{S}_i^\alpha \middle| \Psi_0 \right\rangle \right|^2 \frac{1}{\pi} \frac{\epsilon^2}{(\omega - (E_n^X - E_0))^2 + \epsilon^2} \tag{3.19}$$

$$S_{\vec{q}}(w) = \sum_{n=1}^{nev} \left| \frac{1}{\sqrt{L}} \sum_{i=0}^{L-1} e^{i\vec{q}\cdot\vec{r}_i} \left\langle \Psi_n^X \middle| \hat{S}_i^\alpha \middle| \Psi_0 \right\rangle \right|^2 \frac{1}{\pi} \frac{\epsilon^2}{(\omega - (E_n^X - E_0))^2 + \epsilon^2} \tag{3.20}$$

The three quantities are declared in the `include/phys.hpp` file, and defined in `include/phys.tmpl_func.hpp`.

3.7.1 Output

It is important to mention which is the output from the DSF function. It output a file, named `dsf_d#.dat` where `#` is the number of the disorder realization, and the content consist in a first part that is "commented" (every line starts with `#`), and then a two-column table. The commented part contains the following information:

- Number of spins: `nspins`
- Number of eigenvalues computed: `nev`
- Ground state energy: `GS`
- Hamiltonian parameters: J_1, Δ_1, J_2 and Δ_2
- Values of the disorder coefficients: h_i

In the data part, the first column of the table contains the difference in energy between the ground state and the excited states, that is $E_n^X - E_0$ in Equation 3.20. The second column is the expectation value $\left\langle \Psi_n^X \middle| \hat{S}_i^\alpha \middle| \Psi_0 \right\rangle$ from the same Equation. Is a task for the user to process the data and compute the DSF. In any case, there are a few Python scripts (actually Jupyper notebooks) that do this for you. These can be found under the installation path in the `postprocessing` folder.

Chapter 4

Tutorials

4.1 Create a simple `main` function

Let us create a very simple `main` function to compute the first 10 eigenstates of a XXZ Hamiltonian with nearest neighbours interactions only, and no disorder. This means

$$\mathcal{H} = J_1 \sum_{\langle i,j \rangle} \{ \frac{1}{2}(S_i^+ S_j^- + S_i^- S_j^+) + \Delta_1 S_i^z S_j^z \}. \tag{4.1}$$

Let us suppose also that we do not want to fix the value of J_1 but we want to fix the value of $\Delta_1 = 0.5$. This Hamiltonian represents the interactions of spins in a 2D square lattice with an undefined size.

1. Start by creating a `help` variable, that explains what the code is about. It is not mandatory, but it is always helpful. Also create the `main()` function.

   ```
   static char help[] = "Diagonalization of XXZ-Heisenberg
   Hamiltonian with nearest neighbours interactions in
   a 2D square lattice\n.";

   int main(int argc, char * argv[]) {}
   ```

2. PETSc and SLEPc functions usually have as a return value a `PetscErrorCode`-type of element. Many of the functions in this code have this return value as well. So, we will define a variable called `ierr` (a name that is conventially given to this return value by PETSc). This value is checked by a PETSc macro `CHKERRQ(ierr)`, and we will use it as well.

   ```
   static char help[] = "....";
   int main(int argc, char * argv[]) {
     PetscErrorCode ierr;

     return ierr;
   }
   ```

3. The first thing to do is to instantiate an `Environment` object. For this we also need to include the corresponding file. We will include directly the `Hamiltonian.hpp` file, as this one already includes many of the "basic" files, but you can include them separately, one by one. As we do not want to fix the number of spins in the system, we only construct the `Environment` object with three argument: `argc`, `argv` and `help`.

```
#include <hamiltonian.hpp>

static char help[] = "....";
int main(int argc, char * argv[]) {
  PetscErrorCode ierr;

  Environment env{argc,argv,help};

  return ierr;
}
```

4. Now, we need to parse the command line arguments. For this, we use the `parse_args()` function, and we need to include the corresponding file. We would like also to print the parameters' values on screen, and for that we use `print_args()` function. As we know that three out of four Hamiltonian parameters have a fixed value, we can also specify that.

```
#include <hamiltonian.hpp>
#include <parse_args.hpp>
#include <print_args.hpp>

static char help[] = "....";
int main(int argc, char * argv[]) {
  PetscErrorCode ierr;

  Environment env{argc,argv,help};

  parse_args();
  Delta1 = 0.5;
  J2 = 0.0;
  Delta2 = 0.0;
  print_args();

  return ierr;
}
```

5. Create the basis and the lattice. In the case of the `Lattice` object, as we do not want to fix the dimensions, we just put the two variables (which are defined inside the `parse_args()` function): `lx` and `ly`.

```
#include <hamiltonian.hpp>
#include <parse_args.hpp>
#include <print_args.hpp>

static char help[] = "....";
int main(int argc, char * argv[]) {
  PetscErrorCode ierr;

  Environment env{argc,argv,help};

  parse_args();
  Delta1 = 0.5;
  J2 = 0.0;
  Delta2 = 0.0;
  print_args();

  Basis b{env,0};
  square2D lat{env,lx,ly};

  return ierr;
}
```

6. Create the `Hamiltonian` object, and fill in the elements corresponding to the diagonal and off-diagonal parts of the nearest neighbours interaction. The function `MatView()` is a PETSc function that allows us to take a look at the matrix, it prints it on screen. It is commented out because for “large” systems the matrix is too big and it does not make much sense to print it, this is just to check small cases.

```
#include <hamiltonian.hpp>
#include <parse_args.hpp>
#include <print_args.hpp>

static char help[] = "....";
int main(int argc, char * argv[]) {
  PetscErrorCode ierr;

  Environment env{argc,argv,help};

  parse_args();
  Delta1 = 0.5;
  J2 = 0.0;
  Delta2 = 0.0;
  print_args();

  Basis b{env,0};
  square2D lat{env,lx,ly};

  Hamiltonian<square2D> h{env,b,lat,J1,Delta1,J2,Delta2};
  ierr = h.build_diag(env); CHKERRQ(ierr);
  ierr = h.build_off_diag(env); CHKERRQ(ierr);
  // ierr = MatView(h.hamilt, PETSC_VIEWER_STDOUT_WORLD);
  // CHKERRQ(ierr);

  return ierr;
}
```

7. Create the solver and solve the Hamiltonian, asking for 10 eigenstates. For this we have to create an `EPS` object (that belongs to the SLEPc library) and we need an integer (a `PetscInt` in particular) for the number of converged eigenstates: `eigen`. This one can differ from the number of eigenstates asked.

   Retrieve the solution and print the eigenvalues. We need two `double` numbers: `eigen` and `error`, for the eigenvalue and the error, respectively. We also need a PETSc `Vec` object, which is basically a vector, to store the eigenvector. This requires also the creation of the `Vec`, which is done via `MatCreateVecs()`, that sets the dimensions of the vector to be consistent with the matrix.

```
#include <hamiltonian.hpp>
#include <parse_args.hpp>
#include <print_args.hpp>

static char help[] = "....";
int main(int argc, char * argv[]) {
  PetscErrorCode ierr;
  EPS solver;
  PetscInt nconv;
  double eigen, error;
  Vec state;

  Environment env{argc,argv,help};

  parse_args();
  Delta1 = 0.5;
  J2 = 0.0;
  Delta2 = 0.0;
  print_args();

  Basis b{env,0};
  square2D lat{env,lx,ly};
  Hamiltonian<square2D> h{env,b,lat,J1,Delta1,J2,Delta2};
  ierr = h.build_diag(env); CHKERRQ(ierr);
  ierr = h.build_off_diag(env); CHKERRQ(ierr);
  // ierr = MatView(h.hamilt, PETSC_VIEWER_STDOUT_WORLD);
  // CHKERRQ(ierr);

  ierr = Solver::SolverInit(solver,h.hamilt,10); CHKERRQ(ierr);
  ierr = Solver::solve(env,solver,nconv); CHKERRQ(ierr);
  ierr = MatCreateVecs(h.hamilt,&state,NULL); CHKERRQ(ierr);
  for (int i = 0; i < 10; ++i) {
    ierr = Solver::solution(solver,i,eigen,state,error);
    CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"%d  %10.5f\n",i,eigen);
    CHKERRQ(ierr);
  }

  return ierr;
}
```

8. Finally, release the memory by deleting the `Vec` and the `EPS` objects.

```
#include <hamiltonian.hpp>
#include <parse_args.hpp>
#include <print_args.hpp>

static char help[] = "....";
int main(int argc, char * argv[]) {
  PetscErrorCode ierr;
  EPS solver;
  PetscInt nconv;
  double eigen, error;
  Vec state;

  Environment env{argc,argv,help};

  parse_args();
  Delta1 = 0.5;
  J2 = 0.0;
  Delta2 = 0.0;
  print_args();

  Basis b{env,0};
  square2D lat{env,lx,ly};

  Hamiltonian<square2D> h{env,b,lat,J1,Delta1,J2,Delta2};
  ierr = h.build_diag(env); CHKERRQ(ierr);
```

```
    ierr = h.build_off_diag(env); CHKERRQ(ierr);
    // ierr = MatView(h.hamilt, PETSC_VIEWER_STDOUT_WORLD);
    // CHKERRQ(ierr);

    ierr = Solver::SolverInit(solver,h.hamilt,10); CHKERRQ(ierr);
    ierr = Solver::solve(env,solver,nconv); CHKERRQ(ierr);
    ierr = MatCreateVecs(h.hamilt,&state,NULL); CHKERRQ(ierr);
    for (int i = 0; i < 10; ++i) {
      ierr = Solver::solution(solver,i,eigen,state,error);
      CHKERRQ(ierr);
      ierr = PetscPrintf(PETSC_COMM_WORLD,"%d  %10.5f\n",i,eigen);
      CHKERRQ(ierr);
    }
    ierr = VecDestroy(&state); CHKERRQ(ierr);
    ierr = Solver::SolverClean(solver); CHKERRQ(ierr);

    return ierr;
  }
```

The full code is located inside the installation directory under `examples/ex1.cpp`.

To run this code you have to give the following command line arguments:

```
-n <num_spins> -ltype square2D -lx <x-dim> -ly <y-dim>
```

4.2 Ground state with disorder

In this example we would like to obtain the ground state energy of a 1D system with next-nearest neighbours interactions only. This means that the Hamiltonian is given by

$$\mathcal{H} = J_2 \sum_{\langle\langle i,j \rangle\rangle} \{ \frac{1}{2}(S_i^+ S_j^- + S_i^- S_j^+) + \Delta_2 S_i^z S_j^z \}. \tag{4.2}$$

We want to fix the value of $J_2 = 1.0$ but not of Δ_2. We would like to have a disorder strength between -0.3 and 0.4; we would like to be able to change at runtime the number of repetitions we do, and we would like to have completely random numbers, meaning that we do not want the seed to be fixed.

The structure of the code is very similar to the one in Section 4.1, so we will go fast on those parts and focus on the disorder part.

1. Create the `Environment`, the `Basis`, the `Lattice` and the `Hamiltonian` as in Section 4.1.

```
#include <parse_args.hpp>
```

```
#include <print_args.hpp>
#include <hamiltonian.hpp>

static char help[] = "...";
int main(int argc, char * argv[]) {

  PetscErrorCode ierr;
  EPS solver;

  // Initialize the environment
  Environment env{argc,argv,help};

  // Read command line arguments
  parse_args();
  Delta1 = 0.0;
  J1 = 0.0;
  J2 = 1.0;
  min_dis = -0.3;
  max_dis = 0.4;
  reprod = PETSC_FALSE;
  print_args();

  // Create Basis
  Basis b{env,0};

  // Create Lattice
  chain1D lat{env};

  // Create Hamiltonian
  Hamiltonian<chain1D> h{env,b,lat,J1,Delta1,J2,Delta2};
  ierr = h.build_diag(env); CHKERRQ(ierr);
  ierr = h.build_off_diag(env); CHKERRQ(ierr);

  return ierr;
}
```

2. Now we create the things needed to compute the disorder part of the Hamiltonian and create a loop to do all the disorder realizations.

```
#include <parse_args.hpp>
#include <print_args.hpp>
#include <hamiltonian.hpp>
#include <random_disorder.hpp>
#include <solver.hpp>

static char help[] = "...";
```

```
int main(int argc, char * argv[]) {

  PetscErrorCode ierr;
  EPS solver;
  PetscInt nconv;
  double eigen, error;
  Vec state;

  // Initialize the environment
  Environment env{argc,argv,help};

  // Read command line arguments
  parse_args();
  Delta1 = 0.0;
  J1 = 0.0;
  J2 = 1.0;
  min_dis = -0.3;
  max_dis = 0.4;
  reprod = PETSC_FALSE;
  print_args();

  // Create Basis
  Basis b{env,0};

  // Create Lattice
  chain1D lat{env};

  // Create Hamiltonian
  Hamiltonian<chain1D> h{env,b,lat,J1,Delta1,J2,Delta2};
  ierr = h.build_diag(env); CHKERRQ(ierr);
  ierr = h.build_off_diag(env); CHKERRQ(ierr);

  // Create random disorder strengths
  Tools::RandomDisorder rd{env.nspins,reprod,min_dis,max_dis,rep};

  ierr = MatCreateVecs(h.hamilt,&state,NULL); CHKERRQ(ierr);

  // Solve for every disorder realization
  for (int i = 0; i < rep; ++i) {
    rd.hi_init(i);
    ierr = h.build_disorder(env,rd.hi); CHKERRQ(ierr);
    ierr = Solver::SolverInit(solver,h.hamilt,10); CHKERRQ(ierr);
    ierr = Solver::solve(env,solver,nconv); CHKERRQ(ierr);
    ierr = Solver::solution(solver,0,eigen,state,error); CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"%d %10.5f\n",i,eigen);
```

```
      CHKERRQ(ierr);
      ierr = Solver::SolverClean(solver); CHKERRQ(ierr);
    }

    ierr = VecDestroy(&state); CHKERRQ(ierr);
    return ierr;
  }
```

The full code is available under the installation path in the `examples/ex2.cpp` file.

To run this code you have to give the following command line arguments:

```
-n <num_spins> -ltype chain1D -disorder -rep <num_repetitions>
-d2 <d2_value>
```

4.3 Compute physical quantities

We would like to compute some physical quantities, like the order parameter and the dynamical structure factor (DSF). For the first ones, we need only the ground state of the system. The starting point of this example is the `main` function of Section 4.1, as we are not going to include disorder, with the difference that we will ask for only one eigenstate.

1. Start with a `main` function as the one in Section 4.1.

2. Create the AF and STR sublattices and compute the squared magnetization. You need to include the file `phys.hpp` where the physical quantities are defined.

```
#include <parse_args.hpp>
#include <print_args.hpp>
#include <hamiltonian.hpp>
#include <solver.hpp>
#include <phys.hpp>

static char help[] = "...";

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

  PetscErrorCode ierr;
  EPS solver;
  PetscInt nconv;
  double eigen, error;
  Vec state;
```

```
// Initialize the environment
Environment env{argc,argv,help};

// Read command line arguments
parse_args();
Delta1 = 0.5;
J2 = 0.0;
Delta2 = 0.0;
print_args();

// Create Basis
Basis b{env,0};

// Create Lattice
square2D lat{env,lx,ly};

// Create sublattices
AF<square2D> sub_af{lat,2};
Striped<square2D> sub_str{lat,4};

// Create Hamiltonian
Hamiltonian<square2D> h{env,b,lat,J1,Delta1,J2,Delta2};
ierr = h.build_diag(env); CHKERRQ(ierr);
ierr = h.build_off_diag(env); CHKERRQ(ierr);

// Initiate solver and solve
ierr = Solver::SolverInit(solver,h.hamilt,1); CHKERRQ(ierr);
ierr = Solver::solve(env,solver,nconv); CHKERRQ(ierr);

// Print groundstate
ierr = MatCreateVecs(h.hamilt,&state,NULL); CHKERRQ(ierr);
ierr = Solver::solution(solver,0,eigen,state,error); CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"%d %10.5f\n",0,eigen);
CHKERRQ(ierr);

// Compute order paramters
magAF = Phys::SquaredSubLattMagAF<square2D>(env,b,sub_af,state);
magSTR = Phys::SquaredSubLattMagSTR<square2D>(env,b,sub_str,state);
ierr = PetscPrintf(PETSC_COMM_WORLD,"magAF: %.10f\n",magAF);
CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"magSTR: %.10f\n",magSTR);
CHKERRQ(ierr);

ierr = VecDestroy(&state); CHKERRQ(ierr);
```

```
  ierr = Solver::SolverClean(solver); CHKERRQ(ierr);
  return ierr;
}
```

3. Now, let us compute the DSF. For this, we need to create a `DSF_data` object, that is used to send information about the basis and the Hamiltonians to the DSF function. Then, we just call function that computes it. Just a bit on the data structure.

   - There are two basis, `b0` and `b1`, which correspond to the basis of the Hilbert space of the ground state (`b0`) and of the space obtained after applying the operator S_i^α from Equation 3.18.
   - The same happens with the Hamiltonians, there are two of them, `h0` and `h1`, which correspond to the matrices created with the basis `b0` and `b1`, respectively.
   - There is the lattice used (`lat`).
   - There is the number of eigenstates desired (`nev`).
   - There is the path to where the file will be saved (`path`).
   - There is a vector with the disorder values for each site (`hi`).

```
#include <parse_args.hpp>
#include <print_args.hpp>
#include <hamiltonian.hpp>
#include <solver.hpp>
#include <phys.hpp>

static char help[] = "...";

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

  PetscErrorCode ierr;
  EPS solver;
  PetscInt nconv;
  double eigen, error;
  Vec state;

  // Initialize the environment
  Environment env{argc,argv,help};

  // Read command line arguments
  parse_args();
  Delta1 = 0.5;
  J2 = 0.0;
  Delta2 = 0.0;
```

```
print_args();

// Create Basis
Basis b{env,0};

// Create Lattice
square2D lat{env,lx,ly};

// Create sublattices
AF<square2D> sub_af{lat,2};
Striped<square2D> sub_str{lat,4};

// Create Hamiltonian
Hamiltonian<square2D> h{env,b,lat,J1,Delta1,J2,Delta2};
ierr = h.build_diag(env); CHKERRQ(ierr);
ierr = h.build_off_diag(env); CHKERRQ(ierr);

// Initiate solver and solve
ierr = Solver::SolverInit(solver,h.hamilt,1); CHKERRQ(ierr);
ierr = Solver::solve(env,solver,nconv); CHKERRQ(ierr);

// Print groundstate
ierr = MatCreateVecs(h.hamilt,&state,NULL); CHKERRQ(ierr);
ierr = Solver::solution(solver,0,eigen,state,error); CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"%d %10.5f\n",0,eigen);
CHKERRQ(ierr);

// Compute order parameters
magAF = Phys::SquaredSubLattMagAF<square2D>(env,b,sub_af,state);
magSTR = Phys::SquaredSubLattMagSTR<square2D>(env,b,sub_str,state);
ierr = PetscPrintf(PETSC_COMM_WORLD,"magAF: %.10f\n",magAF);
CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"magSTR: %.10f\n",magSTR);
CHKERRQ(ierr);

// Create data object for DSF
Phys::DSF_data<square2D> dsf_data{};
dsf_data.b0 = &b;
dsf_data.b1 = &b;
dsf_data.lat = &lat;
dsf_data.h0 = &h;
dsf_data.h1 = &h;
dsf_data.nev = 10;
dsf_data.ncv = PETSC_DECIDE;
dsf_data.mpd = PETSC_DECIDE;
```

```
    dsf_data.disorder = PETSC_FALSE;
    dsf_data.hi = NULL;
    dsf_data.path = "./";

    // Compute DSF
    ierr = Phys::DynStructFactor<square2D,Sqz>(env,dsf_data,eigen,state,0);

    ierr = VecDestroy(&state); CHKERRQ(ierr);
    ierr = Solver::SolverClean(solver); CHKERRQ(ierr);
    return ierr;
  }
```

The full code is under the installation path, on the file `examples/ex3.cpp`. In the file `examples/ex4.cpp` there is the code for the same system as in this case, but for the case of S_i^+ instead of S_i^z.

4.4 Add new lattice

Suppose we want to add a new lattice to the code, let us say the honeycomb lattice. In principle, one has two different approaches: you can either add it to the source code (the library itself), which implies that you will have to recompile (and possibly, reinstall it); or, you can add it on your personal project. The difference is on how and where to include the new files, and the command line to compile the new code. We will (try to) explain how to do both of them.

For a new lattice we need:

- A way of creating the vector with the nearest and next-nearest neighbours. Depending on the use you want to give to this new lattice, maybe you would want only the nearest neighbours. This is also possible, just adding some of the features.
- A way of creating the AF and Striped sublattices (for the computation of the magnetizations). Again, these are optional if you are not planning to use them.
- To take care of the `parse_args()` function, and make sure is able to process the new lattice arguments.
- To make sure that the `enum class lattice_type` considers the new lattice type.

4.4.1 Add to the source code

1. **Add the class declaration in the `include/lattice.hpp` file:** See Listing 4.1.

Listing 4.1: include/lattice.hpp

```
class honeycomb2D : public Lattice {

private:
  PetscInt nspins_x;
  PetscInt nspins_y;
  lattice_type type;
  std::vector<PetscInt>& get_nn();
  std::vector<PetscInt>& get_nnn();

  // Uncomment if you set nn, nnn, num_nnXsite and
    num_nnnXsite as private
  /*
  template<typename L> friend class Hamiltonian;
  template<int size,typename L> friend class AF;
  template<int size,typename L> friend class
   Striped;
  template<typename L> friend PetscInt
   HamiltHelper::get_elem_diag(neigh_type nt,
                           PetscInt basis_elem,
                           PetscInt nspins,
                           L& lattice);
  template<typename L> friend void HamiltHelper::
   get_coup_elems(neigh_type nt,
                        Basis& basis,
                        PetscInt elem,
                        L& lattice,
                        PetscInt& ncol,
                        PetscInt* coup_elems);
  */

public:
  std::vector<PetscInt> nn;
  std::vector<PetscInt> nnn;
  std::vector<PetscInt> nnXsite;
  std::vector<PetscInt> nnnXsite;

public:

  honeycomb2D(Environment& env, PetscInt
   m_nspins_x, PetscInt m_nspins_y);
  lattice_type get_type() const {return type;}
  PetscInt num_nn() {return 2;}
  PetscInt num_nnn() {return 3;}
  PetscInt num_nnXsite(PetscInt lat_site) {return
   nnXsite[lat_site];}
  PetscInt num_nnnXsite(PetscInt lat_site) {return
    nnnXsite[lat_site];}
};
```

2. **Add the class definition to the** `src/lattice.cpp` **file:** See Listing 4.2.

Listing 4.2: src/lattice.cpp

```
// Constructor
honeycomb2D::honeycomb2D(Environment& env,
       PetscInt m_nspins_x,
       PetscInt m_nspins_y)
  : Lattice{env},
    nspins_x{m_nspins_x},
    nspins_y{m_nspins_y},
    type{lattice_type::honeycomb2D}
{
    nn = get_nn();
    nn.shrink_to_fit();
    nnn = get_nnn();
    nnn.shrink_to_fit();
}

/* ------------------------------------------- */
// Function to compute nearest neighbours

std::vector<PetscInt>& honeycomb2D::get_nn() {

  PetscInt neighA, neighB;
  PetscInt ix1,iy1;
  int current_spin;

  for (PetscInt ix = 0; ix < nspins_x; ++ix)
    for (PetscInt iy = 0; iy < nspins_y; ++iy) {

      current_spin = ix * nspins_y + iy;
      nn.push_back(current_spin);

      // Even rows of the lattice (xi even)
      if (ix%2 == 0) {
      if (current_spin%2 == 0) {
          iy1 = (iy+1)%nspins_y;
          neighA = ix * nspins_y + iy1;

          ix1 = (ix+1)%nspins_x;
          neighB = ix1 * nspins_y + iy;

            nn.push_back(neighA);
          nn.push_back(neighB);
          num_nnXsite.push_back(2);
      }
      else {
          iy1 = (iy+1)%nspins_y;
          neighA = ix * nspins_y + iy1;
          nn.push_back(neighA);
          nn.push_back(-1);
          num_nnXsite.push_back(1);
      }
      }

      // Odd rows of the lattice (xi odd)
      else {

      if (current_spin%2 != 0) {
          iy1 = (iy+1)%nspins_y;
          neighA = ix * nspins_y + iy1;

          ix1 = (ix+1)%nspins_x;
          neighB = ix1 * nspins_y + iy;

          nn.push_back(neighA);
          nn.push_back(neighB);
          num_nnXsite.push_back(2);
      }
      else {
          iy1 = (iy+1)%nspins_y;
          neighA = ix * nspins_y + iy1;
          nn.push_back(neighA);
          nn.push_back(-1);
          num_nnXsite.push_back(1);
      }
    }
  }
  return nn;
}
```

```
/* ------------------------------------------ */
// Function to compute next-nearest neighbours

std::vector<PetscInt>& honeycomb2D::get_nnn() {

  PetscInt neighA, neighB, neighC;
  PetscInt ix1,iy1,iy2,iym1;
  int current_spin;

  for (PetscInt ix = 0; ix < nspins_x; ++ix)
    for (PetscInt iy = 0; iy < nspins_y; ++iy) {
      current_spin = ix * nspins_y + iy;
      nnn.push_back(current_spin);
      num_nnnXsite.push_back(3);

      iy2 = (iy+2)%nspins_y;
      neighA = ix * nspins_y + iy2;

      ix1 = (ix+1)%nspins_x;
      iy1 = (iy+1)%nspins_y;
      neighB = ix1 * nspins_y + iy1;

      iym1 = (iy-1+nspins_y)%nspins_y;
      neighC = ix1 * nspins_y + iym1;

      nnn.push_back(neighA);
      nnn.push_back(neighB);
      nnn.push_back(neighC);
    }

  return nnn;
}
```

3. **Check, and add if necessary, that the new type of lattice is included in `lattice_type` class:** In the file `include/lattice.hpp`, add the new type to the `lattice_type` class, as you can see in Listing 4.3.

Listing 4.3: Add type to enum class `lattice_type`.

```
enum class lattice_type {
  chain1D, /**< 1D chain type of lattice. */
  square2D, /**< 2D square type of lattice. */
```

```
    honeycomb2D /**< 2D honeycomb type of lattice.
     */
  };
```

4. **Check `parse_args()` function, and modify according to new lattice type:** (if needed)
5. **Add the definition of the sublattices to `src/sublattice.cpp` file:** This is only needed if one wants to compute the antiferromagnetic and striped magnetization. We have to give the definition of `AF<honeycomb2D>::construct_sublat()` and `Striped<honeycomb2D>::construct_sublat()` functions, which are the ones in charge of constructing the sublattice lists.
6. **Recompile (and reinstall) the library:** Inside the `build` folder just type `ninja` or `ninja install`.

4.4.2 Add to personal project

This way is pretty similar to the previous one, just a few small changes.

1. **Create a file named `honeycomb.hpp`:** It should be the same as in Listing 4.1, you just need to add the line `#include "lattice.hpp"` at the beginning.
2. **Create a file named `honeycomb.cpp`:** It has to have the same content as the one in Listing 4.2, just add the line `#include "honeycomb.hpp"` at the beginning.
3. **Check, and add if necessary, that the new type of lattice is included in `lattice_type` class:** If it is not present, you will have to add it (as said in item 5 from the previous subsection), and you will have to recompile and reinstall the library anyway.
4. **Add sublattices to compute magnetization:** Add the definitions of `AF<honeycomb2D>::construct_sublat()` and `Striped<honeycomb2D>::construct_subl` functions to a file named `sublattice.cpp` and also the line at the beginning `#include "sublattice.hpp"`.
5. **Compile:** Assuming that you created a `main.cpp` file with the main function; assuming also that you installed the library and the `pkgconfig` file is in the PKG_CONFIG_PATH environment variable, then you need to do: (all in the same command line)

```
$ mpicxx honeycomb.cpp sublattice.cpp main.cpp
`pkg-config --cflags --libs LSQuantumED` -o main.x
```

Bibliography

[1] D. E. Knuth. The Art of Computer Programming, volume 4. Addison-Wesley, 2009.