Institute for Theoretical Physics
University of Cologne

Prof. Dr. Simon Trebst
Peter Bröcker, Johannes Helmes

Computational Many-Body Physics

Assignment 5

Summer Term 2015

website: http://www.thp.uni-koeln.de/trebst/Lectures/2015-CompManyBody.shtml
due date: Monday, June 29th, 18:00 - send solutions to broecker [at] thp.uni-koeln.de

11. Matrices are a snake's best friend *programming techniques*

In this exercise, we will take a closer look at how to perform **linear algebra** operations in python. As usual, python offers a very intuitive and simple interface that will feel very familiar for those of you who know Matlab. Users of C++ should take a look at Eigen or Armadillo both of which are high-level frontends for the most popular linear algebra package LAPACK. Let it be said though, that if your program spends most of its time doing linear algebra operations, programming in python might save you a lot of time because `numpy` also interfaces `LAPACK` but with all the advantages of python. Specifically, you do not have to worry about the actual function calls and how your matrices are laid out in memory.

Let us recap the most important elements that `numpy` provides us with. An **empty matrix** is constructed using

```
a_matrix = np.empty((rows, cols))
```

where *rows* and *cols* are the number of rows and cols, respectively. Often, we do not need an empty matrix, but one that is filled with zeros for convenience. This can be achieved using:

```
a_matrix = np.zeros((rows, cols))
```

A **vector** can be initialized in much the same way, the only difference being that we specify only one of the dimensions:

```
a_vector = np.zeros((entries))
```

Modifying the entries of vectors and matrices is done using the usual notation with square brackets

```
print a_matrix[2, 0]
print a_vector[10]
```

The so called **slice notation** comes in particularly handy when describing submatrices. The example below shows how to access to top right corner of a larger matrix:

```
print a_matrix[0:5, 5:10]
```

As a reminder, if you leave out one of the delimiters, all values up to or starting from the value given will be used:

```
print a_matrix[:5, 5:]
```

You can also give negative numbers, which refer to values from the right end of the interval:

```
print a_matrix[:-5, -5:]
```

We now turn to linear algebra **operations**. It is very important to note that expressions like

```
matrix_3 = matrix_1 * matrix_2
another_vector = matrix * vector
```

are *not* matrix-matrix or matrix-vector but element-wise multiplications. We instead have to use the `numpy` function `dot`:

```
matrix_3 = np.dot(matrix_1, matrix_2)
another_vector = np.dot(matrix, vector)
```

Last but not least, you will have to find the eigenvalues and eigenvectors of a matrix. This is done using the `linalg` submodule and could hardly be made any easier:

```
from numpy import linalg as la
eigenvalues, eigenvectors = la.eig(a_matrix)
```

It is important to note that the eigenvalues and thus the eigenvectors are not necessarily ordered! If you are dealing with a symmetric or hermitian matrix, you can use the function `la.eigh` which takes this symmetry into account and should speed up your computation significantly.

12. A quantum of Ising

8 points + 10 points

The Ising model, which we have studied extensively in previous exercises, is a very powerful tool for understanding the basic mechanisms of magnetism in solids. Due to its classical nature it is easily accessible via Monte Carlo techniques. But at the same time, we might often want to go beyond the physics of the Ising model and study more complex phenomena. In this exercise, we therefore want to study a more complex, *quantum* version of the Ising model that goes by the name of **Quantum Ising** or **Transverse Field Ising** model. Its Hamiltonian for an arbitrary lattice is given by

$$H = J \sum_{\langle i,j \rangle} \sigma_i^z \sigma_j^z + h \sum_i \sigma_i^x. \tag{1}$$

The degrees of freedom of this model are SU(2) $s = \frac{1}{2}$ spins and σ_i denotes a Pauli matrix. The notation $\langle i, j \rangle$ signifies that the summation is carried out only over nearest neighbors on a given lattice. This transverse field Ising model is one of the simplest examples of a model that exhibits a **quantum phase transition**, i.e. a phase transition at *zero* temperature driven by a coupling parameter, in this case the strength of the transverse field h. In order to make ourselves familiar with such a quantum phase transition, we will use **exact diagonalization** techniques to study small systems.

Your task is to do the following:

- By inspecting the limiting cases $h \gg J$ and $h \ll J$ of the Hamiltonian, describe the two phases at these extremal points
- We choose the eigenstates of the σ^z operators as our eigenbasis. What is the dimension of the Hilbert space?
- Write a function that generates all possible eigenstates and compare with your previous answer to make sure that your function work correctly.
- Set up the Hamiltonian matrix by evaluating the Hamiltonian operator for each relevant

combination of the basis states. How does the limitation to nearest-neighbor interactions and on-site magnetic field help you to lower the needed computational effort?

- Diagonalize the matrix and plot the magnetization in z and x direction as a function of h.

Optional exercises: When writing down the full matrix for a very short chain, e.g. 5 sites, you will see that many of the entries are actually 0. If that is true for a significant portion of the matrix, the matrix it is said to be **sparse**. Such matrices can be diagonalized using much more efficient algorithms that reduce the computational cost significantly. You can read through the scipy.sparse documentation to find out how such matrices are efficiently stored on a computer and then use a package called ARPACK that ships with scipy to diagonalize this sparse matrix. By this rather simple modification, you will be able to study larger system sizes.

13. Mastering exact diagonalization in 10 minutes or less

2 points

Some of the most effective techniques of computational physics can be applied quite generally to a large number of very different models. In that case, it makes a lot of sense to separate the model specific code from the core of the program so that as much as possible can be reused easily when switching models or the underlying lattice. The ALPS project has pushed this idea to its limits for many state of the art methods such as exact diagonalization, various flavors of Quantum Monte Carlo and the density matrix renormalization group (DMRG), which will all be subject of the lectures to come. Its user base extends beyond theoreticians and also includes experimentalists looking to verify assumptions about the models underlying their experiments. The website includes links to binary packages and a documentation for installing from source.

Your task is to familiarize yourself a bit with the ALPS workflow which is also explained in a series of tutorials, most importantly of course the tutorials about exact diagonalization. When looking at the available models, you will notice that the transverse field Ising model is not one of them. We provide a custom model file that you should save in the same directory as the parameter file. A sample parameter file to get you started is as follows:

```
LATTICE = "chain lattice"
MODEL_LIBRARY = "alps_transverse_field_ising.xml"
MODEL = "transverse field ising"
local_S = 1/2
J = 1.0
h = 0.5
{L = 8}
```

This will simulate the transverse field Ising model on a periodic chain lattice of length $L = 8$ using the model that we have defined in the model file.

Your task: Redo and verify your calculations from the previous exercise using the ALPS code. In particular, check if your code is faster or slower than the ALPS version. Can you work with larger lattices? We want to emphasize again that while these types of simulations can become costly very quickly it is very important to use the largest possible lattice to minimize finite-size effects. Production runs easily take weeks to complete. In order to get a feel for how long what kind of lattice size takes, measure and plot the execution time for all accessible lattice sizes and make an estimate for how much longer it would take you to simulate the next bigger system.