Submission Phase

1. Do assignment ☑ (/smac-001/human_grading/view/courses/971628/assessments/11/submissions)

Evaluation Phase

2. Evaluate peers ☑ (/smac-001/human_grading/view/courses/971628/assessments/11/peerGradingSets)

Results Phase

3. See results ☑ (/smac-001/human_grading/view/courses/971628/assessments/11/results/mine)

Your effective grade is 18

Your unadjusted grade is 18, which is simply the grade you received from your peers.

See below for details.

In this homework session 7 of Statistical Mechanics: Algorithms and Computations, we study Bose-Einstein condensation in a three-dimensional harmonic trap. After a start with two preparation programs, you then consider the program markov_harmonic_bosons.py, where you will obtain the distribution of x-positions. You then consider the anharmonic trap.

NB: **This session is much shorter** than the previous sessions, because of the delay incurred. **We hope it is not less interesting**. Thousands of academic research papers are written every year on the subject of Bose-Einstein condensation in harmonic three-dimensional traps.

NNB: To gain time, we eliminated the announced part corresponding to the boson-bunching.

Summary: In this section we provide **two easy Preparation programs** about histograms.

NOTE ADDED (03/22/2014): There was a minor inprecision in the titles of the histograms
Preparation program 1:

```
import random, pylab

data = []
for run in range(100000):
    data.append(random.uniform(0.0, 1.0))
pylab.title('Preparation program 1, SMAC week 7, 2014')
pylab.hist(data, bins=200, normed=True)
pylab.xlim(0.5,0.6)
pylab.xlabel('$x$')
pylab.ylabel('$\\pi(x)$')
pylab.show()
```

Preparation program 2:

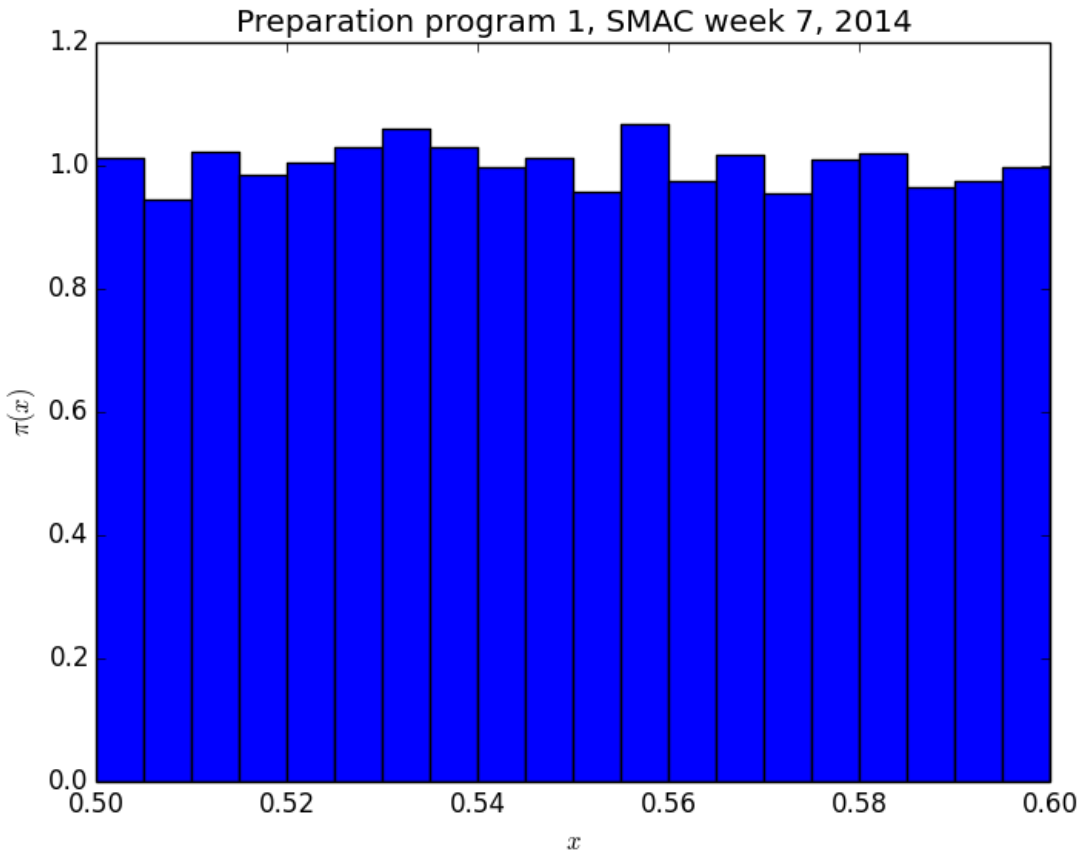
```
import random, pylab

data = []
for run in range(100000):
    data.append(random.uniform(0.0, 1.0))
pylab.title('Preparation program 2, SMAC week 7, 2014')
pylab.hist(data, bins=200, range=[0.5, 0.6], normed=True)
pylab.xlabel('$x$')
pylab.ylabel('$\\pi(x)$')
pylab.show()
```

A1: Download (cut-and-paste) both programs, then run them. Explain the essential difference between them. **THIS IS A LESSON FOR LIFE, NEVER FORGET IT.**

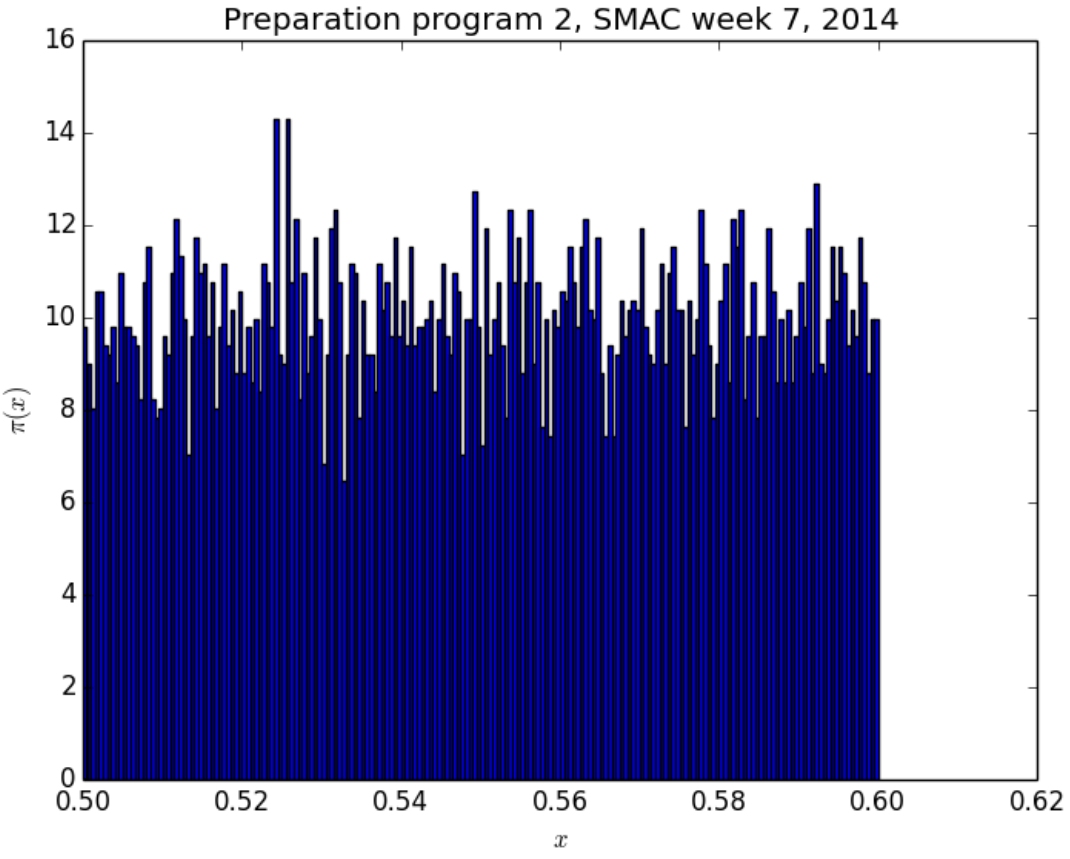
1. What is the distribution of "data" in **Preparation program 1**? What is put into the histogram? What is the **distribution shown**?
2. What is the distribution of "data" in **Preparation program 2**? What is put into the histogram? What is the **distribution shown**?

The importance difference is the use of **xlim** in program 1 and use of **range** in program 2.

1. The data is uniformly distributed from 0 to 1. A normalized histogram is generated. Next, only a certain segment of the histogram is plotted out (for $0.5 < x < 0.6$). This is the correct histogram as Pi(x) is similar for all x and between 0 and 1.



2. The data is uniformly distributed from 0 to 1. A normalized histogram is generated using only x ranging from 0.5 to 0.6 . Since, the normalization is done only for a part of the distribution (for x ranging from 0.5 to 0.6), this leads to incorrect normalization. Next, the incorrect histogram is plotted. Note that the Pi(x) values are more than 1, which indicates the incorrect normalization.



Evaluation/feedback on the above work

Note: this section can only be filled out during the evaluation phase.

Here you have to evaluate your fellow student's correct understanding of when to normalize a histogram.
There are two issues:
In **Preparation Program 1**, points are sampled in the range [0, 1], and the histogram is created there.The plotting range that is afterwards chosen to be between 0.5 and 0.6. The histogram is normalized between [0, 1].
In **Preparation Program 2**, the situation is different: All the points outside the interval [0.5, 0.6] are treated as outliers, and they are not comprised in the histogram. The bins are taken between 0.5 and 0.6, the normalization is done between 0.5 and 0.6.

Here is a solution that would yield full score:

1/ The distribution of data are uniform random numbers in [0,1]. The histogram between 0 and 1 is shown between 0.5 and 0.6.
2/ The distribution of data are uniform random numbers in [0,1]. Data between 0.5 and 0.6 are put into the histogram, between 0.5 and 0.6.

POINTS - DESCRIPTION
Give one point if one issue was understood.
Give two points if both issues were understood.

Score from your peers: **2**

peer 1 → *[This area was left blank by the evaluator.]*

peer 2 → *[This area was left blank by the evaluator.]*

peer 3 → *[This area was left blank by the evaluator.]*

B In this section, you will set up a Monte Carlo programs for bosons in a three-dimensional harmonic trap.

Download (copy-and-paste) the below program markov_harmonic_bosons.py,

```
import random, math, pylab

def levy_harmonic_path(k, beta):
    xk = tuple([random.gauss(0.0, 1.0 / math.sqrt(2.0 *
                math.tanh(k * beta / 2.0))) for d in range(3)])
    x = [xk]
    for j in range(1, k):
        Upsilon_1 = 1.0 / math.tanh(beta) + 1.0 / \
                          math.tanh((k - j) * beta)
        Upsilon_2 = [x[j - 1][d] / math.sinh(beta) + xk[d] /
                     math.sinh((k - j) * beta) for d in range(3)]
        x_mean = [Upsilon_2[d] / Upsilon_1 for d in range(3)]
        sigma = 1.0 / math.sqrt(Upsilon_1)
        dummy = [random.gauss(x_mean[d], sigma) for d in range(3)]
        x.append(tuple(dummy))
    return x

def rho_harm(x, xp, beta):
    Upsilon_1 = sum((x[d] + xp[d]) ** 2 / 4.0 *
                    math.tanh(beta / 2.0) for d in range(3))
    Upsilon_2 = sum((x[d] - xp[d]) ** 2 / 4.0 /
                    math.tanh(beta / 2.0) for d in range(3))
    return math.exp(- Upsilon_1 - Upsilon_2)

N = 512
T_star = 0.8
beta = 1.0 / (T_star * N ** (1.0 / 3.0))
nsteps = 100000
positions = {}
for k in range(N):
    a = levy_harmonic_path(1, beta)
    positions[a[0]] = a[0]
for step in range(nsteps):
    boson_a = random.choice(positions.keys())
    perm_cycle = []
    while True:
        perm_cycle.append(boson_a)
        boson_b = positions.pop(boson_a)
        if boson_b == perm_cycle[0]: break
        else: boson_a = boson_b
    k = len(perm_cycle)
    perm_cycle = levy_harmonic_path(k, beta)
    positions[perm_cycle[-1]] = perm_cycle[0]
    for k in range(len(perm_cycle) - 1):
        positions[perm_cycle[k]] = perm_cycle[k + 1]
    a_1 = random.choice(positions.keys())
    b_1 = positions.pop(a_1)
    a_2 = random.choice(positions.keys())
    b_2 = positions.pop(a_2)
    weight_new = rho_harm(a_1, b_2, beta) * rho_harm(a_2, b_1, beta)
    weight_old = rho_harm(a_1, b_1, beta) * rho_harm(a_2, b_2, beta)
```

```
    if random.uniform(0.0, 1.0) < weight_new / weight_old:
        positions[a_1] = b_2
        positions[a_2] = b_1
    else:
        positions[a_1] = b_1
        positions[a_2] = b_2
```

B1 Now modify the above program (markov_harmonic_bosons.py) so that it reads from and writes to file the keys and values of the dictionary positions. This
is analogous to what was done in homework session 3. Here is the snippet that **reads the dictionary "positions"**....

```
positions = {}
filename = 'boson_configuration.txt'
positions = {}
if os.path.isfile(filename):
    f = open(filename, 'r')
    for line in f:
        a = line.split()
        positions[tuple([float(a[0]), float(a[1]), float(a[2])])] = \
               tuple([float(a[3]), float(a[4]), float(a[5])])
    f.close()
    if len(positions) != N: exit('error input file')
    print 'starting from file', filename
else:
    for k in range(N):
        a = levy_harmonic_path(1, beta)
        positions[a[0]] = a[0]
    print 'starting from scratch', filename
```

On line 8 of this snippet, you find the line "positions[tu... \". **Explain exactly** what this line does (what is "a", exactly, "positions" , why is there a "tuple", what is the "\" good for. Also, explain the line immediately following "tuple([float(a[3]), ...". Note that **homework session 5 already contained similar questions**...

Here is the snippet for writing to file, at the end of the simulation run, so that it can be read into the next iteration:

```
f = open(filename, 'w')
for a in positions:
   b = positions[a]
   f.write(str(a[0]) + ' ' + str(a[1]) + ' ' + str(a[2]) + ' ' +
           str(b[0]) + ' ' + str(b[1]) + ' ' + str(b[2]) + '\n')
f.close()
```

Run this program several times to make sure that it runs smoothly.

The line read from the file "boson_configuration.txt" is split and stored into list **a**. Each line contain six entries and therefore, list **a** has six elements.

1. positions is the dictionary which stores the positions of the bosons.
2. tuple is similar to a list but it's immutable (unchangeable) once created. See below for the exact reason of using a tuple in this piece of code.
**3. ** allows one to extend the code to the second line and helps in improving readability of the code.

Code explanation:

A. positions[tuple([float(a[0]), float(a[1]), float(a[2])])] = tuple([float(a[3]), float(a[4]), float(a[5])])
The above line creates two tuples (one using the first three elements and another using the last three elements of list a respectively). Next, it uses the first tuple as key and second tuple as the corresponding value in the dictionary positions. *So, in essence, for each entry in the dictionary "positions", the key is the position of a boson and the value is the position of the next boson in the cycle. By iterating from key to value, we cycle across the bosons.*

B. f.close().
f.close() closes the file referenced by file handle f i.e. 'boson_configuration.txt'.

Evaluation/feedback on the above work

Note: this section can only be filled out during the evaluation phase.

In Section **B1**, you are asked to evaluate that your fellow student correctly understands how to read to and write from a file. Furthermore, he/she should understand that "positions" is a dictionary, that "a" is a list of strings, that these strings are translated into floats, and then into lists of floats. The "tuple" is used because keys of a dictionary can only be immutable, that "\" denotes a continuation over a lign, Finally, that the tuple is the 'tuple([float(a[3]), ..." denotes the value of a dictionary.
There are **three issues**, and please note that we only ask for "approximate" understanding of Python issues. Nevertheless, we would ask for some details

Here is a minimal solution that would yield full score:

1/ **"positions" is a dictionary**, and **"a" is a list of strings**, "\" indicates a continuation line.
2/ The **"tuple"** is used because keys of a dictionary can only be **immutable.**
3/ tuple([float(a[3]), ..." denotes the value of a dictionary.

POINTS - DESCRIPTION

Give 0 to three points according to how many of these issues are treated correctly (taking again into account that some degree of approximation is OK)

Score from your peers: **3**

peer 1 → *[This area was left blank by the evaluator.]*

peer 2 → *[This area was left blank by the evaluator.]*

peer 3 → Ignoring the 3rd part where fclose is mentioned. Good enough.

B2 In this program, for N=512, and T_star = 0.8, **produce two histograms and one plot, all on the same picture**.

1. The **first histogram** should give the normed distribution of x-position of particles (**normed over all x**), and show it between x = -3 and x = 3 (Remember the **Preparation programs**).
2. The **second histogram** should produce the normed distribution of x-position of particles that are on cycles of length larger than cycle_min.

ATTENTION: **Note that you need two lists of data, one for each histogram**. **For the second histogram**, DO not add **all** the elements of the cycle perm_cycle to the data list, **add only** the x coordinate of the **FIRST ELEMENT**!!!! **HINT:** Two lines, such as...

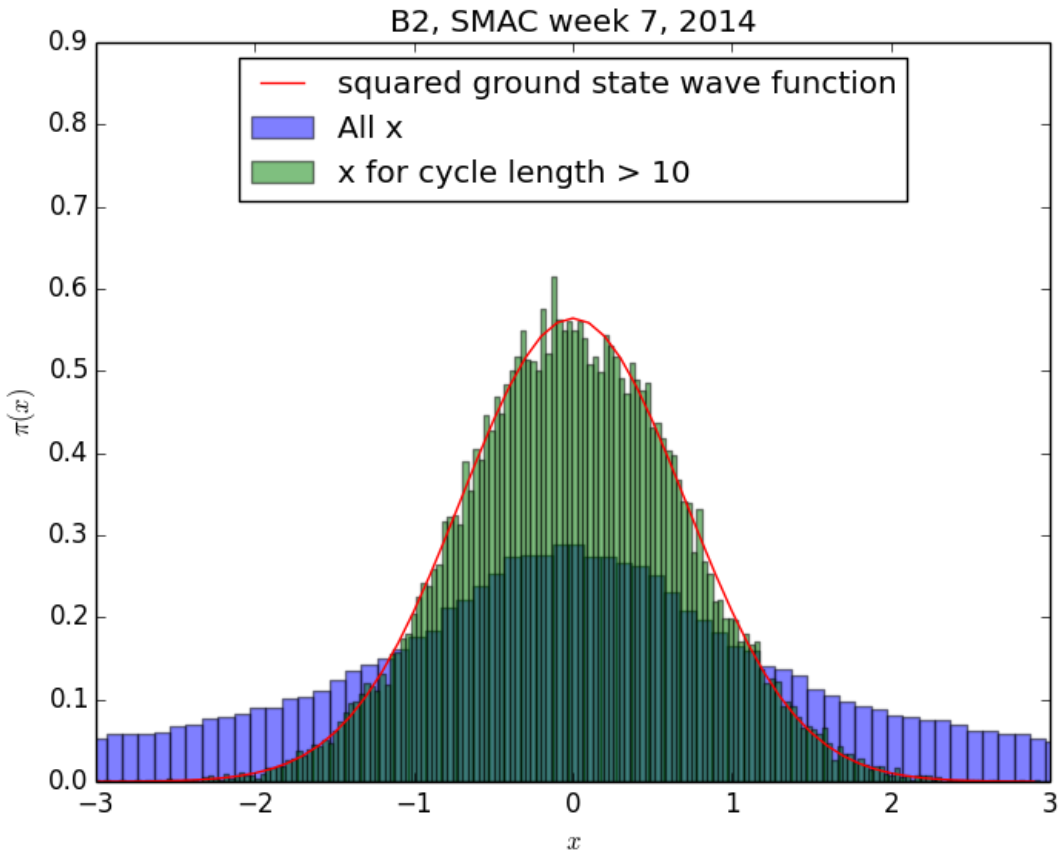
```
if k > cycle_min:
    data_long.append(perm_cycle[0][0])
```

...is all that it takes - this corresponds to 1/ pick a particle 2/ check its cycle length 3/ if k > cycle_min: add to data_long).
Finally, **compare** the **first** and the **second histogram** to the **squared ground-state wave function** of the harmonic oscillator, psi^2 (x) = exp(-x^2)/sqrt(pi). **Upload your results** for cycle_min = 10 **in one graphics file**, **upload your program**. **Explain why** the two histograms give different results, **explain why** one of the histograms can be compared to the squared ground state wave function, but only if cycle_min is sufficiently large.

NB: Make sure you use what you learned in **Preparation programs 1 and 2** to restrict the range of x values displayed in your histogram to the range -3<x<3.

NNB: **Use labels and legends to distinguish the three data sets**, use the **"alpha" keyword in pylab.hist** to make the histograms semi-transparent (alpha = 0.5).

Graphic file:



Explanations:

1. The two histograms (all x [light violet] and x with cycle length > 10 [green]) give different results. The one with long permutation cycles (green) is more peaked and has lower sigma (standard deviation). As explained in the lecture 7 towards the end, the particles with long permutation cycle are at large β i.e. low temperature. At this low temperature, Bose-Einstein condensation starts to take place. Due to this, the particles get more compact leading to narrow distribution. The other histogram (light violet) contains particles with smaller permutation cycles i.e. higher temperatures in addition to those with larger permutation cycles. Due to this, those particles are more scattered leading to widening of the distribution.

2. The histogram of x with longer permutation cycle length (> 10) can be compared to squared ground state wave function as shown in the above plot. They show a good agreement.

As explained in the lecture 7 towards the end, the particles with long permutation cycle are at large β i.e. low temperature. If the permutation cycles are long enough, the temperature will become low enough to reach the ground state. Therefore, the distribution will resemble that of the squared ground state wave function.

Program code:

```
import random, math, pylab, os

def levy_harmonic_path(k, beta):
    xk = tuple([random.gauss(0.0, 1.0 / math.sqrt(2.0 *
                math.tanh(k * beta / 2.0))) for d in range(3)])
    x = [xk]
    for j in range(1, k):
        Upsilon_1 = 1.0 / math.tanh(beta) + 1.0 / \
                            math.tanh((k - j) * beta)
        Upsilon_2 = [x[j - 1][d] / math.sinh(beta) + xk[d] /
                     math.sinh((k - j) * beta) for d in range(3)]
        x_mean = [Upsilon_2[d] / Upsilon_1 for d in range(3)]
        sigma = 1.0 / math.sqrt(Upsilon_1)
        dummy = [random.gauss(x_mean[d], sigma) for d in range(3)]
        x.append(tuple(dummy))
    return x

def rho_harm(x, xp, beta):
    Upsilon_1 = sum((x[d] + xp[d]) ** 2 / 4.0 *
                    math.tanh(beta / 2.0) for d in range(3))
    Upsilon_2 = sum((x[d] - xp[d]) ** 2 / 4.0 /
                    math.tanh(beta / 2.0) for d in range(3))
    return math.exp(- Upsilon_1 - Upsilon_2)

N = 512
T_star = 0.8
beta = 1.0 / (T_star * N ** (1.0 / 3.0))
nsteps = 100000

filename = 'boson_configuration.txt'
positions = {}
if os.path.isfile(filename):
    f = open(filename, 'r')
    for line in f:
        a = line.split()
        positions[tuple([float(a[0]), float(a[1]), float(a[2])])] = tuple([float(a[3]),
 float(a[4]), float(a[5])])
    f.close()

    if len(positions) != N:
        exit('error input file')
    print 'starting from file', filename
else:
    for k in range(N):
        a = levy_harmonic_path(1, beta)
        positions[a[0]] = a[0]
    print 'starting from scratch', filename

x_data_long = []
x_data_all = []
cycle_min = 10
```

```
for step in range(nsteps):
    boson_a = random.choice(positions.keys())
    perm_cycle = []
    while True:
        perm_cycle.append(boson_a)
        boson_b = positions.pop(boson_a)
        if boson_b == perm_cycle[0]:
            break
        else:
            boson_a = boson_b
    k = len(perm_cycle)
    perm_cycle = levy_harmonic_path(k, beta)
    positions[perm_cycle[-1]] = perm_cycle[0]
    for k in range(len(perm_cycle) - 1):
        positions[perm_cycle[k]] = perm_cycle[k + 1]
    a_1 = random.choice(positions.keys())
    b_1 = positions.pop(a_1)
    a_2 = random.choice(positions.keys())
    b_2 = positions.pop(a_2)
    weight_new = rho_harm(a_1, b_2, beta) * rho_harm(a_2, b_1, beta)
    weight_old = rho_harm(a_1, b_1, beta) * rho_harm(a_2, b_2, beta)
    if random.uniform(0.0, 1.0) < weight_new / weight_old:
        positions[a_1] = b_2
        positions[a_2] = b_1
    else:
        positions[a_1] = b_1
        positions[a_2] = b_2

    if len(perm_cycle) > cycle_min:
        x_data_long.append(perm_cycle[0][0])

    x_data_all.append(perm_cycle[0][0])

f = open(filename, 'w')
for a in positions:
   b = positions[a]
   f.write(str(a[0]) + ' ' + str(a[1]) + ' ' + str(a[2]) + ' ' +  str(b[0]) + ' ' + str
(b[1]) + ' ' + str(b[2]) + '\n')
f.close()

x_values = [0.1 * a for a in range (-30,30)]
y_values = [( math.exp( - xx **2) / math.sqrt(math.pi) ) for xx in x_values]

pylab.title('B2, SMAC week 7, 2014')
pylab.hist(x_data_all, bins=200, normed=True, alpha=0.5, label='All x')
pylab.hist(x_data_long, bins=200, normed=True, alpha=0.5, label='x for cycle length > 1
0')
pylab.plot(x_values, y_values, label = 'squared ground state wave function')

pylab.xlim(-3.0, 3.0)
pylab.xlim(0, 0.9)
```

```
pylab.xlabel('$x$')
pylab.ylabel('$\\pi(x)$')
pylab.legend(loc='upper center')
pylab.savefig('B2.png')
pylab.close()
```

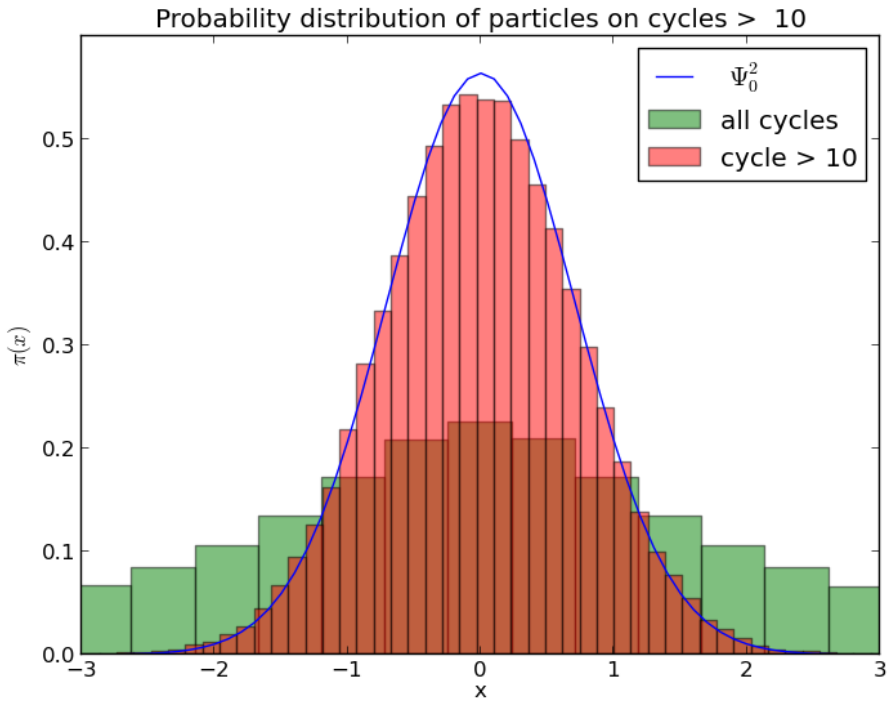
Evaluation/feedback on the above work

Note: this section can only be filled out during the evaluation phase.

B2 Here, you are asked to evaluate your fellow student's ability to extract a histogram of x-values from markov_harmonic_bosons.py. You must also evaluate whether your fellow students can use the many hints given to correctly compute the histogram of x-values for particles on long permutation cycles. Finally, the comparison with the ground state wave function should be successful.
There are five issues:
1/ Plot with correct distribution of all particles.
2/ Plot with correct distribution of particles on long cycles.
3/ Plot of groundstate wave function.
4/ Program that should normally have two lists of data.
5/ Consistent explanation why the "long cycles" live in the groundstate.

Here is a solution that would yield full score:

Here is the graphics file that was asked for: (Peer-evaluator: the plots may be noisier than here, if the simulations were run for less time. This is OK.)



Here's the program:

```
N = 512
T_star = 0.8
beta = 1.0 / (T_star * N ** (1.0 / 3.0))
nsteps = 1000000
cycle_min = 10
positions = {}
data = []
data_long = []
filename = 'boson_configuration.txt'
positions = {}
if os.path.isfile(filename):
    f = open(filename, 'r')
    for line in f:
        a = line.split()
        positions[tuple([float(a[0]), float(a[1]), float(a[2])])] = tupl
e([float(a[3]), float(a[4]), float(a[5])])
    f.close()
    if len(positions) != N: exit('error input file')
    print 'starting from file', filename
else:
    for k in range(N):
        a = levy_harmonic_path(1, beta)
        positions[a[0]] = a[0]
for step in range(nsteps):
    boson_a = random.choice(positions.keys())
    perm_cycle = []
    while True:
        perm_cycle.append(boson_a)
        boson_b = positions.pop(boson_a)
        if boson_b == perm_cycle[0]: break
        else: boson_a = boson_b
    data.append(perm_cycle[0][0])
    k = len(perm_cycle)
    if k > cycle_min:
        data_long.append(perm_cycle[0][0])
    perm_cycle = levy_harmonic_path(k, beta)
    positions[perm_cycle[-1]] = perm_cycle[0]
    for k in range(len(perm_cycle) - 1):
        positions[perm_cycle[k]] = perm_cycle[k + 1]
    a_1 = random.choice(positions.keys())
    b_1 = positions.pop(a_1)
    a_2 = random.choice(positions.keys())
    b_2 = positions.pop(a_2)
    weight_new = rho_harm(a_1, b_2, beta) * rho_harm(a_2, b_1, beta)
    weight_old = rho_harm(a_1, b_1, beta) * rho_harm(a_2, b_2, beta)
    if random.uniform(0.0, 1.0) < weight_new / weight_old:
        positions[a_1] = b_2
        positions[a_2] = b_1
    else:
        positions[a_1] = b_1
        positions[a_2] = b_2
```

```
x_values = [0.1 * a for a in range (-30,30)]
y_values = [1.0 / math.sqrt(math.pi) * \
                  math.exp( - xx **2 ) for xx in x_values]
pylab.plot(x_values, y_values, label=' $\\Psi_0^2$')

pylab.title('Probability distribution of particles on cycles >  ' + str(
cycle_min))
pylab.xlabel('x')
pylab.ylabel('$\\pi(x)$')
pylab.hist(data, bins=50, normed=True, label='all cycles', alpha=0.5)
pylab.hist(data_long, bins=50, normed=True, label='cycle > ' + str(cycle
_min), alpha=0.5)
pylab.axis([-3.0, 3.0, 0.0, 0.6])
pylab.legend()
pylab.show()
f = open(filename, 'w')
for a in positions:
   b = positions[a]
   f.write(str(a[0]) + ' ' + str(a[1]) + ' ' + str(a[2]) + ' ' +  str(b[
0]) + ' ' + str(b[1]) + ' ' + str(b[2]) + '\n')
f.close
```

Here's the explanation:

The histograms are different: The first histogram gives the x-distribution of all particles, and the second those particles on long permutation cycles. These particles "live" effectively at inverse temperature k beta with k > cycle_length. At low temperatures, these particles are in the groundstate [not required: if 1/(T * cycle_length) << 1].

Evaluation
POINTS - DESCRIPTION

1/ Plot with correct distribution of all particles.
2/ Plot with correct distribution of particles on long cycles.
3/ Plot of groundstate wave function.
4/ Program that should normally have two lists of data.
5/ Consistent explanation why the "long cycles" live in the groundstate.

Note that this is great and difficult stuff.

Score from your peers: **5**

peer 1 → *[This area was left blank by the evaluator.]*

peer 2 → *[This area was left blank by the evaluator.]*

peer 3 → *[This area was left blank by the evaluator.]*

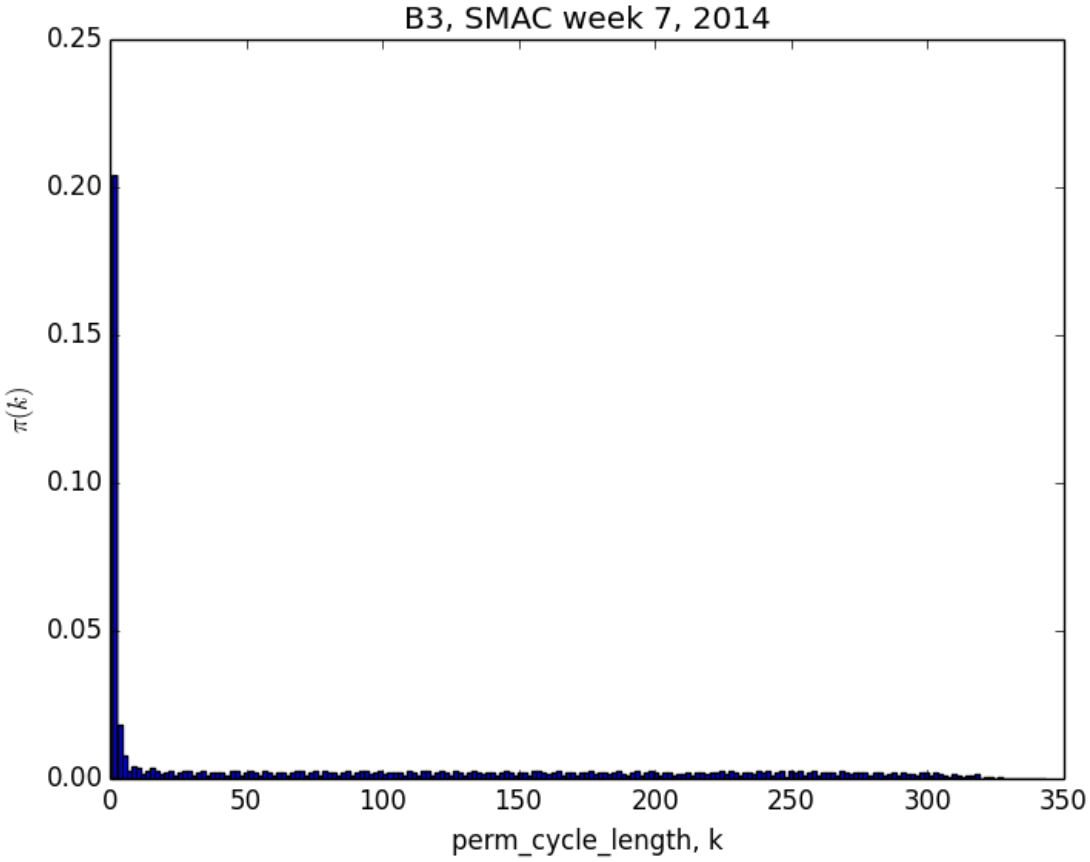
B3 In the program of **B2**, generate the **probability for a particle to be in a cycle of length k**. This is simply the histogram of 'perm_cycle'. Plot this distribution at temperature T* = 0.6, N= 512. Use pylab.ylim(...) to zoom in on the distribution for 0< y < 0.01. **Can you confirm** that the probability distribution for a particle to be on a cycle of length k is independent of k, in a wide range of k values? **Explain what you did to obtain that histogram**, and **upload the histogram**.

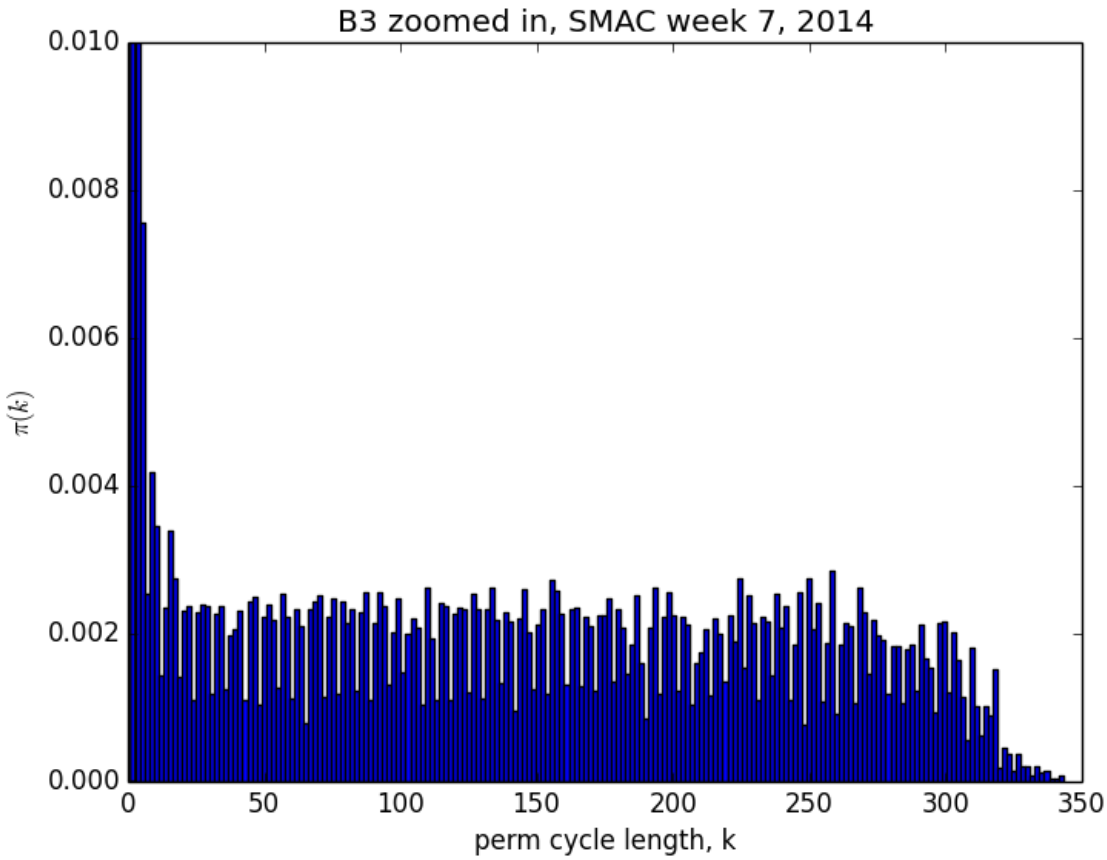
NB: Notice that, after debugging the program, you should run it several times to get a good initial start. Then let it run for 15 minutes (if you can) to produce final output. Treat yourself to a large ice-cream while waiting.

Steps taken to create the histogram:

1. Define a empty list (k_data) to append data to.
2. Append length of perm_cycle to the list k_data every cycle.
3. After the runs are finished, use pyplot to generate the histogram.
4. Next use, pylab.ylim(0,0.01) to create the zoomed in version.

The two histograms (full and zoomed-in versions) are below. The zoomed-in version of the histogram clearly shows that the probability distribution for a particle to be on a cycle of length k is independent of k for a wide range of k values (up to 300 or so in the given case).





The program code is as below:

```
import random, math, pylab, os

def levy_harmonic_path(k, beta):
    xk = tuple([random.gauss(0.0, 1.0 / math.sqrt(2.0 *
                math.tanh(k * beta / 2.0))) for d in range(3)])
    x = [xk]
    for j in range(1, k):
        Upsilon_1 = 1.0 / math.tanh(beta) + 1.0 / \
                            math.tanh((k - j) * beta)
        Upsilon_2 = [x[j - 1][d] / math.sinh(beta) + xk[d] /
                     math.sinh((k - j) * beta) for d in range(3)]
        x_mean = [Upsilon_2[d] / Upsilon_1 for d in range(3)]
        sigma = 1.0 / math.sqrt(Upsilon_1)
        dummy = [random.gauss(x_mean[d], sigma) for d in range(3)]
        x.append(tuple(dummy))
    return x

def rho_harm(x, xp, beta):
    Upsilon_1 = sum((x[d] + xp[d]) ** 2 / 4.0 *
                    math.tanh(beta / 2.0) for d in range(3))
    Upsilon_2 = sum((x[d] - xp[d]) ** 2 / 4.0 /
                    math.tanh(beta / 2.0) for d in range(3))
    return math.exp(- Upsilon_1 - Upsilon_2)

N = 512
T_star = 0.6
beta = 1.0 / (T_star * N ** (1.0 / 3.0))
nsteps = 500000

filename = 'boson_configuration.txt'
positions = {}
if os.path.isfile(filename):
    f = open(filename, 'r')
    for line in f:
        a = line.split()
        positions[tuple([float(a[0]), float(a[1]), float(a[2])])] \
            = tuple([float(a[3]), float(a[4]), float(a[5])])
    f.close()

    if len(positions) != N:
        exit('error input file')
    print 'starting from file', filename
else:
    for k in range(N):
        a = levy_harmonic_path(1, beta)
        positions[a[0]] = a[0]
    print 'starting from scratch', filename

len_k_data = []
cycle_min = 10
```

```
for step in range(nsteps):
    boson_a = random.choice(positions.keys())
    perm_cycle = []
    while True:
        perm_cycle.append(boson_a)
        boson_b = positions.pop(boson_a)
        if boson_b == perm_cycle[0]:
            break
        else:
            boson_a = boson_b
    k = len(perm_cycle)
    len_k_data.append(k)
    perm_cycle = levy_harmonic_path(k, beta)
    positions[perm_cycle[-1]] = perm_cycle[0]
    for k in range(len(perm_cycle) - 1):
        positions[perm_cycle[k]] = perm_cycle[k + 1]
    a_1 = random.choice(positions.keys())
    b_1 = positions.pop(a_1)
    a_2 = random.choice(positions.keys())
    b_2 = positions.pop(a_2)
    weight_new = rho_harm(a_1, b_2, beta) * rho_harm(a_2, b_1, beta)
    weight_old = rho_harm(a_1, b_1, beta) * rho_harm(a_2, b_2, beta)
    if random.uniform(0.0, 1.0) < weight_new / weight_old:
        positions[a_1] = b_2
        positions[a_2] = b_1
    else:
        positions[a_1] = b_1
        positions[a_2] = b_2

f = open(filename, 'w')
for a in positions:
   b = positions[a]
   f.write(str(a[0]) + ' ' + str(a[1]) + ' ' + str(a[2]) + \
           ' ' +  str(b[0]) + ' ' + str(b[1]) + ' ' + str(b[2]) + '\n')
f.close()

pylab.title('B3, SMAC week 7, 2014')
pylab.hist(len_k_data, bins=200, normed=True)

pylab.xlabel('perm_cycle_length, k')
pylab.ylabel('$\\pi(k)$')
pylab.savefig('B3.png')
pylab.close()

pylab.title('B3 zoomed in, SMAC week 7, 2014')
pylab.hist(k_data, bins=200, normed=True)

pylab.ylim(0.0,0.01)
pylab.xlabel('perm cycle length, k')
pylab.ylabel('$\\pi(k)$')
pylab.savefig('B3_zoomed.png')
pylab.close()
```

Evaluation/feedback on the above work

Note: this section can only be filled out during the evaluation phase.

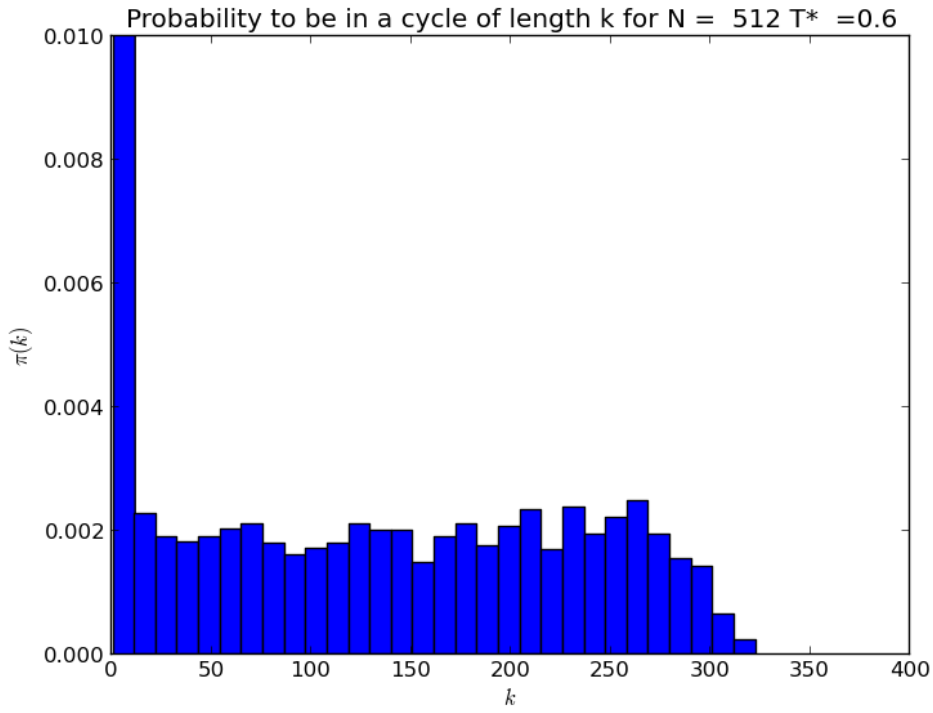
Here you evaluate your fellow student's understanding of cycle statistics.

Here is a solution that would yield full score:

1/ Yes, I can confirm that the probability to be on cycle of length k is independent of k for a large range of values of k
2/ To obtain the histogram, I simply added the second line shown here

```
k = len(perm_cycle)
data.append(k)
```

3/ Here is the output file:



POINTS - DESCRIPTION
Give **one point** if it is confirmed that the distribution \pi(k) is flat for a large window of k (in fact for 10 ~< k ~, 250, approximately).
Give **one point** if it is explained how one can obtain the histogram.
Give **one point** if the histogram is uploaded and if it looks qualitatively like the one shown

Score from your peers: **3**

peer 1 → *[This area was left blank by the evaluator.]*

peer 2 → You have made a tiny mistake and that is the reason why your bins are not really constant. You have bins that take two different ks because there are k up

to 350 and only 200 bins. So some bins have twice the height.

peer 3 → *[This area was left blank by the evaluator.]*

C Each year, numerous research papers are published on "**cigar-shaped**" and on "**pancake-shaped**" **Bose-Einstein condensates** (just google to convince yourself!). The first ones, in a limit, reduce to one-dimensional condensates, and the second one is close to two-dimensional condensates. To study them, download the below program, which gives the functions levy_harmonic_path for spring constants (omega_x, omega_y, omega_z).

```
import random, math, pylab, mpl_toolkits.mplot3d, numpy

omega = [4.0, 4.0, 1.0] # example, please adapt
omega = [1.0, 5.0, 1.0] # example, please adapt

def levy_harmonic_path(k, beta):
    sigma = [1.0 / math.sqrt(2.0 * omega[d] *
             math.tanh(0.5 * k * beta * omega[d])) for d in xrange(3)]
    xk = tuple([random.gauss(0.0, sigma[d]) for d in xrange(3)])
    x = [xk]
    for j in range(1, k):
        Upsilon_1 = [1.0 / math.tanh(beta * omega[d]) +
              1.0 / math.tanh((k - j) * beta * omega[d]) for d in range(3)]
        Upsilon_2 = [x[j - 1][d] / math.sinh(beta * omega[d]) + \
               xk[d] / math.sinh((k - j) * beta * omega[d]) for d in range(3)]
        x_mean = [Upsilon_2[d] / Upsilon_1[d] for d in range(3)]
        sigma = [1.0 / math.sqrt(Upsilon_1[d] * omega[d]) for d in range(3)]
        dummy = [random.gauss(x_mean[d], sigma[d]) for d in range(3)]
        x.append(tuple(dummy))
    return x

def rho_harm(x, xp, beta):
    Upsilon_1 = sum(omega[d] * (x[d] + xp[d]) ** 2 / 4.0 *
                    math.tanh(beta * omega[d] / 2.0) for d in range(3))
    Upsilon_2 = sum(omega[d] * (x[d] - xp[d]) ** 2 / 4.0 /
                    math.tanh(beta * omega[d] / 2.0) for d in range(3))
    return math.exp(- Upsilon_1 - Upsilon_2)

N = 1024
nsteps = 50000
omega_harm = 1.0
for d in range(3): omega_harm *= omega[d] ** (1.0 / 3.0)
T_star = 0.6
T = T_star * omega_harm * N ** (1.0 / 3.0)
beta = 1.0 / T
print 'omega: ', omega
positions = {}
for k in range(N):
    a = levy_harmonic_path(1, beta)
    positions[a[0]] = a[0]

for step in range(nsteps):
    boson_a = random.choice(positions.keys())
    perm_cycle = []
    while True:
        perm_cycle.append(boson_a)
        boson_b = positions.pop(boson_a)
        if boson_b == perm_cycle[0]: break
        else: boson_a = boson_b
    k = len(perm_cycle)
    perm_cycle = levy_harmonic_path(k, beta)
```

```
    positions[perm_cycle[-1]] = perm_cycle[0]
    for j in range(len(perm_cycle) - 1):
        positions[perm_cycle[j]] = perm_cycle[j + 1]
    a_1 = random.choice(positions.keys())
    b_1 = positions.pop(a_1)
    a_2 = random.choice(positions.keys())
    b_2 = positions.pop(a_2)
    weight_new = rho_harm(a_1, b_2, beta) * rho_harm(a_2, b_1, beta)
    weight_old = rho_harm(a_1, b_1, beta) * rho_harm(a_2, b_2, beta)
    if random.uniform(0.0, 1.0) < weight_new / weight_old:
        positions[a_1], positions[a_2] = b_2, b_1
    else:
        positions[a_1], positions[a_2] = b_1, b_2


fig = pylab.figure()
ax = mpl_toolkits.mplot3d.axes3d.Axes3D(fig)
ax.set_aspect('equal')

n_colors = 8
list_colors = pylab.cm.rainbow(numpy.linspace(0, 1, n_colors))[::-1]
dict_colors = {}
i_color = 0
positions_copy = positions.copy()
while positions_copy:
    x, y, z = [], [], []
    starting_boson = positions_copy.keys()[0]
    boson_old = starting_boson
    while True:
        x.append(boson_old[0])
        y.append(boson_old[1])
        z.append(boson_old[2])
        boson_new = positions_copy.pop(boson_old)
        if boson_new == starting_boson: break
        else: boson_old = boson_new
    len_cycle = len(x)
    if len_cycle > 2:
        x.append(x[0])
        y.append(y[0])
        z.append(z[0])
    if len_cycle in dict_colors:
        color = dict_colors[len_cycle]
        ax.plot(x, y, z, '+-', c=color, lw=0.75)
    else:
        color = list_colors[i_color]
        i_color = (i_color + 1) % n_colors
        dict_colors[len_cycle] = color
        ax.plot(x, y, z, '+-', c=color, label='k=%i' % len_cycle, lw=0.75)
pylab.title(str(N) + ' Bosons at T* = ' + str(T_star))
pylab.legend()
ax.set_xlabel('$x$', fontsize=16)
ax.set_ylabel('$y$', fontsize=16)
```

```
ax.set_zlabel('$z$', fontsize=16)
xmax = 10.0
ax.set_xlim3d([-xmax, xmax])
ax.set_ylim3d([-xmax, xmax])
ax.set_zlim3d([-xmax, xmax])
pylab.savefig('Boson_configuration')
pylab.show()
```

C1 Modify this program so that it allows you to do input and output, as in section **B1**. Run the program several times at temperature T* = 0.6 for 1024 particles. Use two sets of parameters

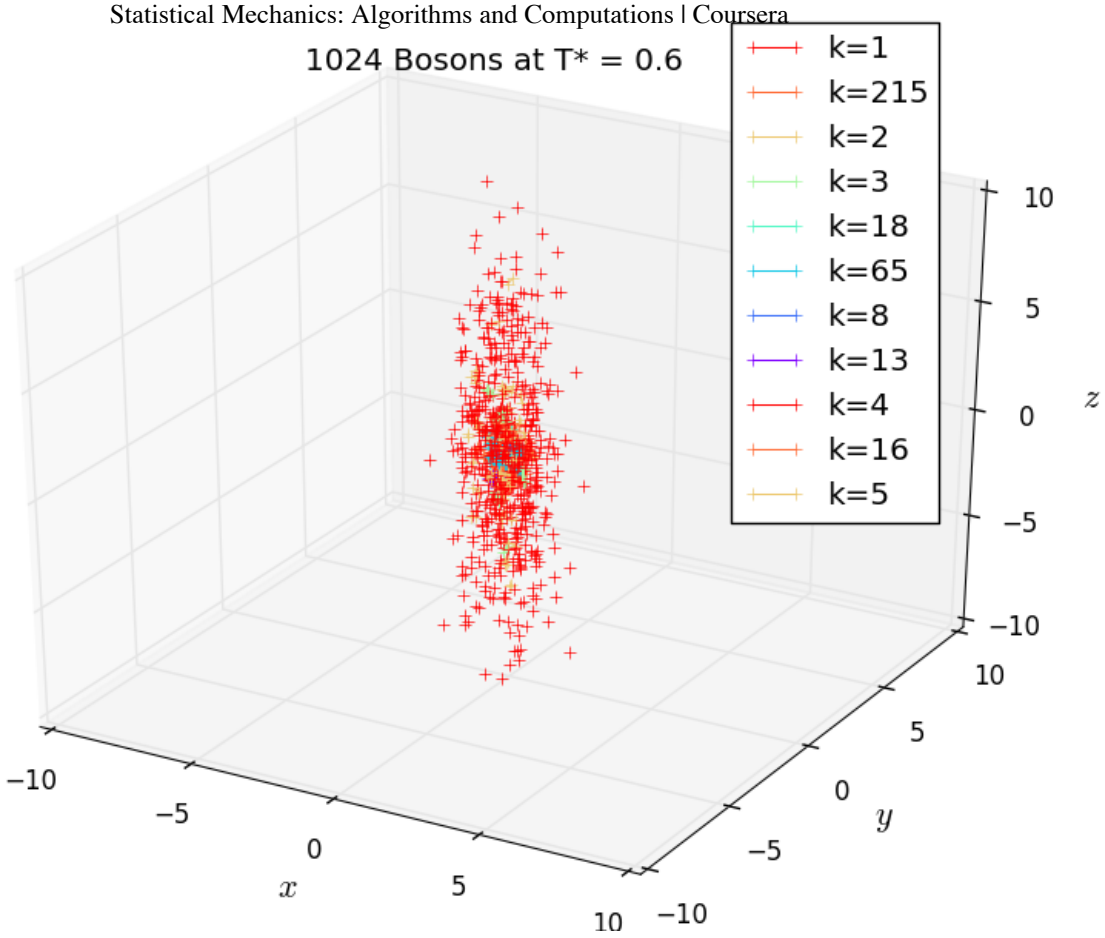
```
omega = [1.0, 5.0, 1.0]
omega = [4.0, 4.0, 1.0]
```

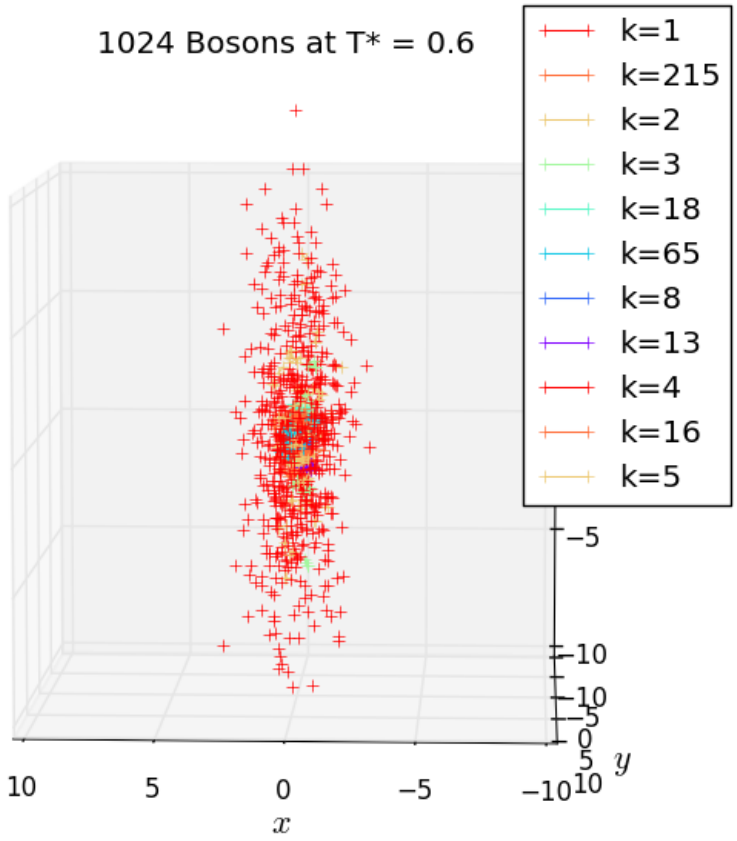
Then answer the following questions:

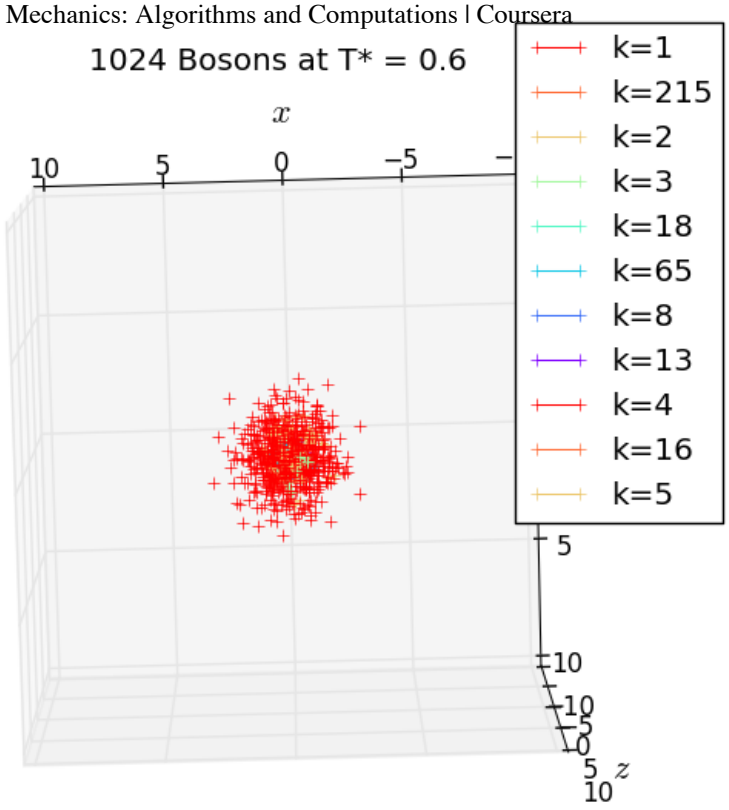
1. which of the two choices corresponds to the "**cigar-shaped**" trap? (Run the simulation for 5 or 10 minutes, then **upload** two screenshots
   of the output under different angles that allow you to understand that you're looking at a cigar.) NB: "Screenshot" means that you simply take a picture of what you see on the screen. Take pictures with your cell-phone if you don't know how to produce a screenshot. Otherwise draw a picture and scan it.
2. which of the two choices corresponds to the "**pancake-shaped**" trap? (Run the simulation for 5 or 10 minutes, then **upload** two screenshots
   of the output, that allow you to understand tat you're looking at a **pancake**.) NB: "Screenshot" means that you simply take a picture of what you see on the screen. Take pictures with your cell-phone if you don't know how to produce a screenshot. Otherwise draw a picture and scan it.
3. Can you confirm that Bose-Einstein condensation sets in at T \sim 0.9 * (omega_x * omega_y * omega_z)**(1./3.) * N ** (1/3) for both choices of omega_x, omega_y, omega_z? Run your codes for different temperatures, and relate your findings, briefly. (**Don't write a master's thesis** on the subject, **just do two brief calculations** for example at T* = 1.2, to check that the formula seems OK. **Notice the relation between permutation cycles** (shown in the legend) **and the Bose-Einstein condensation**).

NB: if you are unable to get mpl_toolkits.mplot3d.axes3d.Axes3D to run, simply produce histograms in two-dimensional cuts in xy, xz, and yz.
NNB: Attention - many points for this exercise. Don't drop it.

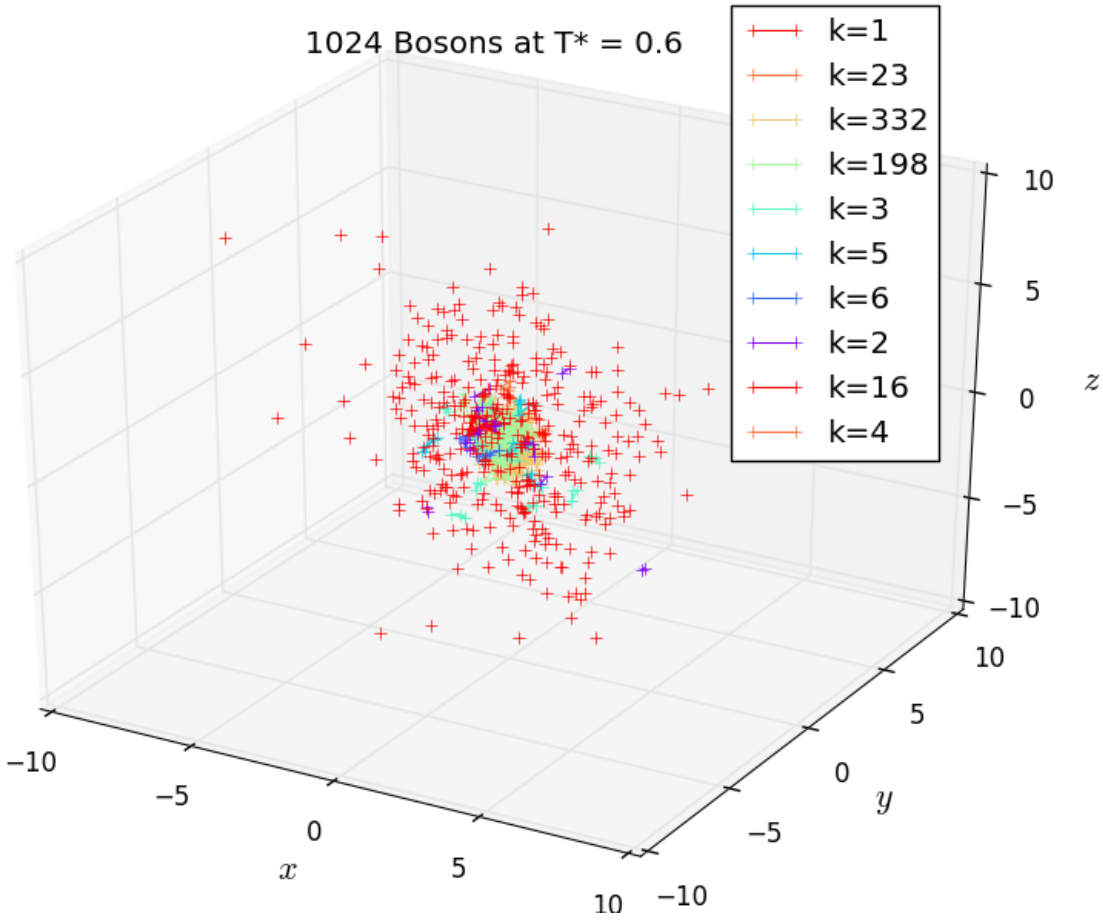
1. **omega = [4.0, 4.0, 1.0] corresponds to the "cigar-shaped" trap at T* = 0.6**. Below are some screenshots.

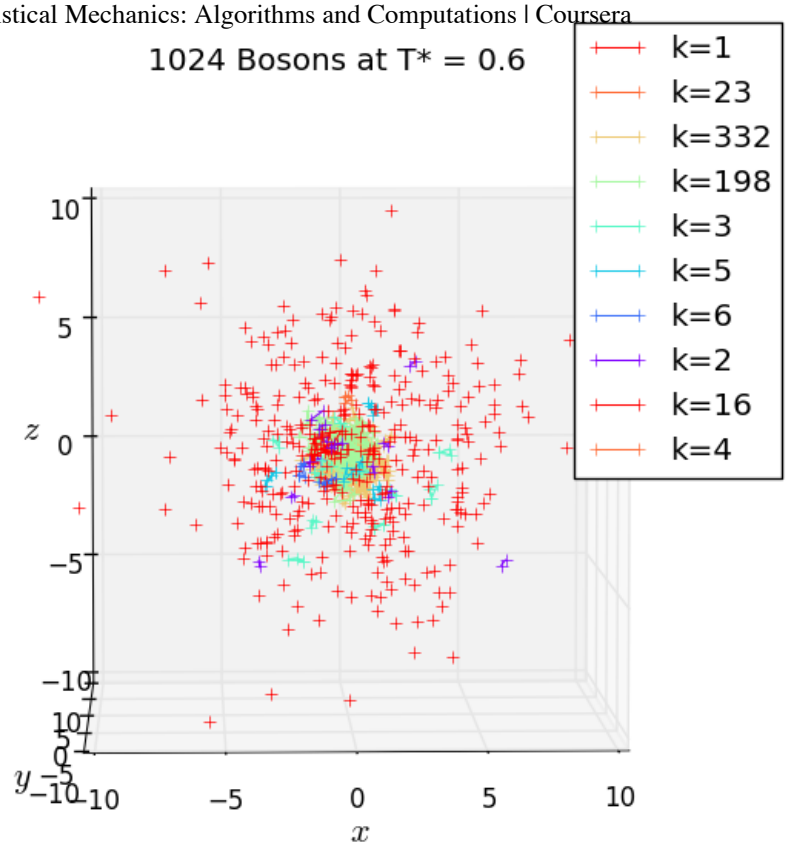

1024 Bosons at T* = 0.6
k=1
k=215
k=2
k=3
k=18
k=65
k=8
k=13
k=4
k=16
k=5
x
y
z

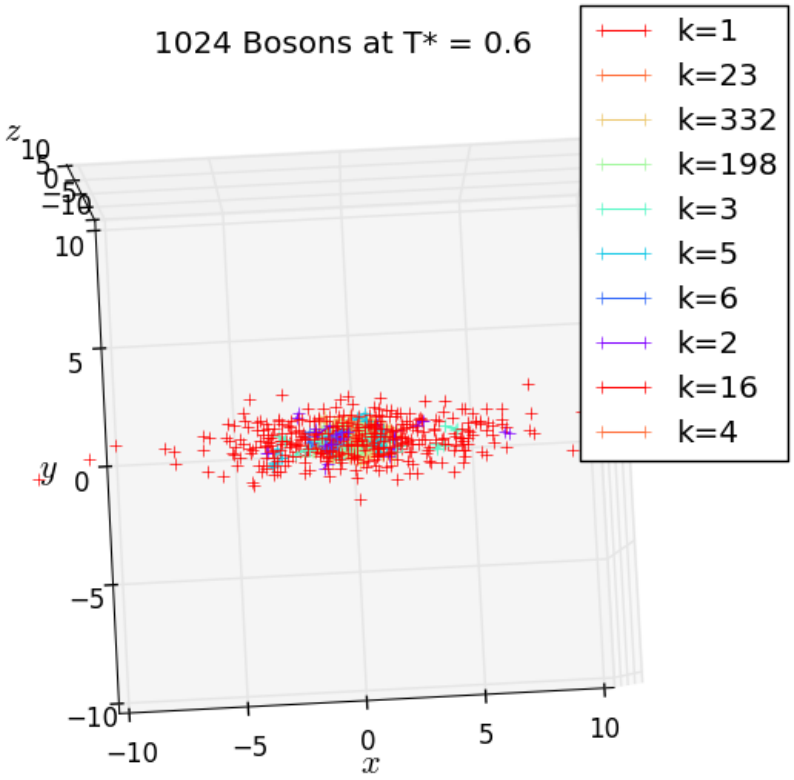

1024 Bosons at T* = 0.6
k=1
k=215
k=2
k=3
k=18
k=65
k=8
k=13
k=4
k=16
k=5
x
y



2. **omega = [1.0, 5.0, 1.0] corresponds to the "pancake-shaped" trap at T* = 0.6.** Below are some screenshots.







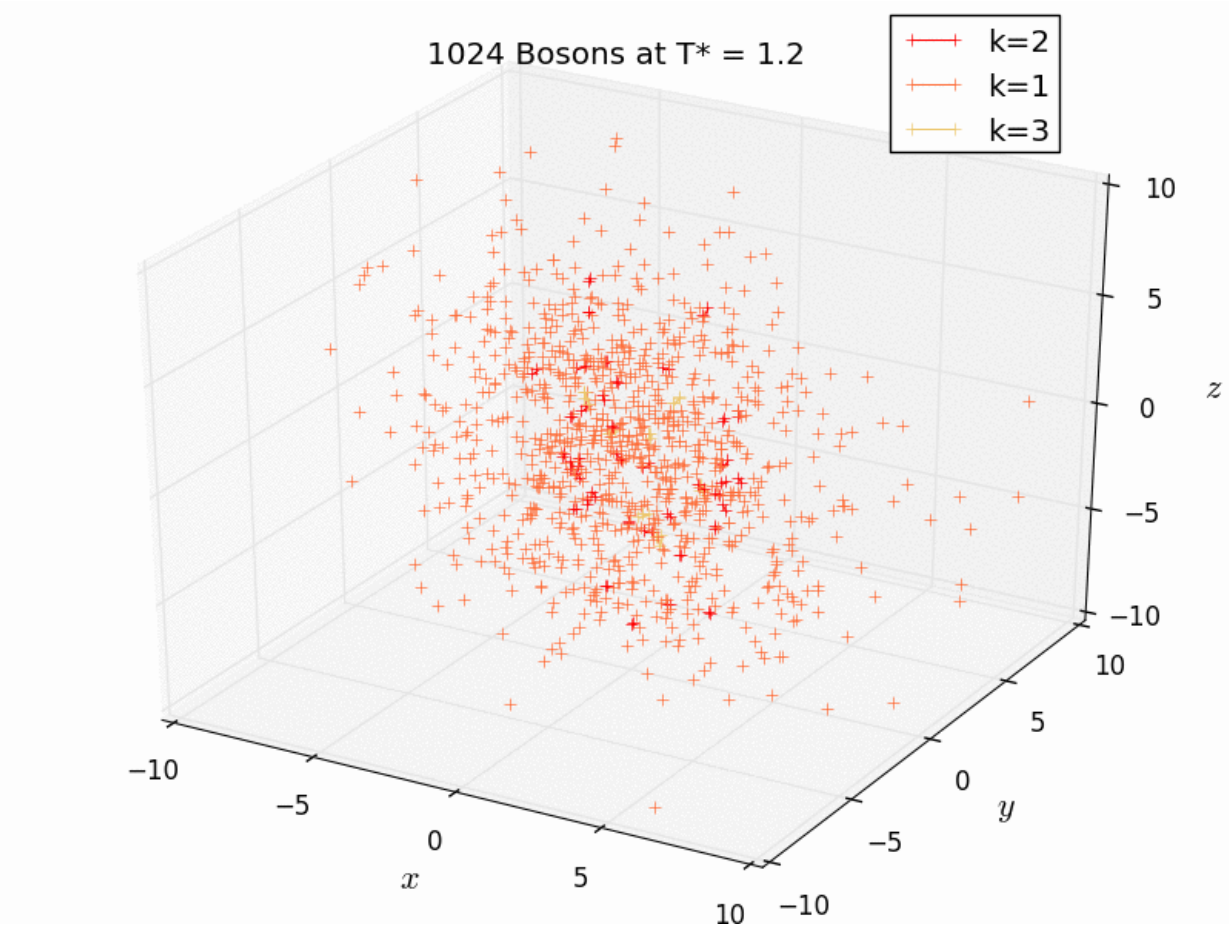
3. The above set of simulations were done at T* = 0.6. As suggested, new simulations at T* = 1.2 and T* = 0.9 were done.

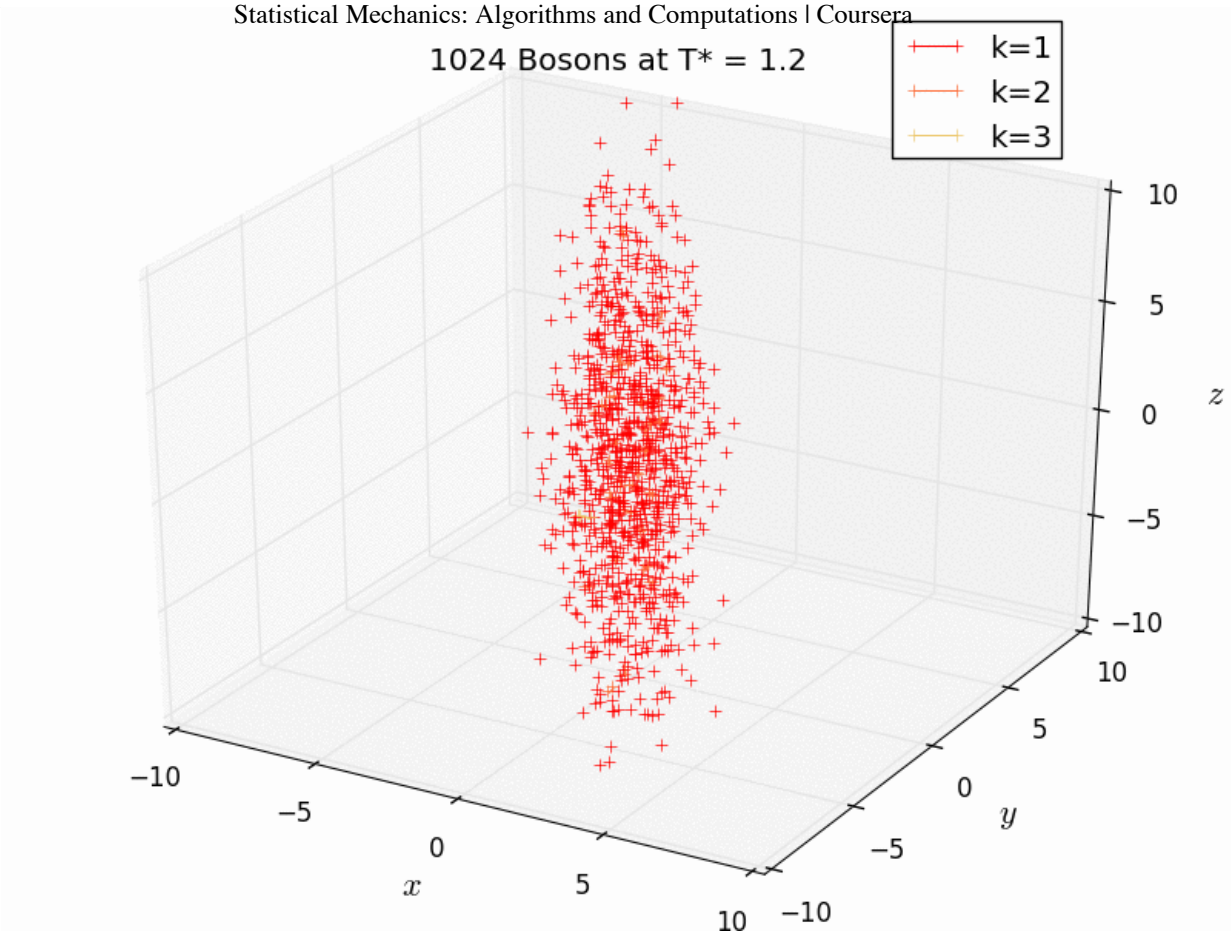
As one can clearly see, at T* = 1.2, we do not have any Bose-Einstein condensation for either of the Omegas (first movie is for (1,5,1) and second movie is for (4,4,1). The permutation cycles are much shorter at T* = 1.2 than at T* = 0.6, showing the relation between length of permutation cycles (k) and

Bose-Einstein condensation.

At T* = 0.9, as indicated in the question, we see onset of Bose-Einstein condensation. There are few permutation cycles of longer length at this temperature.

The two movies below show the condensation process when T* gets lowered from 1.2 to 0.6 in steps of 0.3 (top for (1,5,1) - pancake-shaped trap and below for (4,4,1) - cigar-shaped trap).





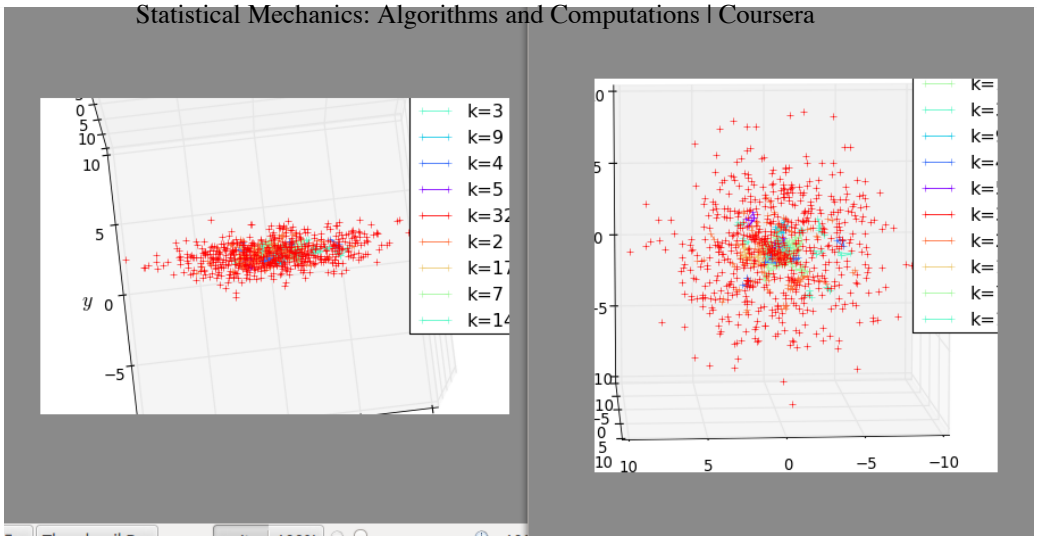
Evaluation/feedback on the above work

Note: this section can only be filled out during the evaluation phase.

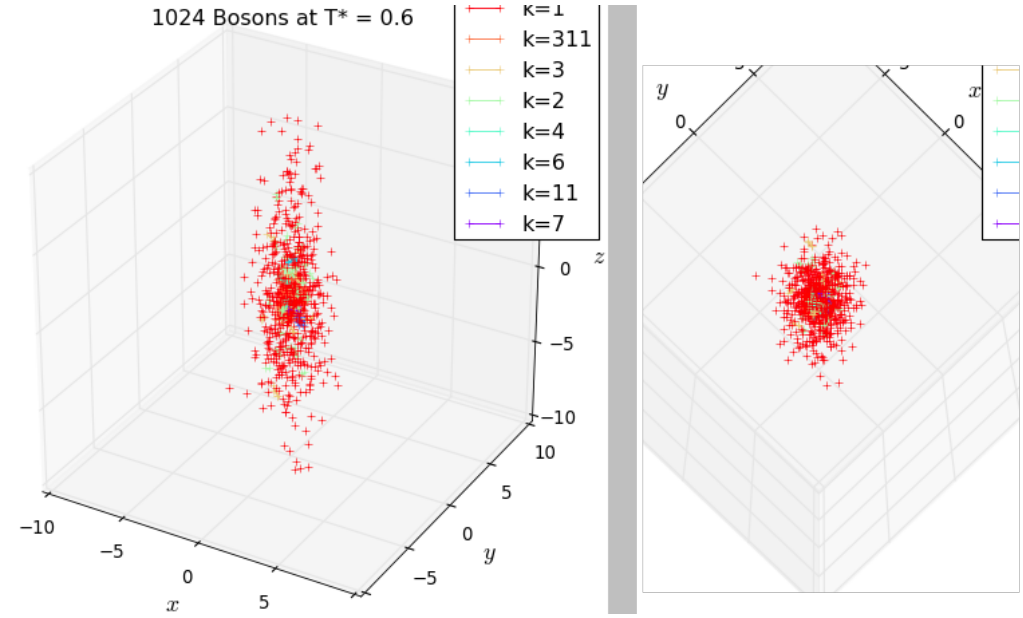
Here you evaluate your fellow student's ability to make sense of a complicated, yet beautiful subject.

Here is a solution that would yield full score:

[1.0, 5.0, 1.0] is the pancake-shaped trap. Here are two views of the pancake-shaped cloud of bosons from different perspectives:



Here are two views of the cloud of particles in the cigar-shaped trap (4,4,1) :



Yes, I can confirm that the formula for the critical temperature seems OK**,** I ran the program for both sets of omega parameters at T* = 1.2, and the permutation cycles were all of a sudden very short.

POINTS - DESCRIPTION

Give one point if the pancake parameters have been identified correctly
Give one point if the screenshots for the pancake-shaped cloud are conclusive
Give one point if the cigar parameters have been identified correctly
Give one point if the screenshots for the pancake-shaped cloud are conclusive
Give one point if it has been understood that at T* larger than 1, the permutations go away

Score from your peers: **5**

peer 1 → *[This area was left blank by the evaluator.]*

peer 2 → *[This area was left blank by the evaluator.]*

peer 3 → *[This area was left blank by the evaluator.]*