Institute for Theoretical Physics
University of Cologne

Prof. Dr. Simon Trebst
Peter Bröcker, Johannes Helmes

Computational Many-Body Physics

Assignment 1

Summer Term 2015

website: http://www.thp.uni-koeln.de/trebst/Lectures/2015-CompManyBody.shtml
due date: Sunday, April 26th, 18:00 - send solutions to broecker [at] thp.uni-koeln.de

2. You can have your cake and eat it too *programming*

Whether you did not have any background in programming at all or are already a seasoned veteran in one of the many languages, last time's introduction to programming should have shown you that python makes scientific programming as easy as possible. This is mostly due to the fact that python offers a lot of nice features that are absent in lower level languages like Fortran and C. In many important cases, however, we trade speed for ease of use. While for the majority of programs outside the scientifc realm the lower speed is hardly noticable, algorithms in computational physics rely on exactly those language elements that are most taxing with regards to speed, such as loops. Here we will show you how to make python faster, often by magnitudes, while only making things marginally more complicated.

This is done using a module called cython whose name is a combination of "C" (from the computer language) and python. In a nutshell, it can help us speed up our programs if we are willing to give up a tiny bit of our laziness and letting python know a little more of what we are about to do. Cython can then translate parts of our code into plain C code that can be compiled and later used as its very own python module. Most of this is going on under the hood and you will never have to touch the C code yourself, so don't worry if you have no knowledge of the C language. Before theorizing too much, let us dive right in with an example that will clear up what we are about to do.

We will start with the ever popular *Hello World* program. The first step is to create a file `helloworld.pyx`. Note the extension `pyx` that is typically used for `cython` modules. Its content is simply

```
print "Hello World"
```

We now have to create a file that represents our python program. To avoid name conflicts, we have to give it a different name, for example `helloworld_cy.py`. Now we want to use the file `helloworld.pyx` as a module: First, we have to import another module called `pyximport` (part of the `cython` package) that will take care of the conversion to C code. Next, we import our custom module that will automatically print *Hello World* to the terminal:

```
import pyximport; pyximport.install()
import helloworld
```

This program was as simple as it gets but at least showed you the basic workflow for a `cython` implementation of your code. Let us move on to show how cython can actually speed up your program. We will use the integration of a simple function as an example, because it includes a loop that potentiall slows down our python program. This example is also used on the official webpage although we will make it a bit fancier. We start with a file called `integrate_python.py`:

```
def f(x):
    return x**2-x

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

We would now like to see how long it takes to execute this piece of code. For this purpose, there is another python module, called *timeit*. In brief, we import the module and an object called `Timer` which executes the function 100 times and measures the time it took in total. Do not worry if the keyword `lambda` is unfamiliar to you. It is simply another possibility for defining a function so that we can tell the `Timer` module to call it.

```
import integration_python
from timeit import Timer

t = Timer(lambda: integration_python.integrate_f(-5, 5, 100000))
print t.timeit(number=100)
```

Now let us check out the cython version. Create a file called `integration_cython.pyx` with almost the same content as its pythonic counterpart:

```
cdef f(double x):
    return x**2-x

cpdef integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Take a good look and note the following differences:

- Functions are defined using either `cdef` or `cpdef`. The difference between the two alternatives is that functions with `cpdef` can be called from a python file while `cdef` functions can be called only from other cython functions. Because `f(x)` is called only from the integration part, we declare it that way.
- Variables are now defined with a definite static type such as `int, double, ...` using `cdef`. In particular, it is important that the loop variable `i` is typed as an integer.
- The rest of the code stays exactly as it is!

Of course, we now want to see how much we have gained by adding just a few type definitions. We set up a corresponding python file for calling the integration routine, `integration_cy.py`:

```
import pyximport; pyximport.install()
import integration_cython
```

```
from timeit import Timer

t = Timer(lambda: integration_cython.integrate_f(-5, 5, 100000))
print t.timeit(number=100)
```

When comparing the output, you will see that the cython code runs about $8-10$ times faster!

Finally, we will show how numpy can be used in combination with cython. You will have to add a few additional lines to the import instructions: In your *.pyx file, you first import the standard numpy module. For numpy, cython also needs some additional information that tells it how to deal with numpy objects. This is achieved via a new command cimport:

```
import numpy as np
cimport numpy as np # new!
```

In order for the compilation to work, pyximport now has to know where to find the information about numpy:

```
import pyximport;
import numpy as np
pyximport.install(setup_args={'include_dirs': np.get_include()})
import my_cython_package
```

All that is left to do now is to actually create a numpy array that we can efficiently use in a cython code. Of course, this is done just like we introduced int and double types previously, namely using a cdef:

```
cdef int w = 10
cdef int h = 100
cdef np.ndarray int_array = np.zeros([w, h], dtype=np.int)
cdef np.ndarray double_array = np.zeros([w, h], dtype=np.double)
```

And that's it already!

Do not worry if this seems overwhelming at first, especially if you do not have much programming experience, yet. Using the recipes we provided here, you can easily speed up your code as long as you follow what we described here. In the next exercise, we give you a skeleton program based on which you can build your own version. It already contains the cythonized version of most variables you will need.

3. Make it or break it *10 points*

Percolation theory is concerned with how **clusters** form and behave. The word cluster, in the context of this exercise, describes a connected region on a randomly occupied lattice. Although seemingly a more academic exercise, it has many real-world applications where the lattice and its occupation structure represent such diverse things as the lattice of a solid and its magnetic moments or a forest and its possibly burning trees in case of a wildfire. As one varies the amount of occupied sites, the clusters undergo a phase transition where a large system-spanning cluster may appear instead of only smaller disconnected clusters. This so called **percolation transition** and is one of the simplest manifestations of a **continuous phase transition**.

In this exercise, we want to study this very phase transition on a square lattice with a random occupation. The central element is to find an efficient algorithm that allows the identification of clusters. In fact, this problem is also well known in the field of computer science, albeit in a different context. The so-called **union-find** algorithm can be used to compute equivalent classes of a set. The corresponding equivalence relation in the percolation problem is whether two occupied lattice sites are connected or not. In statistical physics, this algorithm goes by the name of **Hoshen-Kopelman**.

Let us now describe the algorithm in a bit more detail. We set up a square lattice and initially fill the squares with zeros. For each square of the lattice we pick a random number and mark it “1” with a probability p. Connected regions of 1s are then called a cluster.

The identification using the Hosh-Kopelman algorithm works as follows. We start in the upper left corner and scan the lattice row by row from left to right. To each site, we then associate an integer that serves the purpose of identifying the respective cluster. If the site to the left or on top is also occupied and therefore already carries a label, we associate the same label to the current site. If not, we increment the label by one and assign this new label to the site at hand. It might happen that we encounter a site where both neighbors, left and top, were already assigned some label but not the same. In that case, we choose the smaller one and save that the two seemingly clusters are actually one and the same. The possible processes are depicted in Fig. **??**.

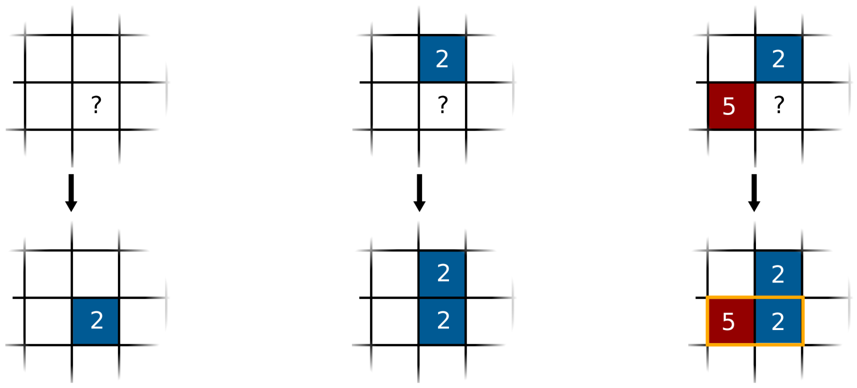

Figure 1: Rules for building clusters in the Hoshen-Kopelman algorithm: If an occupied site does not border any other occupied position yet, we assign a new cluster index (left). If only one site already carries a label, we simply assign the same one to the current site (middle). In the special case where two clusters border the same site, we use the smaller of the two labels and keep in mind that the clusters are actually connected (right).

Your task is to implement this algorithm and identify clusters that span the entire system. Spanning the system means that the same label index appears on opposite boundaries of the square lattice.

We prepared two skeleton programs for you, one is pure **python** and one is in **cython**.

After having implemented the Hoshen-Kopelman algorithm, calculate the number of clusters and the probability that a percolating cluster is formed. Depending on your implementation, a 64×64 lattice should allow you to obtain results in a reasonable amount of time. Plot your result as a function of the occupation probability p, introduced in the beginning of this exercise.

4. Contagious magnetism

alternative assignment – 10 points

In this *optional* exercise, we will treat a slightly more complex version of the percolation problem. For this assignment in particular, we invite you to play around with the assignment that we are going to describe in the following. You will be rewarded with insights into an interesting problem from statistical physics that naturally comes up when studying certain classes of problems in epidemics and ferromagnetism. The connection to the latter was the motivation for a recent research paper, in which the authors study ferromagnetism in a special kind of Hubbard model. Because of its origin, this specific percolation problem is also referred to as **Pauli percolation**.

An in-depth motivation and derivation of the entire model is be beyond the scope of this assignment, which is why are going to treat it simply as a peculiar statistical physics problem. Not only do we need our knowledge from percolation theory, but we will also employ **Monte Carlo** to tackle this problem.

Without further ado, let us jump straight in and consider the following problem: We start by setting up a so-called **Cayley Graph** of order $z = 3$. This is a special kind of graph, or in a more familiar language lattice, which is set up by starting with a root site to which $z = 3$ additional sites are connected. To each of these sites, two more sites are connected so that every site we added in the first step now has 3 neighbors. Continuing in this fashion, a graph is set up in which all sites except for those on the bounadary have 3 neighbors. A sample graph is depicted in Fig. (**??**).

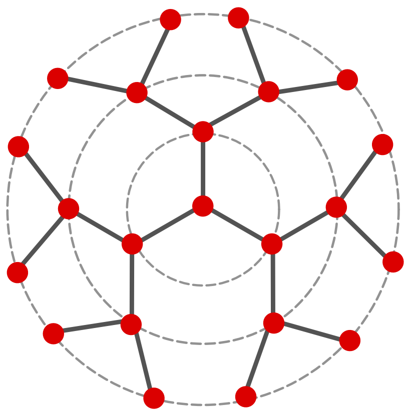

Figure 2: A Cayley graph of order $z = 3$. Starting from a root site in the middle, each site is connected to three neighboring sites.

The reason for choosing this particular graph is that it is very similar to the so called **Bethe lattice**. Many models turn out to be exactly solvable on the Bethe lattice, among them the one studied in this exercise. The only difference between the Bethe lattice and the Cayley graph is that the Bethe lattice is infinite and therefore has neither a root site nor a surface. For this particular simulation, it is not possible to find a formulation in the inifite system size limit and we thus have to work with the Cayley graph.

To formulate up the model, we start in the framework of the canonical ensemble where a fixed number of sites M are marked as occupied. Clusters are then defined as connected components of occupied sites on this graph. A particular graph that fulfills this constraint is denoted by $\mathcal{C}$

and is made up of clusters $\mathcal{C}_i$ whose size is $n(\mathcal{C}_i)$. By associating to each graph $\mathcal{C}$ a weight $W(\mathcal{C})$

$$W(\mathcal{C}) = \prod_i (n(\mathcal{C}_i) + 1). \tag{1}$$

we may define the partition as the sum over all possible graph realizations

$$Z = \sum_{\{\mathcal{C}\}} W(\mathcal{C}). \tag{2}$$

Our definition of a weight has the following interesting property: Imagine two clusters of size $n(\mathcal{C}_1) \equiv n_1$ and $n(\mathcal{C}_2) \equiv n_2$. Their contribution to the total weight is $(n_1+1)(n_2+1)$. If we now move them closer until they merge into one larger cluster, the contribution to the weight drops sharply to (n_1+n_2+1). Effectively, this leads to a repulsive interaction between clusters.

Motivated by the underlying physical problem, we actually want to work in the grand-canonical ensemble where the number of occupied sites may fluctuate. We introduce a parameter p that can be interpreted as something similar to a probability to occupy sites, although it will *not* be equal to the actual density of occupied sites – you will see why in a second. The grand-canonical partition sum is *defined* as the following:

$$Z = \sum_{\{\mathcal{C}\}} \left(\frac{p}{1-p}\right)^{n(\mathcal{C})} W(\mathcal{C}) \tag{3}$$

The first term may also be rewritten as $\exp(\mu n)$ to emphasize its effective role as a chemical potential that controls the particle number. Also, note that the sum now runs over all possible graphs, because we allow an arbitrary number of occupied and unoccupied sites, respectively.

We now want to study this model numerically. Because of the large number of possible graph realizations, it is not possible to enumerate them all in a reasonable amount of time. We thus turn to Monte Carlo to select only those that contribute most. Our goal is to study the same observables as before, namely the average cluster size and the possibility of a percolating cluster. A percolating cluster, in this case, is defined to be one that stretches from the root site to any of the boundary sites. In the Monte Carlo procedure, we start with a random graph. A Monte Carlo step then consists of choosing a random site and proposing to occupy it if it was previously unoccupied or to unoccupy it if it was occupied. To accept the step, the ratio of weights has to be calculated and accepted with Metropolis probability

$$p < \min\left(1.0, \frac{W_{\text{new}}}{W_{\text{old}}}\right).$$

The simplest way to calculate this ratio is to recalculate the entire cluster structure for the proposed move which in itself is not a trivial calculation because of the graph structure. It is in principle also possible to track directly whether two clusters will combine or dissociate when a site is occupied or unoccupied, respectively. This is, however, more involved and will not necessarily be much faster, especially in the limit of large clusters. When performing calculations, you can try graphs of up to 500 sites and perform 10^6 Monte Carlo steps without measuring and then again 10^6 steps with measurements. We would like to stress again that this assignment is somewhat more challenging than the others and should be viewed as an opportunity to make contact with current research. If you have any questions, please get in contact with us!