(http://help.coursera.org/customer/widget/emails/new)

Peer Assessments (https://class.coursera.org/smac-001/human_grading/)
/ Homework session 5: Quantum statistical mechanics and Quantum Monte Carlo Mecha
Help (https://class.coursera.org/smac-001/help/peergrading?url=https%3A%2F%2Fclass.coursera.org%2Fsmac-001%2Fhuman_grading%2Fview%2Fcourses%2F971628%2Fassessments%2F9%2Fresults%2Fmine)

Submission Phase

1. Do assignment ☑ (/smac-001/human_grading/view/courses/971628/assessments/9/submissions)

Evaluation Phase

2. Evaluate peers ☑ (/smac-001/human_grading/view/courses/971628/assessments/9/peerGradingSets)

Results Phase

3. See results ☑ (/smac-001/human_grading/view/courses/971628/assessments/9/results/mine)

Your effective grade is 19.5

Your unadjusted grade is 19.5, which is simply the grade you received from your peers.

See below for details.

In this Homework Session 5 of **Statistical Mechanics: Algorithms and Computations**, we do a few calculations in quantum mechanics. We consider the harmonic and the anharmonic oscillator. You will do matrix squaring and, at the end, perform two Quantum Monte Carlo simulation.

NB: The upload of python programs should now be possible. Otherwise, copy-and-paste them but please transform them with the <code> button.

A In this section we provide three preparation programs.

- Preparation program 1 is dedicated to Python "dictionaries".
- Preparation program 2 is dedicated to discussion of "numpy arrays".
- Preparation program 3 defines anharmonic potentials, plots them and computes exact and approximate partition functions.

Preparation program 1

```
Names = {}
Names['Albert'] = 'Einstein'
Names['Satyendra'] = 'Bose'
Names['Richard'] = 'Feynman'
Names['Ludwig'] = 'Boltzmann'
# checkpoint 1
for name in Names: print name, Names[name]
a = Names.pop('Albert')
# checkpoint 2
del Names['Richard']
# checkpoint 3
L = Names.keys()
M = Names.values()
#checkpoint 4
b = 'Wolfgang' in Names
#checkpoint 5
```

Preparation program 2

```
import numpy

a = numpy.array([[1, 2, 3], [4, 5, 6]])
b = numpy.array([[1, 2], [3, 4], [5, 6]])
c = numpy.dot(a, b)
d = numpy.dot(b, a)
e = d * 2
f = numpy.diag(c)
g = numpy.diag(c).sum()
```

Preparation program 3

```
import pylab, math

def V(x):
    pot =  x ** 2 / 2 + cubic * x ** 3 + quartic * x ** 4
    return pot

def Energy(n, cubic, quartic):
    return n + 0.5 - 15.0 / 4.0 * cubic **2 * (n ** 2 + n + 11.0 / 30.0) \
         + 3.0 / 2.0 * quartic * (n ** 2 + n + 1.0 / 2.0)

def Z(cubic, quartic, beta, n_max):
    Z = sum(math.exp(-beta * Energy(n, cubic, quartic)) for n in range(n_max + 1))
    return Z

cubic = -0.45
quartic = 0.45
x_max = 5.0
nx = 100
dx = 2.0 * x_max / (nx - 1)
x = [i * dx for i in range(-(nx - 1) / 2, nx / 2 + 1)]
y = [V(a) for a in x]
pylab.plot(x, y)

cubic = 0.0
quartic = 0.0
y = [V(a) for a in x]
pylab.title('XYZ')
pylab.xlabel('x_label')
pylab.ylabel('y_label')
pylab.plot(x, y, label='song of joy')
pylab.legend()
pylab.axis([-4.0, 4.0, 0.0, 3.0])
pylab.show()
```

A1 Download (cut-and-paste) **Preparation program 1**. If you are unfamiliar with the concept of Python dictionaries (also called hashes, or hash tables), consult an online help. Run **Preparation program 1**, then answer the following questions:

1/ At Checkpoint 1, what are the keys and values of the dictionary *Names* (print to find out)?
2/ At Checkpoint 2, what is *a*? What are the keys and values now?
3/ At Checkpoint 3, what is the new content of *Names*?
4/ At Checkpoint 4, what datatype are *L* and *M* (they are not dictionaries)?
5/ At Checkpoint 5, what is *b*?
6/ In our example, the keys of the dictionaries are first names, or in other words strings. Can the keys be floats (real numbers), like 3.1415?
Can the keys be lists, like L =[1.4,1.5]? Same question for the values, can they be integers, floats or lists?

1. Keys - Albert, Satyendra, Richard and Ludwig.

Corresponding values - Einstein, Bose, Feynman and Boltzmann.

2. a = Einstein.
   Keys - Satyendra, Richard and Ludwig.
   Corresponding values - Bose, Feynman and Boltzmann.

3. Content of Names dictionary:
   Keys - Satyendra and Ludwig.
   Corresponding values - Bose and Boltzmann.

4. L and M are lists. L and M contain the lists of keys and values of dictionary 'Names' respectively.

5. b is a boolean variable which returns True or False depending on whether "Wolfgang" is present in Names or not. In this case, **b is False**.

6. Keys: Floats - Yes, Lists - No
   Values: Integers - Yes, Floats - Yes, Lists - Yes

Evaluation/feedback on the above work

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

A1 Here, you are asked to evaluate your fellow student's understanding of **dictionaries**, at least of some of their aspects. There are 6 aspects (questions).

An example of an answer that would receive full score is as follows:

1/ Checkpoint 1: The keys are the first names ('Albert', 'Satyendra', 'Richard', 'Ludwig') and the values are the corresponding last names ('Einstein', 'Bose', 'Feynman', 'Boltzmann').
2/ Checkpoint 2: a is 'Einstein', the value of key 'Albert'. We have removed that key, and thus also lost its value 'Einstein'.
3/ Chekpoint 3: we have removed the key 'Richard' and so also lost the value 'Feynman'.
4/ Checkpoint 4: L and M are lists, L contains the keys (first names) from our dictionary. M contains the values (last names).
5/ Checkpoint 5: b is the boolean False, as our dictionnary doesn't contain the key 'Wolfgang'.
6/ As keys we can also use floats, but not lists. As values we can also use floats and list.

NB: See the following example

```
Dict = {}
Dict[3.1415] = 'the transcendental number Pi'
Dict[5] = 2.78
Dict['fibonacci'] = [1, 1, 2, 3, 5, 8, 13]
```

Remember that we do not expect your fellow students to be experts in computer science. Working knowledge is enough....

POINTS - DESCRIPTION
Give 0 points if 0 aspect is treated correctly.
Give 1 points if 1 or 2 aspects are treated correctly.
Give 2 points if 3 or 4 aspects are treated correctly.
Give 3 points if 5 or 6 aspects are treated correctly.

Score from your peers: **2.5**

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

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

A2
Download (cut-and-paste) **Preparation program 2**. Run **Preparation program 2**, then answer the following questions (add print statements into the program where needed). We are concerned with numpy, a convenient numerical library inside Python.

1/ *a* is a numpy array. Can you confirm that it stands for the matrix
1 2 3
4 5 6
with elements a_00 = 1, a_10 = 4, a_01 = 2, a_11 = 5, a_02 = 3, a_12 = 6?
2/ Write down all elements of *b*.
3/ What is numpy.dot? Write down all elements of *c*.
4/ Write down all elements of *d*. Can you confirm that *a.b* differs from *b.a*?
5/ Explain *e*, write down all its elements.
6/ The trace of a matrix is defined as the sum of its diagonal elements. How is it computed in **Preparation program 2**? Write down the trace of the matrix *c*.

1. Printing out a gives:
[[1 2 3]
[4 5 6]]
Therefore,indeed, it stands for the matrix
1 2 3
4 5 6

2. It's a 3x2 matrix
b=1 2
3 4
5 6

3. numpy.dot multiplies two matrices. c = a x b. We get a 2x2 matrix since a is 2x3 and b is 3x2.
c = 22 28
49 64

4. d = b x a. We get a 3x3 matrix since b is 3x2 and a is 2x3.
d = 9 12 15

19 26 33
29 40 51

5. e = d * 2. All elements of d are multiplied by scalar 2.
e = 18 24 30
38 52 66
58 80 102

6. f stores the diagonal elements of c and g stores the sum.
f = 22 64
g = 86

Evaluation/feedback on the above work

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

A2 Here, you are asked to evaluate your fellow student's understanding of a **simple numpy program**. There are 6 aspects (questions).

An example of an answer that would receive full score is as follows:
1/ a indeed stands for the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

The entry a_ij can be obtained with a[i][j]
2/ The entries of b are b_00 = 1, b_01 = 2, b_10 = 3, b_11 = 4, b_20 = 5, b_21 = 6
3/ numpy.dot(a,b) computes the matrix multiplication of a with b. Its result is:

$$c = \begin{pmatrix} 22 & 28 \\ 49 & 64 \end{pmatrix}$$

4/ d is different from c: matrix mutliplication is not commutative. The result is:

$$d = \begin{pmatrix} 9 & 12 & 15 \\ 19 & 26 & 33 \\ 29 & 40 & 51 \end{pmatrix}$$

5/ e is the scalar multiplication of d by 2. Every entry of d is multiplied by 2. The result is:

$$d = \begin{pmatrix} 18 & 24 & 30 \\ 38 & 52 & 66 \\ 58 & 80 & 102 \end{pmatrix}$$

6/ The trace of c is calculated by extracting its diagonal entries into an array f, whose entries are then added up to g. The trace of c is 86.

POINTS - DESCRIPTION
Give 0 points if 0 aspects are treated correctly
Give 1 points if 1 or 2 aspects are treated correctly
Give 2 points if 3 or 4 aspects are treated correctly
Give 3 points if 5 or 6 aspects are treated correctly

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.]*

A3 In physics, 'harmonic' means 'quadratic'. The harmonic oscillator thus corresponds to a potential 1/2 x^2 and, as we discussed in the lecture, the quantum harmonic oscillator has energy levels E_0 = 1/2, E_1 = 3/2, E_2 = 5/2,...

Download (cut-and-paste) **Preparation program 3**. This program provides the anharmonic corrections to the harmonic potential. Run and modify this program and address the following questions and issues:

1/ The **Preparation program 3** is able to plot harmonic and anharmonic potentials. Which values of the parameters correspond to the harmonic oscillator? What is the condition for the potential to be + infinity for x -> +/- infinity?
2/ Plot the potential for the harmonic case and for cubic = -0.5 and quartic = 0.5. Make sure you provide a descriptive title, arrange for good xlabels and y-labels, and provide one missing curve label (for the moment only one of the plots has a label, but not a very good one). **Upload this plot.**
3/ The **Preparation program 3** provides the 'perturbation theoretical' corrections for the change of the energy levels of the anharmonic oscillator for ***very small* values of cubic and quartic corrections**. Plot the energy values E_n for 0 < quartic = -cubic < 1. The program is already written, but you should provide appropriate title, axis labels and good labels for the individual curves. **Upload this plot.**
4/ Much work has gone into computing the perturbation theoretical formula for E_n (contained in our function Energy). Given that the exact eigenvalues satisfy: E_n > 0 if V(x) > 0 \forall x, can you show that the formula ceases to be valid quite quickly?

NB: The perturbation-theoretical calculation is performed, for example,in Landau Lifshitz: "Quantum Mechanics (vol 3)", exercise 3 of chap 38. No need to look this reference up for this exercise.

5/ The **Preparation program 3** is also able to compute the partition function Z = sum_n \exp (- beta E_n) where beta = 1/(k_b T) is the inverse temperature. Can you confirm (numerically) that the partition function for the harmonic oscillator equals Z^harm (beta) = 1/(2 sinh(beta/2)) ?

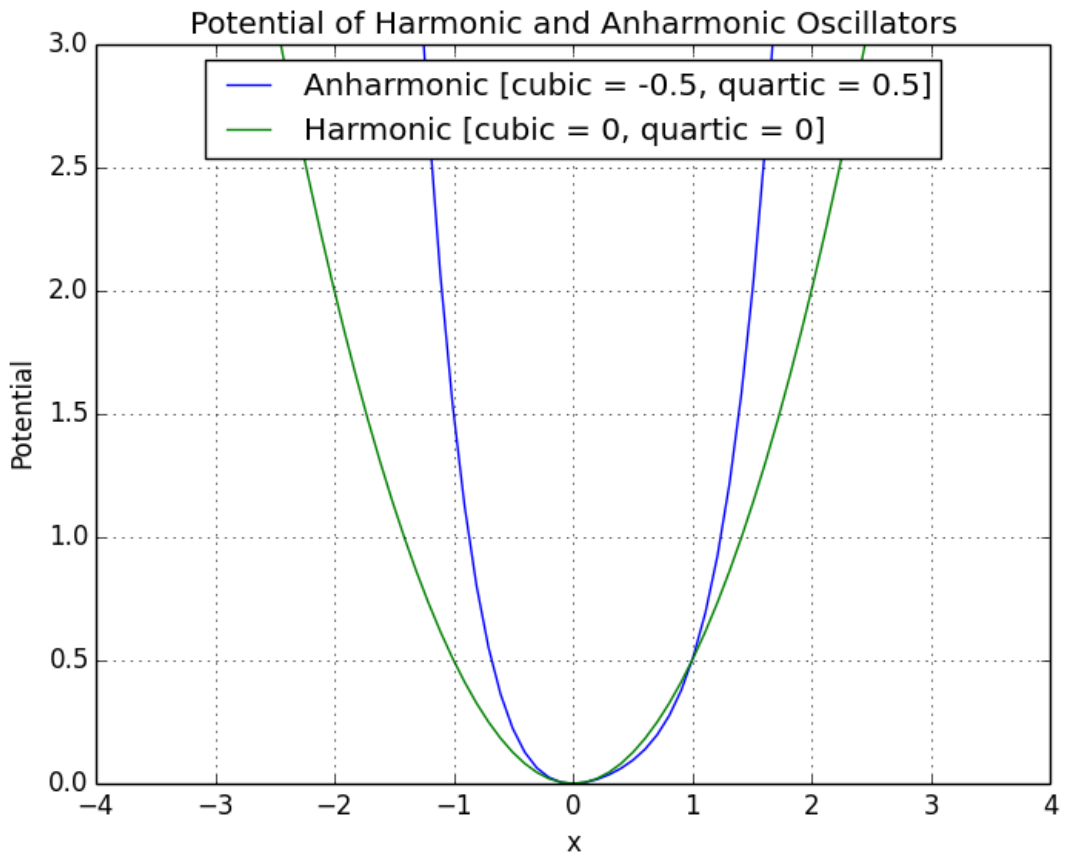
1. cubic=0 and quartic=0 correspond to harmonic oscillator.

$Potential = \frac{x^2}{2}$

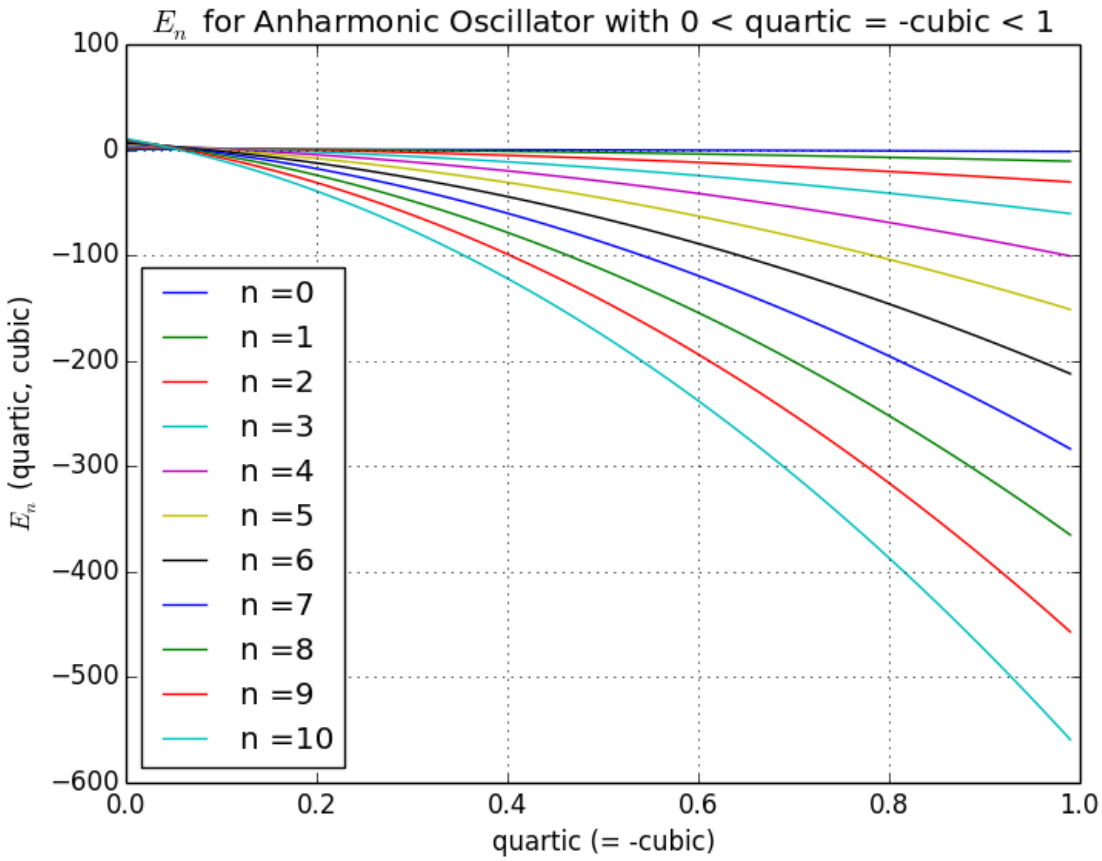
Therefore, $Potential \to \infty$ when $x \to \infty$ or $\to -\infty$

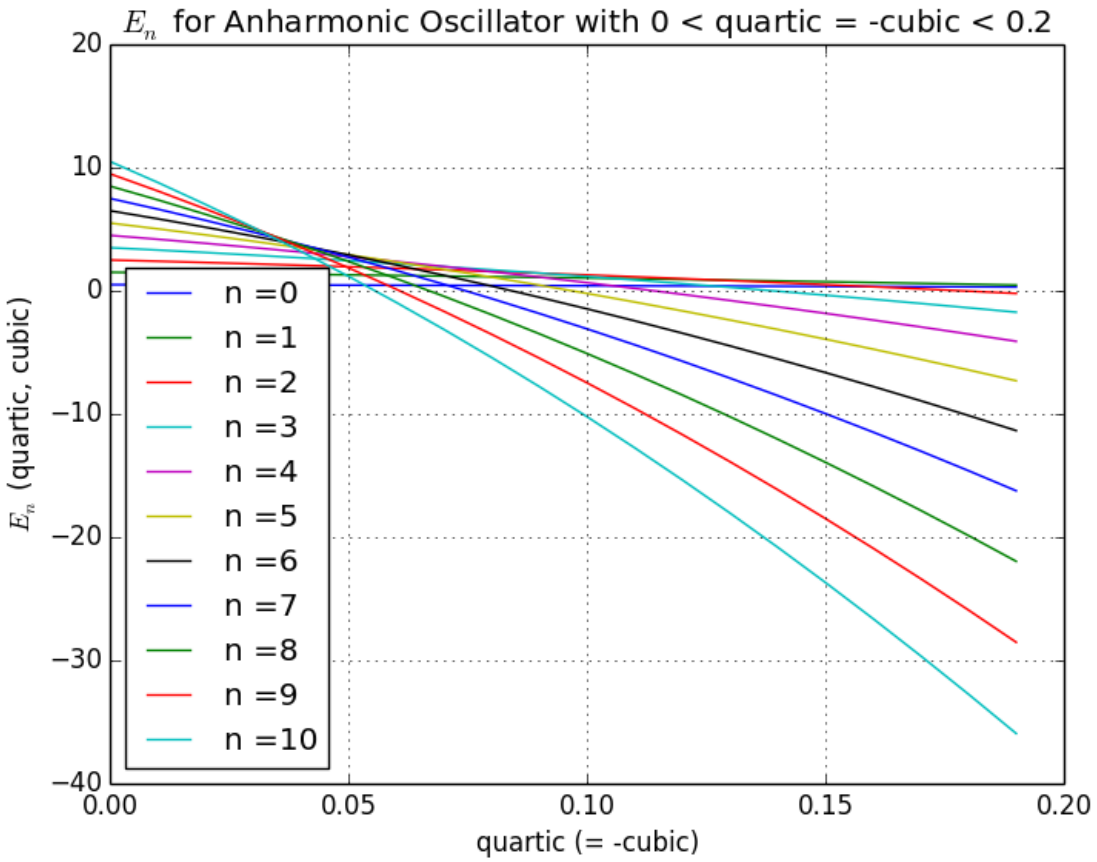
For other values of parameters cubic and quartic, if Potential remains positive for all x, then we can also have this requirement satisfied.

2. Plot is below:



3. Plots are below (first is for 0 to 1.0 and second for 0 to 0.2 as zoomed in version)





4. Clearly, Energy becomes negative for increasing values of quartic parameter. The exact value of quartic where this happens gets lower and lower with increasing n. Since, we should have $E_n > 0$ as $V(x) > 0$ for all x, the formula ceases to be valid with increasing n.

5. $E_n^{h.o.} = n + \frac{1}{2}$

$Z^{h.o.}(\beta) = \sum_{n=0}^{\infty} e^{-\beta E_n^{h.o.}} = e^{-\beta/2} + e^{-3\beta/2} + e^{-5\beta/2} + \ldots$

Solving, we get $Z^{h.o.}(\beta) = e^{-\beta/2}(\frac{1}{1-e^{-\beta}}) = \frac{1}{e^{\beta/2} - e^{-\beta/2}} = \frac{1}{2sinh(\beta/2)}$

Following list provides results for β = 0.1, 1, 10. n_max = 100, and cubic = quatric = 0 since we are considering a simple harmonic oscillator.

Partition Sum (n=100, beta=0.1) is (function) 9.99542392385 (equation) 9.99583454829
Partition Sum (n=100, beta=1) is (function) 0.959517375667 (equation) 0.959517375667
Partition Sum (n=100, beta=10) is (function) 0.00673825291529 (equation) 0.00673825291529

We see a good match between the partitions sums obtained via the function in the program and the analytical expression derived above.

Evaluation/feedback on the above work

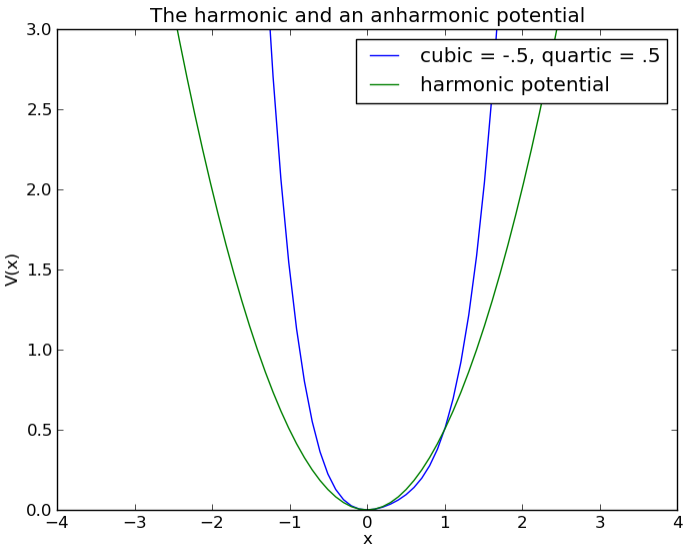
Note: this section can only be filled out during the evaluation phase.

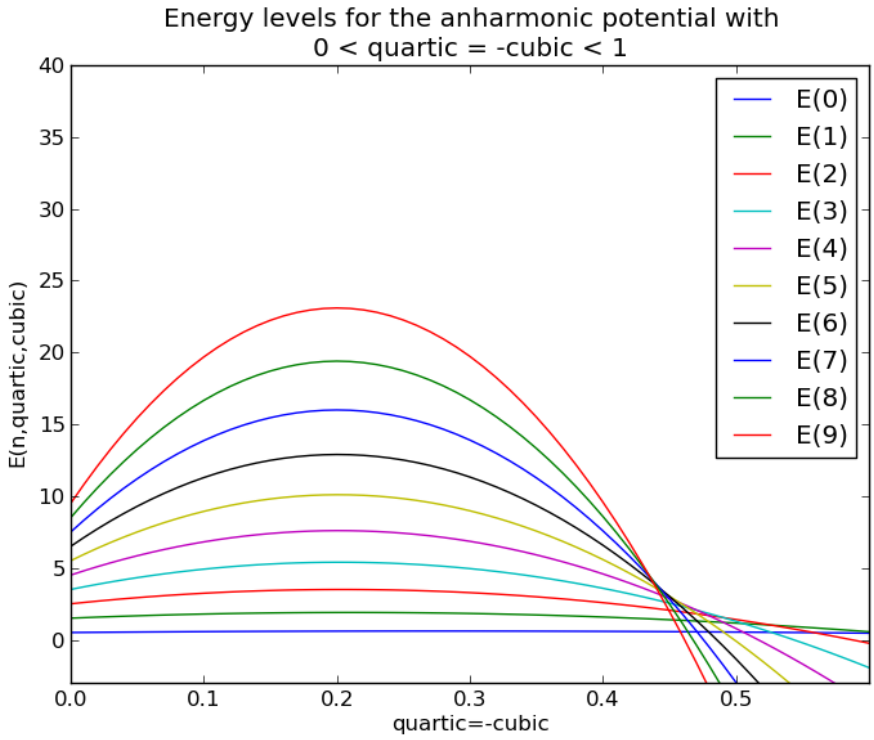
A3 Here, you are asked to evaluate your fellow student's understanding of the

harmonic and anharmonic potential and of the meaning of the perturbation approximation.

An example of an answer that would receive full score is as follows:

1/ The harmonic oscillator is obtained for the parameters cubic = 0, and quadric = 0.
2/ The potential goes to infinity for quartic>0 [and also for quartic=0 and cubic = 0] ([] part not required).
3/ Upload of first plot.
4/ Upload second plot.
5/ Understanding that for quartic = -cubic > 0.5, the perturbation theory is certainly rubbish.





The partition function is almost indistinguishable from the exact solution.

POINTS - DESCRIPTION

Cumulative, as shown here

Give one point if these two aspects are understood:
- The harmonic oscillator is obtained for the parameters cubic = 0, and quadric = 0.
- The potential goes to infinity for quartic>0 [and also for quartic=0 and cubic = 0] ([] part not required).

Give one point if both plots are uploaded.

Give one point if it is understood that perturbation theory cannot apply for quartic = -cubic > 0.5.

Score from your peers: **2.5**

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

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

B In part **B**, you will redo calculations for the harmonic oscillator (in lecture 5 and tutorial 5) and verify the consistency of the sum over wavefunctions and energy eigenvalues, Matrix squaring and Quantum Monte Carlo.
NB: The density matrix is given by

rho(x, x', beta) = \sum_n \exp(- beta * E_n) \psi_n(x) \psi_n*(x')

where the asterisk (*) denotes complex conjugation and is without action in our case. The diagonal density matrix corresponds to x = x'. Note that there is a '=' sign in the above formula, without a normalizing factor.

NNB: The Trotter decomposition is given by
rho(x, x', beta) = exp(-beta V(x)/2) rho_free(x, x', beta) exp(-beta V(x')/2)

as discussed in the lecture and the tutorial. Notice that there is a "one half" from the Trotter decomposition, and another "one half" from the potential V(x) = x^2/2.

B1 Download (cut-and-paste) the below program (harmonic_wavefunction.py) and modify it so that it computes the diagonal density matrix at inverse temperatures beta. Furthermore, compute the partition function Z using the three formulas:

a) Z(beta) = \sum_n exp(-beta E_n) (general, any quantum system)
b) Z(beta) = \int dx rho(x, x, beta) (general, any quantum system) **NOTE ADDED** (03/11/2014: Nothing complicated here, just a sum of terms)
c) Z(beta) = Z^harm (beta) = 1/(2 sinh(beta/2)) (harmonic oscillator only)
(see Section A3)

```
import math

n_states = 40
Energies = [0.5 + i for i in range(n_states)]
grid_x = [i * 0.2 for i in range(-25, 26)]
psi = {}
for x in grid_x:
    psi[x] = [math.exp(-x ** 2 / 2.0) / math.pi ** 0.25]
    psi[x].append(math.sqrt(2.0) * x * psi[x][0])
    for n in range(2, n_states):
        psi[x].append(math.sqrt(2.0 / n) * x * psi[x][n - 1] -
                      math.sqrt((n - 1.0) / n) * psi[x][n - 2])
```

Then address the following aspects:
1/ Compute the partition function using the three methods for beta = 2. **Communicate your results.**

2/ The normalized probability to be at position x is given by pi(x) = const rho(x, x, beta). **What is this constant** if we require that
\int_-infty^infty \pi(x) = 1. Modify your program so that it outputs the normalized pi(x) in a graphics file. Use labels for plots, as in Section **A3** (titles, axis labels, graph labels). **Upload this graphics** file for temperature beta = 2.

3/ **Upload your program.**

1. The partition functions calculated by methods a, b and c are:

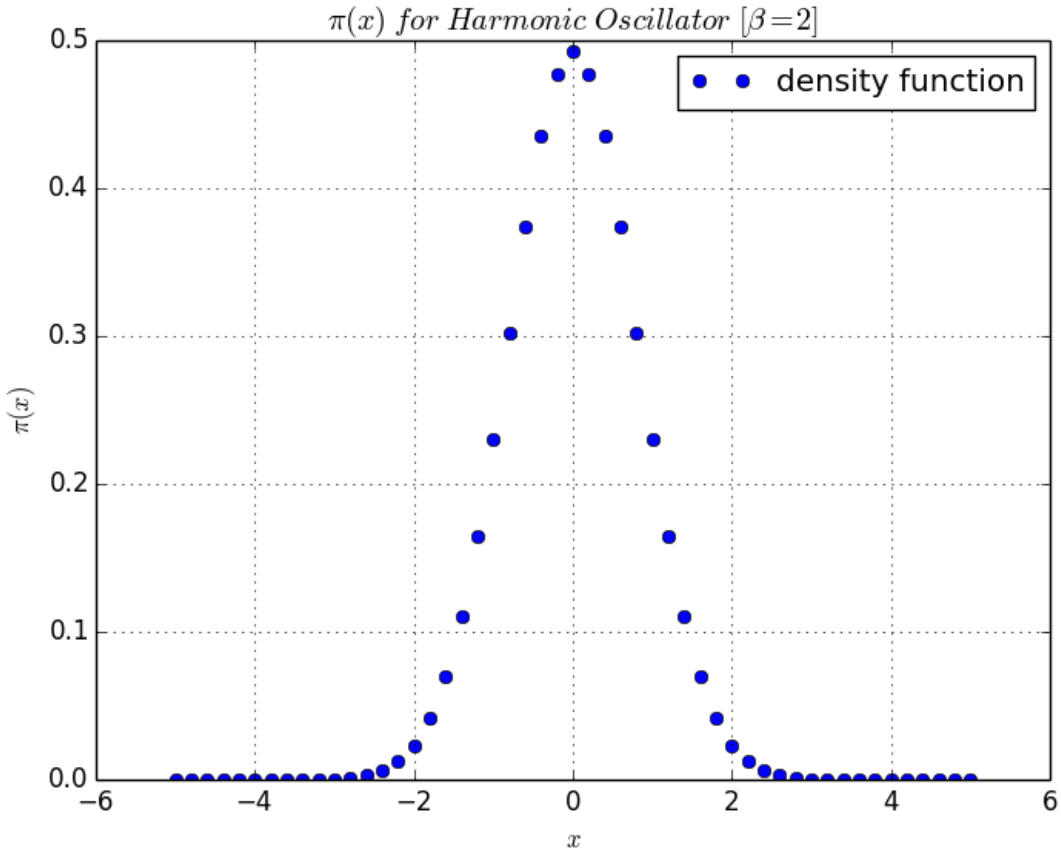
Za = 0.42545906412
Zb = 0.425459063622
Zc = 0.42545906412

2. The normalization constant is easily determined by realizing that
$\int_{-\infty}^{\infty} \pi(x)dx = const * \int_{-\infty}^{\infty} \rho(x, x, \beta)dx = const * Z(\beta).$

Since, $\int_{-\infty}^{\infty} \pi(x)dx = 1$, we get $const = \frac{1}{Z(\beta)}$.

Therefore, $const = \frac{1}{Zb} = 2.35040239003.$

The plot of Pi(x) is as follows for beta = 2:



3. The program code for parts 1 and 2 is:

```
import math, pylab

n_states = 40
Energies = [0.5 + i for i in range(n_states)]
grid_x = [i * 0.2 for i in range(-25, 26)]
psi = {}
for x in grid_x:
    psi[x] = [math.exp(-x ** 2 / 2.0) / math.pi ** 0.25]
    psi[x].append(math.sqrt(2.0) * x * psi[x][0])
    for n in range(2, n_states+1):
        psi[x].append(math.sqrt(2.0 / n) * x * psi[x][n - 1] - math.sqrt((n - 1.0) / n)
 * psi[x][n - 2])

beta = 2

#Part 1
#Method a
Za = sum(math.exp(-beta * (n+0.5)) for n in range(n_states))
print 'Za =', Za

#Method b
Zb = 0
for n in range(n_states):
    sum = 0
    # Use trapezoidal rule to find the integral
    for x in grid_x:
        sum += psi[x][n+1]**2+psi[x][n]**2

    psi_sq = (0.2/2) * sum
    Zb += math.exp(-beta * (n+0.5)) * psi_sq
print 'Zb =', Zb

#Method c
Zc = 1/(2*math.sinh(beta/2.0))
print 'Zc =', Zc

#Part 2
constant = 1/Zb
print 'constant =', constant

pi_list = []
for x in grid_x:
    rho = 0
    for n in range(n_states):
        rho += math.exp(-beta * (n+0.5)) * psi[x][n] ** 2

    pi_list.append(constant * rho)

pylab.plot(grid_x, pi_list, 'o', label='density function')
pylab.title('$\pi(x)\ for\ Harmonic\ Oscillator\ [\\beta = 2]$')
pylab.xlabel('$x$')
```

```
pylab.ylabel("$\pi(x)$")
pylab.grid()
pylab.legend()
pylab.savefig("Probability.png")
pylab.show()
```

Evaluation/feedback on the above work

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

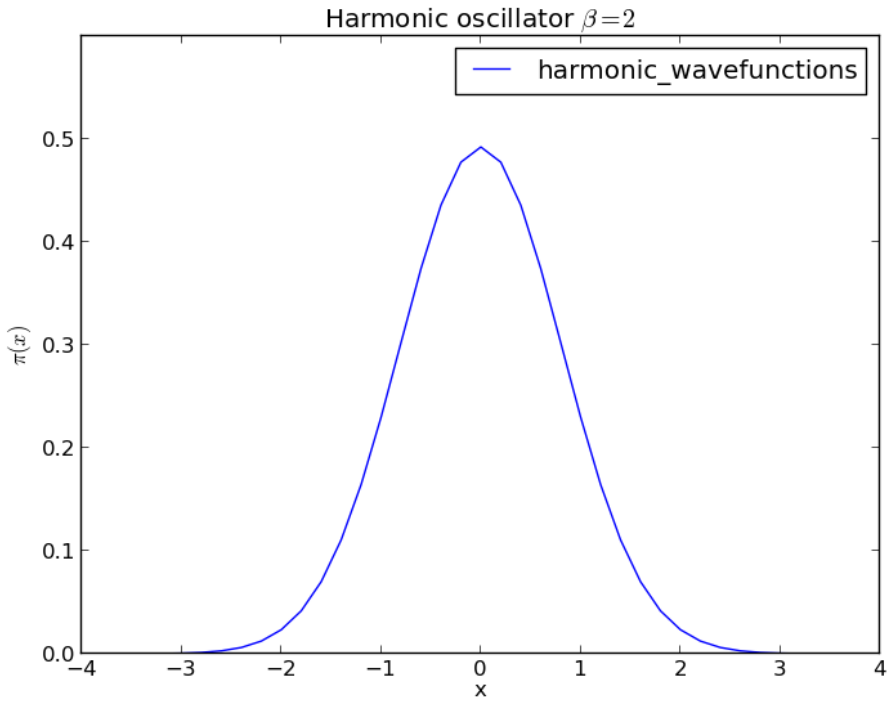
B1 Here, you are asked to evaluate your fellow student's understanding of the **harmonic wavefunction program**. There are 3 aspects (questions).

1/ The numerical values of Z, that should be close to 0.42
2/ The understanding that the normalization is by the partition function + fileupload
3/ Program is provided

An example of an answer that would receive full score is as follows:

1/ Numerical values
0.42545906412 Z from \sum_n exp(-beta E_n)
0.42545906412 Z (exact)
0.425459064001 Z (from trace over density matrix)

2/ Constant: The constant is equal to 1/Z. Here is the plot of \pi(x)



Here is the program

```
import math, pylab

n_states = 40
Energies = [0.5 + i for i in range(n_states)]
dx = 0.2
grid_x = [i * dx for i in range(-25, 26)]
psi = {}
for x in grid_x:
    psi[x] = [math.exp(-x ** 2 / 2.0) / math.pi ** 0.25]  # ground state
    psi[x].append(math.sqrt(2.0) * x * psi[x][0])         # first excite
d state
    # other excited states (through recursion):
    for n in range(2, n_states):
        psi[x].append(math.sqrt(2.0 / n) * x * psi[x][n - 1] -
                      math.sqrt((n - 1.0) / n) * psi[x][n - 2])
beta = 2.0
Z = sum(math.exp(-beta * Energies[k]) for k in range(n_states))
print beta, 'beta'
print Z, ' Z from \sum_n exp(-beta E_n)'
print 1.0 / (2.0 * math.sinh(beta/2)), ' Z (exact)'
Z_trace = 0.0
x_values = []
y_values = []
for x in grid_x:
    rho_xx = sum(math.exp(-beta * Energies[i]) * psi[x][i] ** 2 for i in
 range(n_states))
    Z_trace += rho_xx * dx
    x_values.append(x)
    y_values.append(rho_xx/Z)
print Z_trace, ' Z (from trace over density matrix)'
pylab.title('Harmonic oscillator $ \\beta = 2$ ')
pylab.xlabel('x')
pylab.ylabel('$\pi(x)$')
pylab.plot(x_values, y_values, label='harmonic_wavefunctions')
pylab.legend()
pylab.axis([-4.0, 4.0, 0.0, 0.6])
pylab.savefig('density_x_harm_wave.png')
pylab.show()
```

Aspects:

1/ The numerical values of Z, that should be close to 0.42.

2/ The understanding that the normalization is by the partition function + file upload.

3/ Correct program is provided.

POINTS - DESCRIPTION

Give 0 points if 0 aspect is treated correctly.

Give 1 points if 1 aspect is treated correctly.

Give 2 points if 2 aspects are treated correctly.

Give 3 points if 3 aspects are treated correctly.

Score from your peers: **2.5**

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

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

B2 Download (cut-and-paste) the below program (matrix_square_harmonic.py).

```
import math, numpy

def rho_free(x, xp, beta):
    return (math.exp(-(x - xp) ** 2 / (2.0 * beta)) /
            math.sqrt(2.0 * math.pi * beta))

def rho_harmonic_trotter(grid, beta):
    return numpy.array([[rho_free(x, xp, beta) * \
                         numpy.exp(-0.5 * beta * 0.5 * (x ** 2 + xp ** 2)) \
                         for x in grid] for xp in grid])

x_max = 5.0
nx = 100
dx = 2.0 * x_max / (nx - 1)
x = [i * dx for i in range(-(nx - 1) / 2, nx / 2 + 1)]
beta_tmp = 2.0 ** (-5)
beta     = 2.0 ** (3)
rho = rho_harmonic_trotter(x, beta_tmp)
while beta_tmp < beta:
    rho = numpy.dot(rho, rho)
    rho *= dx
    beta_tmp *= 2.0
```

Modify it (using the technique of section **A2**) so that it computes the diagonal density matrix and the partition function, using the two methods:

a) Z(beta) = \int dx rho(x, x, beta) (general, any quantum system)

b) Z(beta) = Z^harm (beta) = 1/(2 sinh(beta/2)) (harmonic oscillator only)
(see Section **A3**)

NB: This requires just a few-line change in the program.

Then adress the following aspects:

1/ **Briefly explain** how the Trotter decomposition is implemented in this program. Also, briefly discuss the choice of the initial value of beta_tmp.

2/ **Compute the partition function** using the two methods for beta between 1/16 and 8. **Communicate your results**. Comment on the precision obtained. **NOTE ADDED** 03/11/2014 - **Nothing complicated here, just look into A2**

3/ **Modify your program** so that it outputs the normalized pi(x) (normalization from your data, NOT from an analytic formula). **Produce a graphics file.** Use labels for plots, as in Section A3 (titles, axis labels, graph labels). **Compare (on the same plot) with the exact formula:**

pi^exact (x) = (\sqrt(tanh (beta/2))) / sqrt(pi) exp(- x^2 tanh(beta/2))

Upload this graphics file for temperature beta = 8. **Explain** which starting value of beta_tmp appears ok for you.
4/ **Upload your program.**

1. Trotter decomposition is as follows (valid for $\beta \to 0$):

$\rho(x, x', \beta) \to e^{-\frac{1}{2}\beta V(x)}\ \rho^{free}(x, x', \beta)\ e^{-\frac{1}{2}\beta V(x')}$

$\rho(x, x', \beta)$ is implemented by **rho_harmonic_trotter** function which in turn calls **rho_free** function to calculate $\rho^{free}(x, x', \beta)$.

Next, we use the convolution property of density matrix i.e.
$\int dx'\ \rho(x, x', \beta_1)\ \rho(x', x'', \beta_2) = \rho(x', x'', \beta_1 + \beta_2)$

If the two temperatures are same, the convolution becomes product of the density matrix with itself and results in:
$\int dx'\ \rho(x, x', \beta)\ \rho(x', x'', \beta) = \rho(x', x'', 2\beta)$

This is done in the following code segment:

```
rho = numpy.dot(rho, rho)
rho *= dx
```

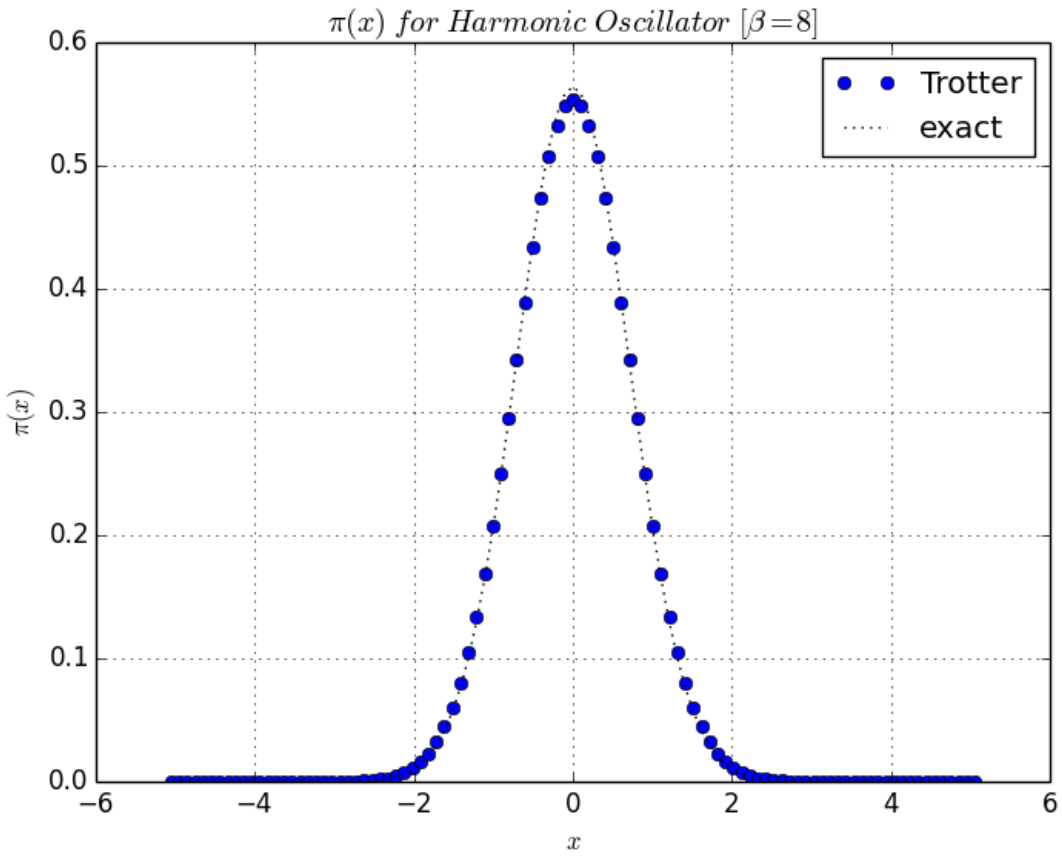
The resulting ρ is for the β which is twice than the starting β. Since $\beta = \frac{1}{32}$ to start with, it becomes $\frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \ldots$ and we can get the density matrices for higher β in a sequential manner .

Trotter decomposition is valid for high temperatures i.e. $\beta \to 0$. Therefore, we need a low value of beta to start with. This is why we start with $\beta = \frac{1}{32}$

2. The table of partition functions is as below: Z_a and Z_b refer to the partition function obtained by methods (a) and (b) respectively. The precision improves with increasing beta.

Beta	Z_a	Z_b
0.03125	20.0969349307	31.9986979538
0.0625	12.6470833924	15.99739613
0.125	7.36329820503	7.99479403928
0.25	3.93131100767	3.98960229081
0.5	1.9778343624	1.97931758165
1.0	0.959553704626	0.959517375667
2.0	0.425481790325	0.42545906412
4.0	0.137871919356	0.137860282386

3. Pi(x) obtained by Trotter method and direct formula for beta=8 is shown in plot below:



As explained in (1), Trotter decomposition is valid for high temperatures i.e. $\beta \to 0$. Therefore, we need a low value of beta_tmp to start with. In this program, we start with the value of 1/32. This leads to excellent agreement with the analytical calculations at higher β . If we choose, a higher beta_tmp, the agreement might not be good for $\beta = 8$.

4. The program is as follows:

```
import math, numpy, pylab

def rho_free(x, xp, beta):
    return (math.exp(-(x - xp) ** 2 / (2.0 * beta)) /
            math.sqrt(2.0 * math.pi * beta))

def rho_harmonic_trotter(grid, beta):
    return numpy.array([[rho_free(x, xp, beta) * \
                         numpy.exp(-0.5 * beta * 0.5 * (x ** 2 + xp ** 2)) \
                         for x in grid] for xp in grid])

x_max = 5.0
nx = 100
dx = 2.0 * x_max / (nx - 1)
grid_x = [i * dx for i in range(-(nx - 1) / 2, nx / 2 + 1)]
beta_tmp = 2.0 ** (-5)
beta     = 2.0 ** (3)
rho = rho_harmonic_trotter(grid_x, beta_tmp)
print 'Beta \t Z_a \t Z_b'

while beta_tmp < beta:
    rho_x_x = numpy.diag(rho)
    sum_rho_x_x = numpy.diag(rho_x_x).sum()

    # Part 1
    #Use trapezoidal rule to calculate the integral from density
    Z_a = (2*sum_rho_x_x - rho_x_x[0] - rho_x_x[-1])*dx/2

    #Use analytical formula
    Z_b = 1/(2*math.sinh(beta_tmp/2))

    rho = numpy.dot(rho, rho)
    rho *= dx
    beta_tmp *= 2.0
    print beta_tmp,'\t',Z_a, '\t', Z_b

# Part 2
# Last iteration stored the density (rho_x_x) for beta = 8

constant = 1/Z_a
print 'constant =', constant

normalized_rho_x_x = rho_x_x * constant

pi_list_exact = []
for x in grid_x:
    pi_list_exact.append(math.sqrt(math.tanh(beta/2.0)/math.pi)*\
    math.exp(-x**2 * math.tanh(beta/2.0)))

pylab.plot(grid_x, normalized_rho_x_x, 'o', label='Trotter')
pylab.plot(grid_x, pi_list_exact, ':k', label='exact')
```

```
pylab.title('$\pi(x)\ for\ Harmonic\ Oscillator\ [\\beta = 8]$')
pylab.xlabel('$x$')
pylab.ylabel('$\pi(x)$')
pylab.legend()
pylab.grid()
pylab.savefig("Probability_trotter.png")
pylab.show()
```

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 understanding of **Matrix squaring.**

There are 4 aspects (questions).

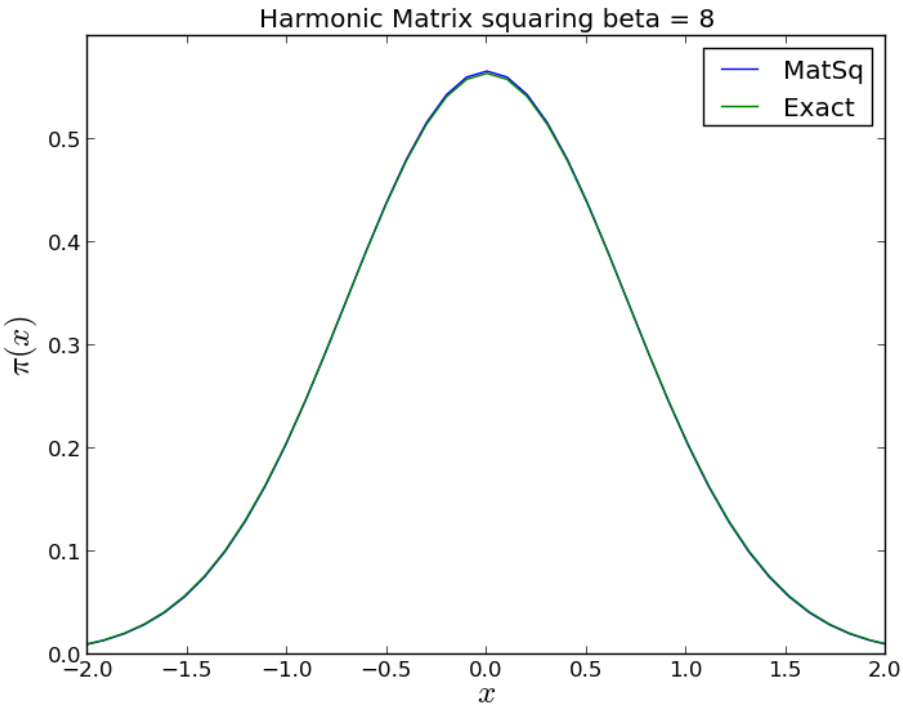
1/ Understanding of the Trotter decomposition
2/ Calculation of the partition function
3/ Graphics file of normalized probability
4/ Program

An example of an answer that would receive full score is as follows:

1/ The Trotter decomposition is used only at the initilisation step, as exp(-beta_tmp V(x)/2) rho_free(x, x', beta_tmp), exp(-beta_tmp V(x'), where beta_tmp is the initial inverse temperature.
3/ calculation of normalized density
4/ upload of program

beta Z (from trace) Z(exact)
0.0625 12.69564712 15.99739613
0.125 7.37274957677 7.99479403928
0.25 3.93217671108 3.98960229081
0.5 1.97785690853 1.97931758165
1.0 0.959553785845 0.959517375667
2.0 0.425481790366 0.42545906412
4.0 0.137871919356 0.137860282386
8.0 0.0183247691398 0.0183217851629

Here is the graphics file: An initial value of beta = 1/4 gives good results (but beta = 1/8 or even smaller ... or beta = 1/2 are OK, too)



Here is the program

```
import math, numpy, pylab

def rho_free(x, xp, beta):
    return (math.exp(-(x - xp) ** 2 / (2.0 * beta)) /
            math.sqrt(2.0 * math.pi * beta))

def rho_harmonic_trotter(grid, beta):
    return numpy.array([[rho_free(x, xp, beta) * \
                         numpy.exp(-0.5 * beta * 0.5 * (x ** 2 + xp ** 2
)) \
                         for x in grid] for xp in grid])

x_max = 5.0
nx = 100
dx = 2.0 * x_max / (nx - 1)
x = [i * dx for i in range(-(nx - 1) / 2, nx / 2 + 1)]
beta_initial = 2.0 ** (-2)
beta_tmp = beta_initial
beta     = 2.0 ** (3)
rho = rho_harmonic_trotter(x, beta_tmp)
while beta_tmp < beta:
    rho = numpy.dot(rho, rho)
    rho *= dx
    beta_tmp *= 2.0
    Z_trace = dx * (numpy.diag(rho)).sum()
    print beta_tmp, Z_trace, 1.0 / (2.0 * math.sinh(beta_tmp / 2.0))

y_values = numpy.diag(rho) / Z_trace

pylab.plot(x, y_values, label= 'MatSq')
y_values = [math.sqrt(math.tanh(beta_tmp / 2.0)) / math.sqrt(math.pi) *
\
                  math.exp( - xx **2 * math.tanh( beta_tmp / 2.0)) for x
x in x]

pylab.plot(x, y_values, label= 'Exact')
pylab.axis([-2.0, 2.0, 0.0, 0.8])
pylab.title('Harmonic Matrix squaring beta = 8')
pylab.xlabel('$x$', fontsize=18)
pylab.ylabel('$\\pi(x)$', fontsize=18)
pylab.axis([-2.0, 2.0, 0.0, 0.6])
pylab.legend()
pylab.savefig('Matrix_square_harmonic_8.png')
pylab.show()
```

POINTS - DESCRIPTION

Give 0 points if 0 aspects are treated correctly

Give 1 points if 1 - 2 aspects are treated correctly

Give 2 points if 3 - 4 aspects are treated correctly

Score from your peers: **1.5**

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

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

B3 Download (cut-and-paste) the below program (naive_harmonic_path.py). Modify it so that it generates a histogram of positions x. Attention: use the "if step % separation == 0: " construction to avoid data overflow in your computer.

```
import math, random

def rho_free(x, y, beta):
    return math.exp(-(x - y) ** 2 / (2.0 * beta))

beta = 8.0
N = 8
dtau = beta / N
delta = 1.0
n_steps = 100000
x = [0.0] * N
for step in range(n_steps):
    k = random.randint(0, N - 1)
    knext, kprev = (k + 1) % N, (k - 1) % N
    x_new = x[k] + random.uniform(-delta, delta)
    old_weight  = (rho_free(x[knext], x[k], dtau) *
                   rho_free(x[k], x[kprev], dtau) *
                   math.exp(-0.5 * dtau * x[k] ** 2))
    new_weight  = (rho_free(x[knext], x_new, dtau) *
                   rho_free(x_new, x[kprev], dtau) *
                   math.exp(-0.5 * dtau * x_new ** 2))
    if random.uniform(0.0, 1.0) < new_weight / old_weight:
        x[k] = x_new
    print x
```

Then address the following aspects:

1/ By moving from one configuration to another, does this algorithm use the Trotter decomposition? Explain in a few words the "factors of two" in old_weight and new_weight (wouldn't one expect a "0.25 dtau * ...", rather that the "0.5 * ... "?

2/ Does this program compute the partition function?

3/ Does this program compute the off-diagonal density matrix?

4/ **Produce** a histogram for pi(x) at inverse temperature beta = 2. Compare (on the same plot) with the exact formula:

pi^exact (x) = \sqrt(tanh (beta/2)) / \sqrt(pi) exp(- x^2 tanh(beta/2))

NB an error was corrected here on 03/09/2014, thanks to a remark on the forum.

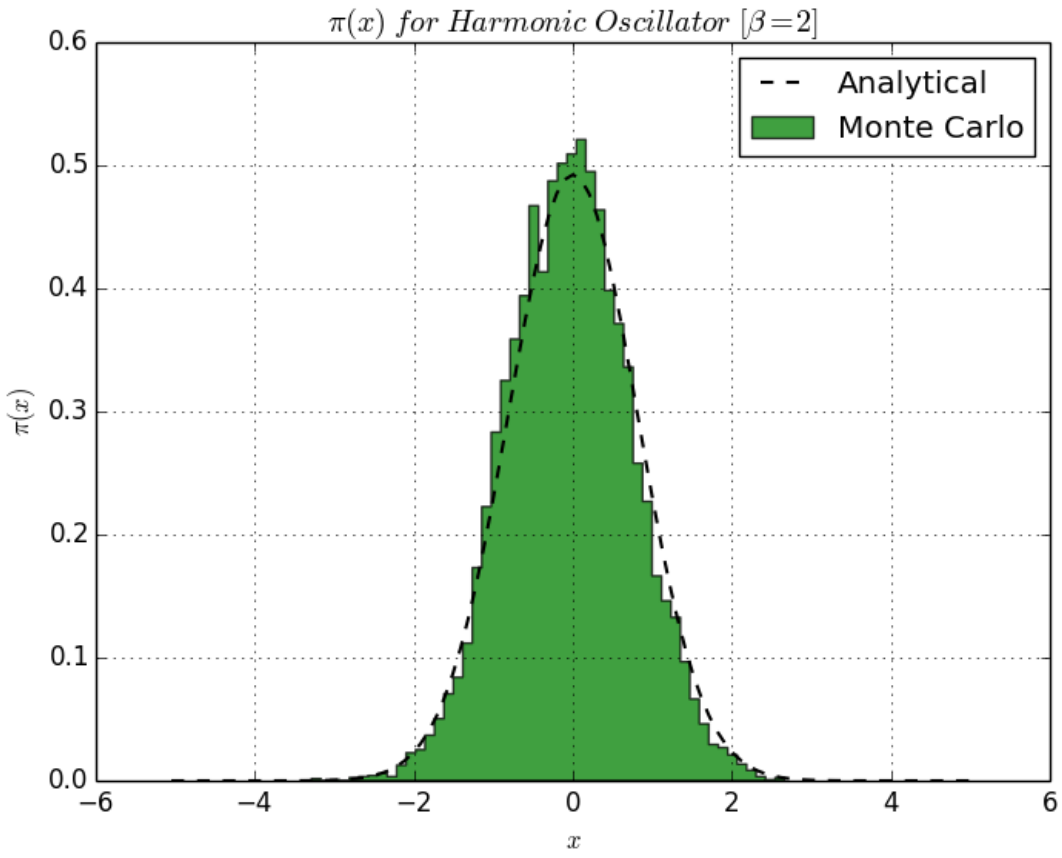
5/ **Upload your program**. Specify the value of dtau you used in your code.

1. The program uses Trotter decomposition to move from one configuration to other. The factor of two comes from the fact that x[k] comes twice (once with x[knext] and once with x[kprev]) in the expression. Therefore, we need to double the contribution resulting in "0.5 dtau * .. " and not "0.25 dtau * ... " as one might expect.

2. No. it does not compute the partition function explicitly. It does compute the diagonal density matrix from which we can obtain the partition function.

3. It does not compute the off-diagonal density matrix.

4. Histogram for beta=2 and the exact pi(x) are shown together in the plot below. The histograms are plotted with "histfilled" option of pyplot for better visualization.



5. The code is as follows:

```
import math, random, pylab

def rho_free(x, y, beta):
    return math.exp(-(x - y) ** 2 / (2.0 * beta))

beta = 2.0
N = 8
dtau = beta / N
delta = 1.0
n_steps = 100000
x = [0.0] * N
x_list = []
for step in range(n_steps):
    k = random.randint(0, N - 1)
    knext, kprev = (k + 1) % N, (k - 1) % N
    x_new = x[k] + random.uniform(-delta, delta)
    old_weight  = (rho_free(x[knext], x[k], dtau) *
                   rho_free(x[k], x[kprev], dtau) *
                   math.exp(-0.5 * dtau * x[k] ** 2))
    new_weight  = (rho_free(x[knext], x_new, dtau) *
                   rho_free(x_new, x[kprev], dtau) *
                   math.exp(-0.5 * dtau * x_new ** 2))
    if random.uniform(0.0, 1.0) < new_weight / old_weight:
        x[k] = x_new
    if step % 10 == 0:
        k = random.randint(0, N - 1)
        x_list.append(x[k])
    print x

#Generate the analytical equation data
x_max = 5.0
nx = 100
dx = 2.0 * x_max / (nx - 1)
grid_x = [i * dx for i in range(-(nx - 1) / 2, nx / 2 + 1)]

pi_list_exact = []
for x in grid_x:
    pi_list_exact.append(math.sqrt(math.tanh(beta/2.0)/math.pi) * \
                         math.exp(-x**2 * math.tanh(beta/2.0)))
pylab.plot(grid_x, pi_list_exact, 'k--', label='Analytical', linewidth=1.5)

n, bins, patches=pylab.hist(x_list, bins=50, normed=1, histtype='stepfilled', label='Mo
nte Carlo')
pylab.setp(patches, 'facecolor', 'g', 'alpha', 0.75)
pylab.title('$\pi(x)\ for\ Harmonic\ Oscillator\ [\\beta = 2]$')
pylab.xlabel('$x$')
pylab.ylabel('$\pi(x)$')
pylab.legend()
pylab.grid()
pylab.savefig("Probability_MC.png")
pylab.show()
```

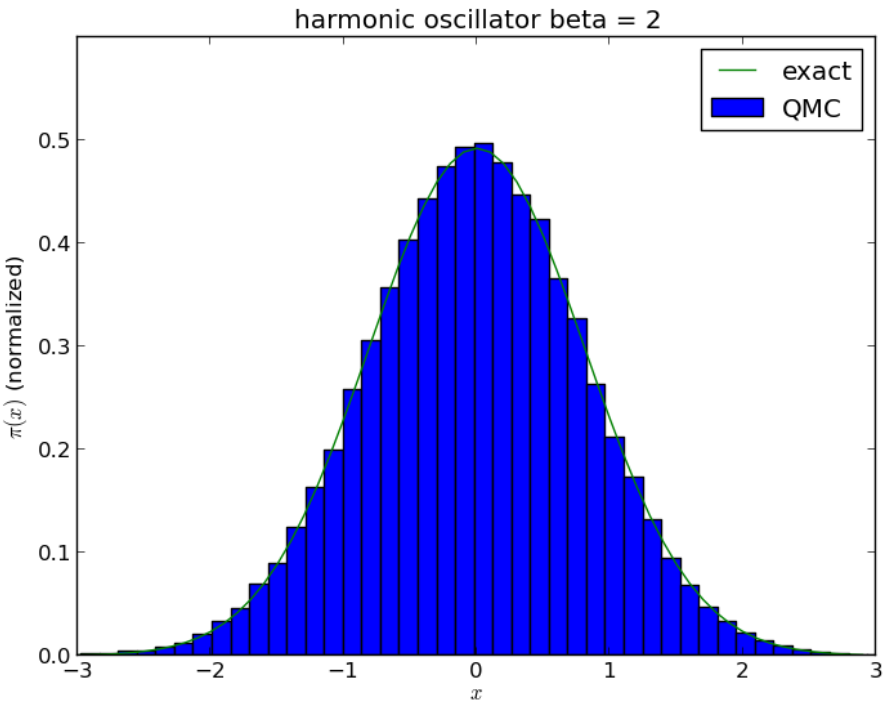
beta = 2.0, N = 8. Therefore, dtau = beta/N = 2.0/8 = 0.25

Evaluation/feedback on the above work

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

B3 Here you evaluate whether a **Quantum Monte Carlo simulation** was set up correctly, in a case that was already treated in the lecture.

An example of an answer that would receive full score is as follows:
1/ in rho_free(x_k-1, x_k, beta/N) rho_free(x_k, x_k+1, beta/N), there are two factors exp(-beta x^2/4) that combine to exp(-beta x^2/2)
2/ No
3/ No
4/ Histogram and comparison with exact solution as shown below (dtau = 1/4)
5/ Program as shown below



```
import math, random, pylab

def rho_free(x, y, beta):
    return math.exp(-(x - y) ** 2 / (2.0 * beta))

beta = 2.0
N = 8
dtau = beta / N
delta = 1.0
n_steps = 10000000
x = [0.0] * N
data = []
for step in range(n_steps):
    k = random.randint(0, N - 1)
    knext, kprev = (k + 1) % N, (k - 1) % N
    x_new = x[k] + random.uniform(-delta, delta)
    old_weight  = (rho_free(x[knext], x[k], dtau) *
                   rho_free(x[k], x[kprev], dtau) *
                   math.exp(-0.5 * dtau * x[k] ** 2))
    new_weight  = (rho_free(x[knext], x_new, dtau) *
                   rho_free(x_new, x[kprev], dtau) *
                   math.exp(-0.5 * dtau * x_new ** 2))
    if random.uniform(0.0, 1.0) < new_weight / old_weight:
        x[k] = x_new
    if step%100 == 0:
        k = random.randint(0, N - 1)
        data.append(x[k])
pylab.hist(data, 50, normed = True, label='QMC')
x_values = [0.1 * a for a in range (-30,30)]
y_values = [math.sqrt(math.tanh(beta / 2.0)) / math.sqrt(math.pi) * \
                  math.exp( - xx **2 * math.tanh( beta / 2.0)) for xx in
 x_values]
pylab.title('harmonic oscillator beta = 2')
pylab.xlabel('$x$')
pylab.ylabel('$\\pi(x)$ (normalized)')
pylab.axis([-3.0, 3.0, 0.0, 0.6])
pylab.plot(x_values, y_values,label= 'exact')
pylab.legend()
pylab.savefig('harmonic_beta_2.png')
pylab.show()
```

1/ in rho_free(x_k-1, x_k, beta/N) rho_free(x_k, x_k+1, beta/N), there are two factors exp(-beta x^2/4) that combine to exp(-beta x^2/2)
2/ No
3/ No
4/ Histogram and comparison with exact solution as shown below
5/ Program as shown below

POINTS - DESCRIPTION
Cumulative:
Aspects 1,2,3 give together 1 point

Histogram and comparison give 1 point
Correct program gives 1 point

Score from your peers: **2.5**

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

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

C

In part C, you will modify your program of section 2 for the anharmonic oscillator. The key methods are now matrix squaring and Quantum Monte Carlo, although you will do some comparison with the approximate partition function computed using the Preparation program 3.

C1 Start a new program for matrix squaring from matrix_square_harmonic_movie.py (see section B2).

1/ Incorporate the function V(x) from **Preparation program 3**.
2/ Modify the function rho_harmonic_trotter into a rho_trotter, which explicitly uses V(x). Attention: There is a "one half" from the Trotter decomposition on both sides of the free density matrix.
3/ Implement the function Z from **Preparation program 3**.
4/ Check your program (including the function Z) for cubic = 0 and quartic = 0 against the version of section B.

Then:
Run the program for the anharmonic oscillator, for different values of cubic < 0
and quartic > 0, and compute the partition function from

a) Z^approx (beta) from the function that you just incorporated
b) Z(beta) = \int dx rho(x, x, beta) (general, any quantum system)
c) Z(beta) = Z^harm (beta) = 1/(2 sinh(beta/2)) (comparison with the harmonic oscillator only)
(see Section A3)

Then adress the following aspects: for cubic = - quartic = 0.1, 0.2...

1/ Compute the partition function from the trace of partition function and compare with the approximate perturbation-theoretical result. **Communicate your results for beta = 2** for cubic, quartic = (-0.1, 0.1) (-0.2, 0.2), (-0.4,0.4), (-0,8, 0.8). Short comment.

2/ Compute the normalized probability to be at position x for cubic = -0.5, quartic = 0.5.
pi(x) = const rho(x, x, beta)
Upload this graphics file for temperature beta = 2.
3/ **Upload your program.**

1. The table is below. Z_a, Z_b and Z_c refer to the partition functions obtained using methods a, b and c listed above. We set n_max = 10 while calculating Z_a to avoid the overflow error for higher values of

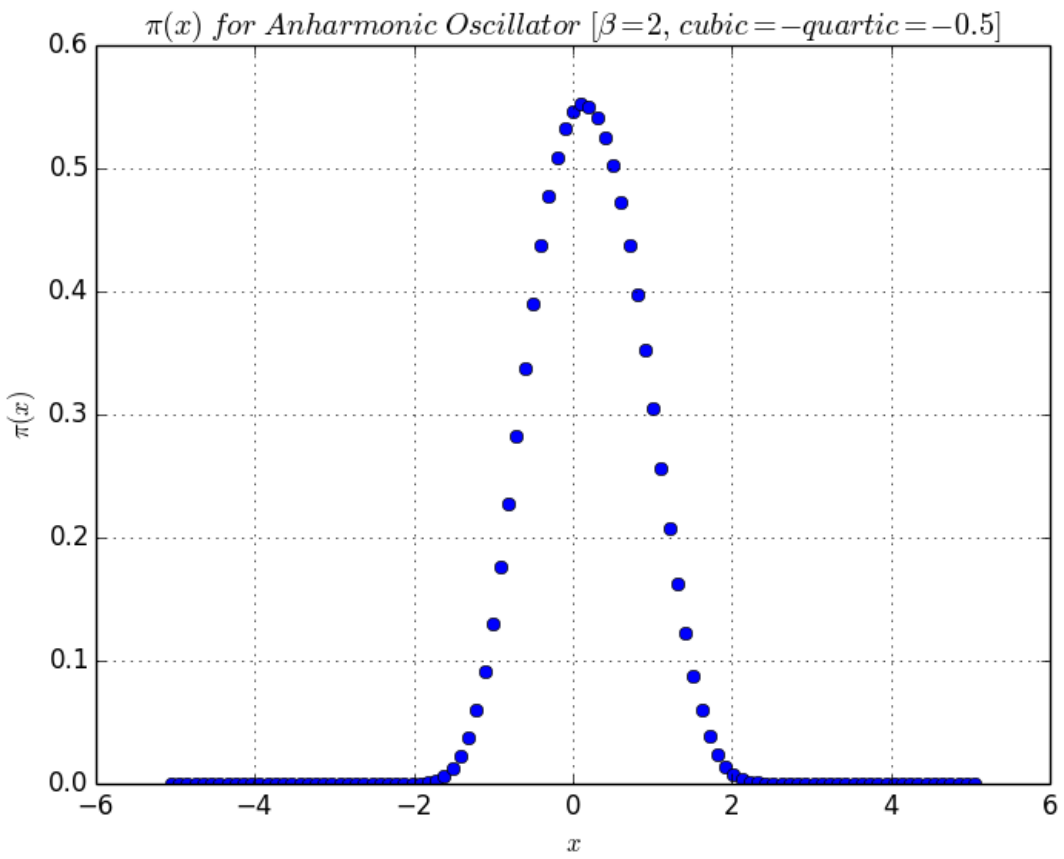
cubic and quartic.

cubic	quartic	Z_a	Z_b	Z_c
0.0	**0.0**	**0.959501350095**	**0.959553704626**	**0.959517375667** [TEST CASE]
-0.1	**0.1**	**0.786003667501**	**0.807831983004**	**0.959517375667**
-0.2	**0.2**	**0.737315742714**	**0.743809182495**	**0.959517375667**
-0.3	0.3	0.75518402287	0.702623293193	0.959517375667
-0.4	**0.4**	**0.885731380653**	**0.67249743639**	**0.959517375667**
-0.5	0.5	25907.9700373	0.648935777761	0.959517375667
-0.6	0.6	9.06095203565e+16	0.629723511653	0.959517375667
-0.7	0.7	1.33434738746e+33	0.613603026296	0.959517375667
-0.8	**0.8**	**7.73360535354e+52**	**0.599791068716**	**0.959517375667**

Z_a estimate to the partition function starts to diverge from cubic = -quartic = -0.5 onwards. As explained in **A3.4**, the perturbation approximation ceases to be valid for this region of cubic and quartic values. E_n becomes negative with increasing n, which leads to blowing up of the exponential term in the partition sum expression of Z_a which is $Z = \sum_{n=0}^{n=n_{max}} e^{-\beta E_n}$

For n_max = 10, we see that Z_a becomes inaccurate when the magnitude of cubic and quartic exceeds 0.4. The results of Z_a, Z_b and Z_c match for cubic = quartic = 0.0. Z_a and Z_b tend to agree for small values of quartic and cubic.

2. Normalized probability distribution for cubic = -0.5, quartic = 0.5 and beta = 2 using density function.



3. The program code is below:

```
import math, numpy, pylab

# V(x) function from preparation program 3
def V(x):
    pot =  x ** 2 / 2 + cubic * x ** 3 + quartic * x ** 4
    return pot

# Z function from preparation program 3
def Z(cubic, quartic, beta, n_max):
    Z = sum(math.exp(-beta * Energy(n, cubic, quartic)) for n in range(n_max + 1))
    return Z

def Energy(n, cubic, quartic):
    return n + 0.5 - 15.0 / 4.0 * cubic **2 * (n ** 2 + n + 11.0 / 30.0) + 3.0 / 2.0\
           * quartic * (n ** 2 + n + 1.0 / 2.0)

def rho_free(x, xp, beta):
    return (math.exp(-(x - xp) ** 2 / (2.0 * beta)) / math.sqrt(2.0 * math.pi * beta))

def rho_aharmonic_trotter(grid, beta):
    return numpy.array([[rho_free(x, xp, beta) * numpy.exp(-0.5 * beta * (V(x) + V(xp))
)\
           for x in grid] for xp in grid])

x_max = 5.0
nx = 100
dx = 2.0 * x_max / (nx - 1)
grid_x = [i * dx for i in range(-(nx - 1) / 2, nx / 2 + 1)]
beta_tmp = 2.0 ** (-5)
beta     = 2.0

print 'cubic \t quartic \t Z_a \t Z_b \t Z_c'

for int_cubic in range(0, 9):
    cubic = -0.1*int_cubic
    quartic = -1*cubic
    beta_tmp = 2.0 ** (-5)
    rho = rho_aharmonic_trotter(grid_x, beta_tmp)

    while beta_tmp < beta:
        rho_x_x = numpy.diag(rho)
        sum_rho_x_x = numpy.diag(rho_x_x).sum()

        #a. Use Z to calculate Partition function
        Z_a = Z(cubic, quartic, beta_tmp, 10)

        #b. Use trapezoidal rule to calculate the integral from density
        Z_b = (2*sum_rho_x_x - rho_x_x[0] - rho_x_x[-1])*dx/2

        #c. Use analytical formula for comparison with simple harmonic oscillator
        Z_c = 1/(2*math.sinh(beta_tmp/2))
```

```
        rho = numpy.dot(rho, rho)
        rho *= dx
        beta_tmp *= 2.0

#Store results for cubic = quartic = -0.5
if int_cubic == 5:
        rho_x_x_qc_0p5 = rho_x_x
        Z_b_qc_0p5 = Z_b

    print cubic, '\t', quartic, '\t', Z_a, '\t', Z_b, '\t', Z_c

# Part 2
constant = 1/Z_b_qc_0p5
print 'constant =', constant

normalized_rho_x_x = rho_x_x_qc_0p5 * constant

pylab.plot(grid_x, normalized_rho_x_x, 'o')
pylab.title('$\pi(x)\ for\ Anharmonic\ Oscillator\ [\\beta = 2,\ cubic = -quatric = -0.
5]$')
pylab.xlabel('$x$')
pylab.ylabel('$\pi(x)$')
pylab.grid()
pylab.savefig("Probability_trotter2.png")
pylab.show()
```

Evaluation/feedback on the above work

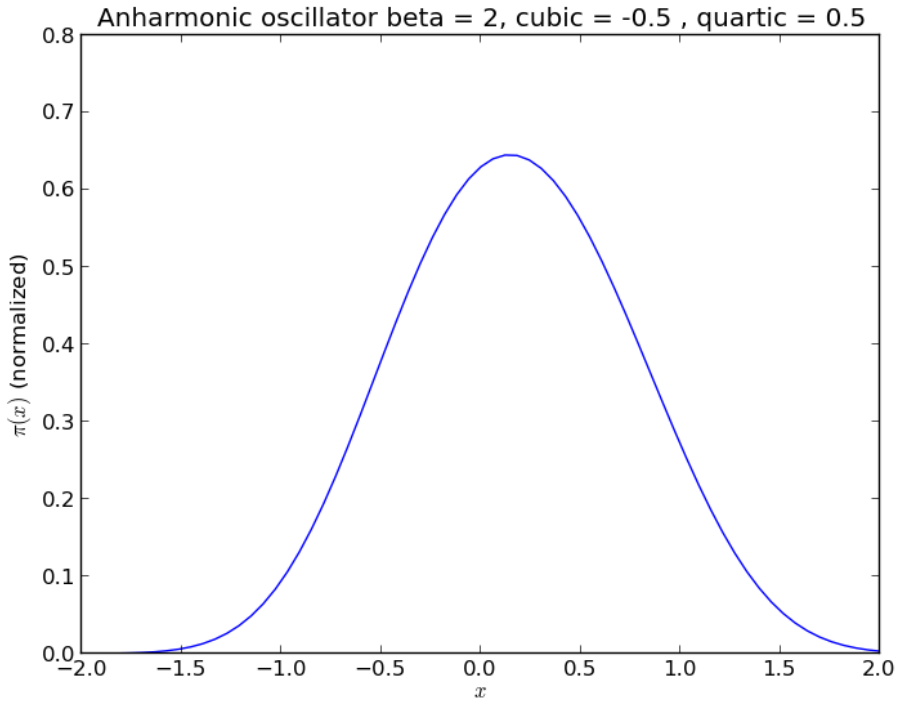
Note: this section can only be filled out during the evaluation phase.

C1 Here you check that **matrix squaring** has been implemented correctly. There is really only one complicated point here:
The Trotter term is math.exp(-0.5 * beta * (V(x) + V(xp))), or equivalent.

NOTE ADDED 03/18/2014: By mistake, we used cubic = -1, quartic = 1 in the solution text, rather than cubic = -0.5, quartic = 0.5.
The central density should go up to a value between 0.6 and 0.7. Below is the corrected figure.

There are three aspects:
1/ calculation of the partition function: good agreement of exact solution with perturbation theory for small perturbation, nonsensical results of perturbation theory for large perturbation.
2/ Normalized probability, that should go up about to 0.65 **(NOTE ADDED 03/18/2014: corrected from earlier wrong value : 0.72)**
3/ Correct Program

An example of an answer that would receive full score is as follows:



Here data for z_trace, perturbation, harmonic result...

- (-0.1, 0.1) 2.0 0.363187108183 0.355151471602 0.42545906412
- (-0.2, 0.2) 2.0 0.331954730409 0.327758771651 0.42545906412
- (-0.4, 0.4) 2.0 0.295045349617 0.362552298949 0.42542
- (-0.8, 0.8) 0.8 2.0 0.255923398585 6.29822026272e+85 0.42545906412

```
import math, numpy, pylab

def V(x):
    pot =  x ** 2 / 2 + cubic * x ** 3 + quartic * x ** 4
    return pot

def rho_free(x, xp, beta):
    return (math.exp(-(x - xp) ** 2 / (2.0 * beta)) /
            math.sqrt(2.0 * math.pi * beta))

def rho_trotter(grid, beta):
    return numpy.array([[rho_free(x, xp, beta) * \
                         math.exp(-0.5 * beta * (V(x) + V(xp))) \
                         for x in grid] for xp in grid])

def Energy(n, a, b):
    return n + 0.5 - 15.0 / 4.0 * a **2 * (n **2 + n + 11.0 / 30.0) \
          + 3.0 / 2.0 * b * (n **2 + n + 1.0 / 2.0)

cubic = -0.8
quartic = 0.8
x_max = 3.0
nx = 100
dx = 2.0 * x_max / (nx - 1)
x = [i * dx for i in range(-(nx - 1) / 2, nx / 2 + 1)]
beta_tmp = 2.0 ** (-5)
beta     = 2.0
rho = rho_trotter(x, beta_tmp)
while beta_tmp < beta:
    rho = numpy.dot(rho, rho)
    rho *= dx
    beta_tmp *= 2.0
    print 'beta: %s -> %s' % (beta_tmp / 2.0, beta_tmp)
    Z_trace = dx * (numpy.diag(rho)).sum()
#    Z_anharm = sum(math.exp(-beta_tmp*Energy(n, cubic, quartic)) for n
in range(10))
#    print beta_tmp, Z_trace, Z_anharm, 1.0 / (2.0 * math.sinh( beta_tmp
/2.0))

y_values = numpy.diag(rho) / Z_trace
pylab.plot(x, y_values)

f = open('matrix_squaring_beta_2.txt', 'w')
for k in range(len(x)):
    f.write(str(x[k]) + ' ' + str(y_values[k]) + '\n')
f.close()
pylab.axis([-2.0, 2.0, 0.0, 0.8])
pylab.title('Anharmonic oscillator beta = 2, cubic = -1 , quartic = 1')
pylab.xlabel('$x$')
pylab.ylabel('$\\pi(x)$ (normalized)')
pylab.savefig('anharmonic_matrix_square.png')
```

```
pylab.show()
```

POINTS - DESCRIPTION
Give 0 points if 0 aspects are treated correctly
Give 1 points if 1aspect is treated correctly
Give 2 points if 2 aspects are treated correctly
Give 3 points if 3 aspects are treated correctly

Score from your peers: **2.5**

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

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

C2 Start a new Path-integral Monte Carlo program (see section **B3**). Incorporate V(x) as in section **C1**, then run it an produce the normalized
probability pi(x) at beta = 2. **Compare** to the output of **C1**. **To do so, plot the results of matrix squaring, if possible together with the Monte Carlo results in one file**. This will require you to write the matrix squaring results into a file and to recover them in the Monte Carlo program:

For writing into a file, you may use, for example,

```
f = open('matrix_squaring_beta_2.txt', 'w')
for k in range(len(x)):
    f.write(str(x[k]) + ' ' + str(y_values[k]) + '\n')
f.close()
```

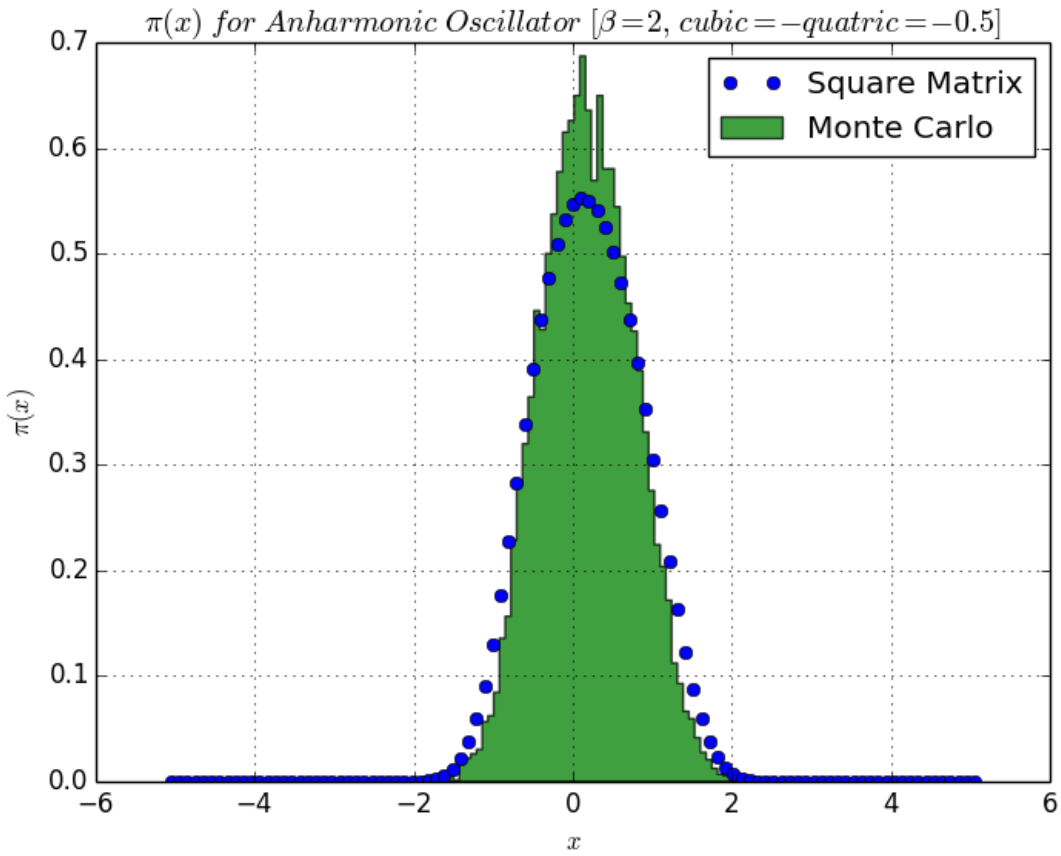
For reading, you may use, for example,

```
f = open(filename, 'r')
x = []
y = []
for line in f:
    a, b = line.split()
    x.append(float(a))
    y.append(float(b))
f.close()
```

Upload your program.

NB: In this one-dimensional problem, we can compare Quantum Monte Carlo with Matrix squaring, but already in three dimensions, Matrix squaring
would be utterly lost.
NNB: Setting up a Quantum Monte Carlo calculation as the present one is a considerable achievement.

The plot is as follows. There is a good agreement between the MC and square matrix (Trotter decomposition) approaches.



The program is below:

```
import math, random, pylab

# V(x) function from preparation program 3
def V(x):
    pot =  x ** 2 / 2 + cubic * x ** 3 + quartic * x ** 4
    return pot

def rho_free(x, y, beta):
    return math.exp(-(x - y) ** 2 / (2.0 * beta))

cubic = -0.5
quartic = 0.5

beta = 2.0
N = 8
dtau = beta / N
delta = 1.0
n_steps = 100000
x = [0.0] * N
x_list = []

for step in range(n_steps):
    k = random.randint(0, N - 1)
    knext, kprev = (k + 1) % N, (k - 1) % N
    x_new = x[k] + random.uniform(-delta, delta)
    old_weight  = (rho_free(x[knext], x[k], dtau) *
                   rho_free(x[k], x[kprev], dtau) *
                   math.exp(-1.0 * dtau * V(x[k])))
    new_weight  = (rho_free(x[knext], x_new, dtau) *
                   rho_free(x_new, x[kprev], dtau) *
                   math.exp(-1.0 * dtau * V(x_new)))
    if random.uniform(0.0, 1.0) < new_weight / old_weight:
        x[k] = x_new

    if step % 10 == 0:
        k = random.randint(0, N - 1)
        x_list.append(x[k])

    print x

filename = 'matrix_squaring_beta_2.txt'
f = open(filename, 'r')
x_sq_matrix = []
y_sq_matrix = []
for line in f:
    a, b = line.split()
    x_sq_matrix.append(float(a))
    y_sq_matrix.append(float(b))
f.close()

pylab.plot(x_sq_matrix, y_sq_matrix, 'o', label='Square Matrix', linewidth=1.5)
```

```
n, bins, patches=pylab.hist(x_list, bins=50, normed=1, histtype='stepfilled', label='Mo
nte Carlo')
pylab.setp(patches, 'facecolor', 'g', 'alpha', 0.75)
pylab.title('$\pi(x)\ for\ Anharmonic\ Oscillator\ [\\beta = 2,\ cubic = -quartic = -0.
5]$')
pylab.xlabel('$x$')
pylab.ylabel('$\pi(x)$')
pylab.legend()
pylab.grid()
pylab.savefig("Probability_MC2.png")
pylab.show()
```

Evaluation/feedback on the above work

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

C2 Here, the grand finale. You evaluate whether the **Quantum Monte Carlo program for the anharmonic oscillator** has been programmed correctly.

Three aspects:
1 Quantum Monte Carlo program (reading-in of Matrix squaring not required)
2 Plot produced (direct comparison with Matrix squaring not required)
3 Comparison **in one figure** of Quantum Monte Carlo and Matrix squaring

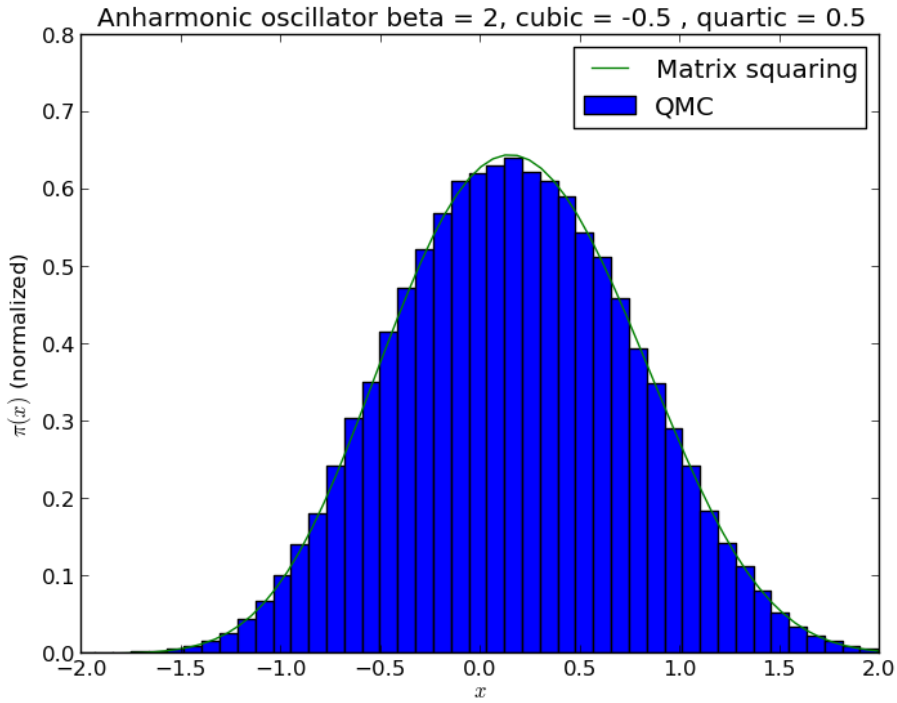
An example of an answer that would receive full score is as follows:

```
import math, random, pylab

def V(x):
    pot =  x ** 2 / 2 + cubic * x ** 3 + quartic * x ** 4
    return pot

def rho_free(x, y, beta):
    return math.exp(-(x - y) ** 2 / (2.0 * beta))

cubic = -1.0
quartic = 1.0
beta = 2.0
N = 8
dtau = beta / N
delta = 0.5
n_steps = 10000000
x = [0.0] * N
data = []
for step in range(n_steps):
    k = random.randint(0, N - 1)
    knext, kprev = (k + 1) % N, (k - 1) % N
    x_new = x[k] + random.uniform(-delta, delta)
    old_weight  = (rho_free(x[knext], x[k], dtau) *
                   rho_free(x[k], x[kprev], dtau) *
                   math.exp(-  dtau * V(x[k])))
    new_weight  = (rho_free(x[knext], x_new, dtau) *
                   rho_free(x_new, x[kprev], dtau) *
                   math.exp(- dtau * V(x_new)))
    if random.uniform(0.0, 1.0) < new_weight / old_weight:
        x[k] = x_new
    if step%100 == 0:
        k = random.randint(0, N - 1)
        data.append(x[k])
pylab.hist(data, 50, normed = True, label='QMC')
f = open('matrix_squaring_beta_2.txt', 'r')
x = []
y = []
for line in f:
    a, b = line.split()
    x.append(float(a))
    y.append(float(b))
f.close()
pylab.title('Anharmonic oscillator beta = 2, cubic = -1 , quartic = 1')
pylab.xlabel('$x$')
pylab.ylabel('$\\pi(x)$ (normalized)')
pylab.axis([-2.0, 2.0, 0.0, 0.8])
pylab.plot(x, y, label='Matrix squaring')
pylab.legend()
pylab.savefig('anharmonic_qmc_matrix.png')
pylab.show()
```

NOTE ADDED 03/18/2014: By mistake, we used cubic = -1, quartic = 1 in the

solution text, rather than cubic = -0.5, quartic = 0.5.

The central density should go up to between 0.6 and 0.7. Below is the corrected figure.



Three aspects:

1 Quantum Monte Carlo program (reading-in of Matrix squaring not required)
2 Plot produced (direct comparison with Matrix squaring not required)
3 Comparison in one figure of Quantum Monte Carlo and Matrix squaring

POINTS - DESCRIPTION
Give 0 points if 0 aspects are treated correctly.
Give 1 points if 1 aspect is treated correctly.
Give 2 points if 2 aspects are treated correctly.
Give 3 points if 3 aspects are treated correctly.

Score from your peers: **2.5**

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

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

Overall evaluation/feedback

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

peer 1 → very good !

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