Institute for Theoretical Physics
University of Cologne

Prof. Dr. Simon Trebst
Peter Bröcker, Johannes Helmes

Computational Many-Body Physics

Assignment 0

Summer Term 2015

website: http://www.thp.uni-koeln.de/trebst/Lectures/2015-CompManyBody.shtml
due date: Wednesday, April 15th

The upcoming homework assigments will guide you through small projects implementing and applying some of the numerical methods discussed in the lecture. Light programming experience in python, C++ or some other programming language will be needed to do these exercises. While we will leave it up to you which programming language you will use, we somewhat encourage the use of python – which beyond its general ease of use allows to very efficiently plot numerical data in various ways.

This zeroths exercise is intended to make you familiar with the basics of python and to introduce some of the most important features that will be needed throughout the course.

1. Python warm-up

Python is a **scripting** language. This means that individual commands of the source code are **interpreted** line-by-line (instead of compiled as a whole). The python interpreter directly checks and executes a given command before reading the next command. It is therefore possible to use python from a system's command line by calling `python` and entering commands piece-by-piece. An example for a linux machine might looks like this (the user input is marked **bold**):

```
user@machine:~$ python
Python 2.7.3 (default, Feb 27 2014, 20:00:17)
>>> name="Eve"
>>> print "Hello",name
Hello Eve
>>> a=12
>>> print "The square of",a,"is",a**2
The square of 12 is 144
```

Basic commands

You might have noticed python's basic semantics for the definition of variables and printing in the above example. In general we will however use source files. They can be run by typing `python basics.py` on the command line. The first code example contains several important structures of python. Read it carefully and find out what tasks are performed by the script.

```
#first routine
n=10
prod=1
for i in range(1,n):
```

```
  prod*=i
print prod

#second routine
numbers=[36,2,9,45,12,30,4,29]
a=numbers[0]
b=numbers[0]
for i in range(1,len(numbers)):
  if numbers[i]<a:
    a=numbers[i]
  elif numbers[i]>b:
    b=numbers[i]
print a,b

#third routine
def mysterious_function(p,q):
  a,b=(p,q) if p>=q else (q,p)
  while b!=0:
    a,b=(b,a%b)
  return a

print mysterious_function(12,18)
print mysterious_function(40,20)
print mysterious_function(7,6)
```

These examples demonstrate the easy readability of python code and introduced the following features (check!):

- for loop
- in-place arithmetic operation (*=)
- if-else statement
- while loop
- definition of functions with parameters and return value
- list
- unpacking of lists (what is this?)
- comments in the source code

When you get accustomed to python you will appreciate that it offers extremely extended capabilities in handling **lists**. Single commands can *join, slice, sum over, iterate, zip, etc.* n-dimensional lists. If you are interested in these, you may check out `https://docs.python.org/2/tutorial/introduction.html#lists`.

NumPy – a very important library

A standard installation of python offers general purpose functions such as `min()`, `max()`. Beyond those additional **libraries** of more specialized functions can be installed and **imported** into the working environment, e.g. usually on the top of a script. Some examples are:

- **NumPy** – for numerical techniques
- **SciPy** – a collection of functions used in scientific applications

- **Matplotlib** – the most widely used plotting library
- **pyMPI** – for doing parallel computing

Obviously, the NumPy library will be of major importance for our exercises such that you can import it per default in the first line of any of your scripts:

```
import numpy as np
```

Mathematical constants and functions like `np.pi, np.cos(), np.log(), np.sqrt(), np.floor(), ...` as well as random number generation and a more advanced type of lists – so-called numpy arrays. The following example demonstrates theses capabilities. We want to implement the **Box-Muller method**, which generates Gaussian distributed random numbers from a stream of uniformly distributed random numbers. Carefully study the code skeleton which you can retrieve here and add the three commands which are missing at the indicated lines.

```
import numpy as np

def boxmuller(u1,u2):
  #TODO: uncomment the following lines and type in the formulas of the Box
  #Muller transformation using np.log, np.cos, np.sin, np.sqrt, np.pi
  #z1=
  #z2=
  return z1,z2

N=100000
random_numbers=np.zeros(N)
for i in range(N):
  random_numbers[i]=np.random.random()

gaussian_rn1,gaussian_rn2=boxmuller(random_numbers[::2],random_numbers[1::2])

#TODO: Check that the variances of the uniformly and the gaussian distributed
#random numbers are as expected, i.e. 1/12=0.833 and 1.0
#uniform_var=np.var(random_numbers)
#gaussian_var1=
#gaussian_var2=
print uniform_var, gaussian_var1, gaussian_var2

#TODO: As an evidence for the claim, that gaussian_rn1/2 are gaussian
#distributed with sigma=1, calculate the first six moments of the generated
#random numbers
moments=np.zeros(7)
for rn in gaussian_rn1:
  #moments[1]+=rn**1
  #moments[2]+=
  #...

moments/=(N/2)
print moments
```

Plotting

Besides the actual computation and simulation of a physical problem the presentation of numerical results – oftentimes in large data sets – is also an important issue. The aforementioned library **Matplotlib** is a very powerful tool to accomplish precisely this task. Let us briefly introduce it via the following code snippet. First we plot a simple function on a two-dimensional canvas.

```
import matplotlib.pylab as plt
import numpy as np

plt.figure(figsize=(10,6))

x_int = range(10)
x_values = np.linspace(0, 9, 1000)
#creates a numpy array with 1000 equally-spaced values between 0 and 9
squares=[i**2 for i in x_int]
y_values = x_values**2
plt.plot(x_int, squares,'o', label="Squares" )
plt.plot(x_values, y_values, label="f(x)=x*x")

plt.xlabel("x");
plt.ylabel("y");
plt.legend(loc="upper left")

plt.savefig("squares.pdf")
plt.show()
```

The instructions are probably self-explanatory. As you might see, one needs to pass the two lists or arrays (of equal size!) to the `plt.plot()` command.

A nice demonstration of three-dimensional plotting is given by the following lines which you should **append** to the boxmuller.py example. Does it look like a Gaussian bell?

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pylab as plt

#The following lines are technical tricks to create the histogram
rounded_gauss1=np.around(gaussian_rn1,decimals=1)
rounded_gauss2=np.around(gaussian_rn2,decimals=1)
gaussian_points=(zip(rounded_gauss1.tolist(),rounded_gauss2.tolist()))
X=np.around(np.arange(-3.0,3.0,0.1),decimals=1)
Y=np.around(np.arange(-3.0,3.0,0.1),decimals=1)


Histogram=np.zeros((len(X),len(Y)))
for i,x in enumerate(X): #what does enumerate do?
  for j,y in enumerate(Y):
    Histogram[i][j]=gaussian_points.count((x,y))

X,Y = np.meshgrid(X,Y) #what is this?
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Histogram, cmap='coolwarm', linewidth=0,
```

```
        antialiased=True, vmin=0, vmax=300, rstride=1, cstride=1)
    plt.show()
```

There are many more plotting styles for three dimensional plotting. Feel free to browse a small collection of Matplotlib's plotting features:
http://matplotlib.org/1.4.3/users/screenshots.html.

A first application – Central limit theorem

Let us exercise our new python skills by showing an example of the central limit theorem. It states that the arithmetic mean of a sufficiently large number of arbitrary random variables is a normal distribution. The distribution we start from is depicted in Fig. 1. We will only use this distribution and take the arithmetic mean of, say, $N = 100$ random variables.

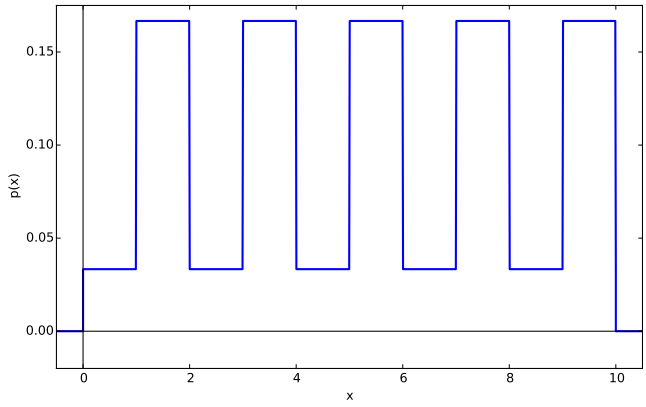

Figure 1: Comb distribution: The high bars are five times as high as the low bars and $p(x) = 0$ anywhere else than $0 \leq x \leq 10$

Instructions:

1. Find a mathematical function describing the comb distribution and verify it by producing a plot similar to Fig. 1
2. Write a function that produces random numbers following this distribution. You could use the rejection-method.
3. Fix a replication number N and determine the arithmetic mean as a set of sampled random numbers (each random number is the arithmetic mean of N random numbers sampled according to the comb distribution).
4. Verify that the mean is normally distributed by determining the first moments as above. How large does N need to be?

5*. Plot a histogram of the arithmetic mean in order to see whether it is Gaussian.