Getting Started with Content Library

Thursday, June 9, 2022 9:22 AM

The Content Library is a place for materials such as readings or worksheets. Think of it as the big filing cabinet for your classroom documents. Only a teacher can put materials into the Content Library. Students can read or copy anything in the Content Library to their own notebooks, but can't modify or delete that content.

Here are some ideas for using the Content Library:

- Ask students to copy pre-made pages from sections like Handouts or Book Reports
- Post important information, such as syllabi, calendars, permission slips, or class rules
- Share course readings and texts
- Store lecture notes
- Capture class whiteboards for future reference

A teacher could organize materials into sections in the Content Library or rely on chronological ordering to keep recent materials near the top for easy discovery by students.

Project Plan and Notebook Organisation

05 October 2022

Project 38 – Quantum Monte Carlo

Toby Kidd & Thomas Watson
Supervisor: Stephen Powell

Project Plan

Aim:

In this project, we shall confirm results obtained from the path integral (PI) formalism of quantum mechanics agree with the operator method (OP), when looking at simple quantum systems such as the harmonic oscillator (HO). The Monte Carlo (MC) method will be used to model the paths needed for the PIs we would like to solve.

Background:

The MC method, named after the Monegasque casino, is a computational technique that uses a uniform distribution of random numbers to acquire numerical results – in this project we shall use it to model several paths needed to solve PIs. The PI formalism, shown by Feynman to be equivalent to the OP method for quantum mechanics, allows us to find the probability amplitude of a particle starting at point A - e.g. (x_0, t_0) - and ending at point B – e.g. (x_1, t_1). To do this, we must integrate over all possible paths from A to B. The MC method allows us to find a given number of random paths, therefore allowing us to approximate the result of the PI for an infinite number of paths.

Tasks & Timeline:

Main Task	Sub-Task(s)	Results	Rough Timeline
Project Plan			Completed by 07/10/2022
Classical Case	Metropolis Monte Carlo (MMC) Produce classical calculations for HO Comparing to classical results for HO	Plot of accepted random points – within certain parameters Ensure results correlate with classical calculations	End by Uni Week 6
Analytical Case for HO	Produce results from quantum theory to compare with our simulation		End by Week 7
Metropolis Monte Carlo for N>1		Plot of paths produced by MMC for N>1	End by Week 10
Multiple Paths	Compare results to analytic calculations – at given temperatures	Plots of paths used Plots of results for a given temperature Ensure results correlate with quantum calculations	End by Week 14
Use MC method on a more complex system (If we have time)	Adapt above method for a different quantum system		Only if we have time
Project Report			Upload by 11/01/2023

Equipment & Theoretical Techniques:

The project will be completed on standard computers – it should not be hardware intensive. We shall use the programming language of Python to apply the MC method to our simulations, producing graphical and numerical results.

Notebook Organisation:

- Each university week will have its own section.
- Each section will contain an entry of that week's work (dated).
- Each section will contain an outline of any meetings we have with our supervisor and with each other.
- A section without initials is completed by both of us. Otherwise, our initials will be contained in the title of the section if the entry was completed by just one of us. (TK - Toby Kidd; TW - Thomas Watson)

Week 2

Supervisor meeting: 28/09/2022

28 September 2022

Meeting Outline:

- Discussed the what the project would involve.
- Spoke about the background theory behind the Quantum Monte Carlo (QMC) method and the Path Integral Formalism (PI) of quantum mechanics.
- Discussed the project plan - with the aim of looking at individual tasks next week.
- Set a date for the next meeting: 05/10/2022 at 16:00

Tasks for the week:

- Read recommended textbooks to get an idea of the PI formalism and QMC method.
- Start to write the project plan.

Detail of the QMC:

We were given a couple of textbooks to familiarise ourselves with the QMC method - which are spoken about in the 'Project Plan and Background Research' section of 'Week 2'. Although we do not need to understand the PI formalism in great detail for our project, a rough idea of the theory is useful. In the meeting we discussed how the PI formalism is equivalent to the operator method we are familiar with and why it is used. For complex systems - with complex potentials - exact analytical answers are very difficult / impossible to obtain using either method - approximations need to be made. The PI formalism allows us to find solutions to these complex systems by running a Monte Carlo simulation to generate some of the paths needed for the integral (we approximate the integral as a sum - this involves the action of the system). The more paths created, the better the estimate. This can then be check against experimental data. (More detail on paths in the 'Project Plan and Background Research' section)

For our project we will mainly be discussing the Quantum Harmonic Oscillator, which has a known analytical solution from the operator method. We can thus check the results of our simulations against the known solution - if these agree, we know our simulation is correct and we can then apply this to more complex potentials (if we have time.)

Project Plan and Background Research (29-30/09/2022)

29 September 2022

Aims:

- Read the recommended textbooks about Quantum Monte Carlo (QMC) and the Path Integral Formalism (PI)
- Start to produce the project plan

Background research: (29/09/2022)

We got some background information from our supervisor in our meeting - see 'Supervisor meeting' section in 'Week 2' for more details. This, when combined with the information from the textbooks, gives a good overview of the QMC method and the PI formalism.

Textbooks read:

1. J.M.Thijssen (2007). Computational Physics - Second Edition, Cambridge University Press. Chapter 10.
2. MacKenzie, R. (2000). Path Integral Methods and Applications. U. o. Montreal. Chapters 1,2,5.

Textbook 1:
Used to get a rough outline of the MC method - mainly for what was needed later down the line. The introduction section (10.1) was the most useful, with the other areas not being that relevant at the moment - or the whole project.

Textbook 2:
Chapter 1:
Gave a good historical outline of the PI formalism - and how it was equivalent to the operator method we are familiar with from 2nd year.
Chapter 2:
This chapter went into a large amount of detail to show how both methods are equivalent. Although this was interesting, and gave good background, we do not need to know the whole detail at this moment in the project. The author uses the standard Dirac notation - which we did not cover in our 2nd year module, so we had to do further research on this (mainly by looking ahead in our MA3010 lecture notes). This will be covered in our 3rd year Quantum module, so it is not vitally important to go into much detail on the notation at the moment - especially for our project.
Chapter 2 does talk about the harmonic oscillator - the system we will be studying - so will be a useful reference later on.
Chapter 5:
This chapter spoke about how we can derive statistical mechanical results from the PI method. We shall use this later on in our project when comparing our model to the known results for the harmonic oscillator. The partition function was mainly spoken about.

A thorough understanding of the PI formalism is not needed for our project - understanding where it came from is not essential. Although chapter 2 covered this - I did not spend too much time looking into it as it wasn't relevant. The main ideas is to use the action of a path to calculate an integral - this will be approximated by a sum for our project. The formalism is similar to the classical idea of the principle of least action - the most probable path is the one that has the smallest action. The probability of a path is the negative exponential of action.

Project Plan: (30/09/2022)

The project plan was started, where we spoke about the aim of the project and the background of the QMC method and the PI formalism. Space was left to write in the timeline of tasks next week.

Conclusions:

We completed the two main tasks for the week, and started to learn the background of our project. We believe these textbooks will also be useful later on in our project; they will be good references.

Week 3

Weekly Plan and Supervisor Meeting: 05/10/2022

05 October 2022

Supervisor meeting outline:

- Discussed the details of the project and the QMC method for PIs.
- Discussed the timeline of tasks and the project plan.
- Set a date for the next meeting: 12/10/2022 at 11:00

Tasks for the week:

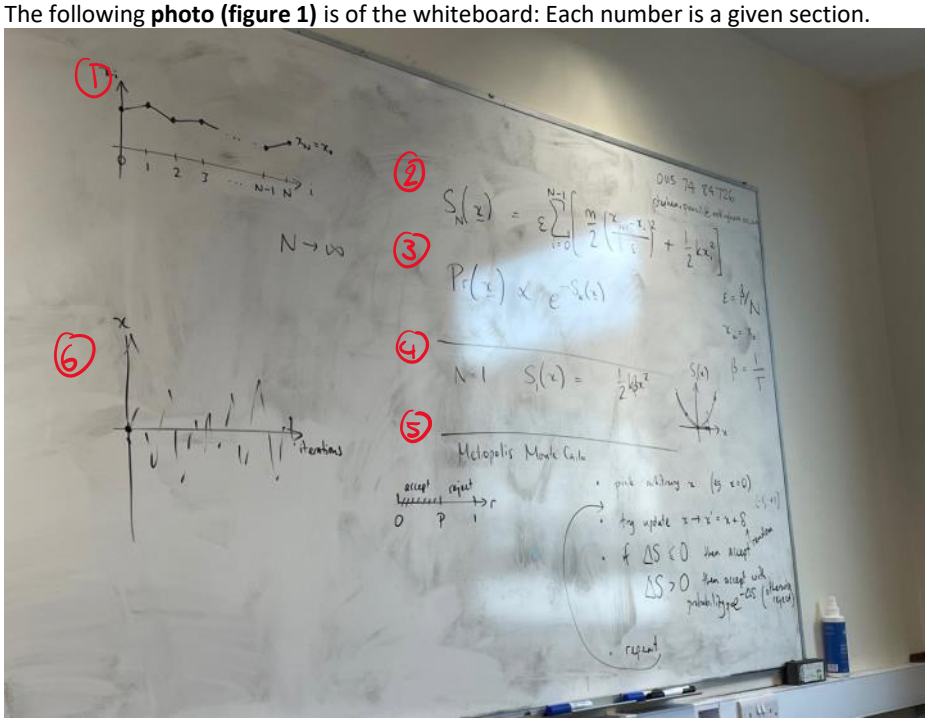
- Finish the project plan and upload by 07/10/2022
- Produce the code needed for the Metropolis Monte Carlo (MMC) method for a path consisting of one point - producing plots of the change of x over a number of iterations, with statistical analysis of the overall set of points (mean, variance, etc.).

Distribution of tasks:

- Project plan: completed together after the meeting.
- MMC code: both will attempt to create the model - the reasoning is explained below.

Project details - as discussed in the meeting:

The following **photo (figure 1)** is of the whiteboard: Each number is a given section.



1: This is an example of a random paths, which we should eventually be able to produce via our code.

2: This is the action, which is needed for the PI method as discussed in MacKenzie, R. (2000). Path Integral Methods and Appl ications. Here it has been applied to the quantum harmonic oscillator. The first term of the sum is the 'kinetic energy term', while th e second is the 'potential energy term'. ε represents the 'time-interval' between two points on the imaginary time (i) axis.

3: The probability of a given path is given by this expression - proportional to the exponential of the action for a given path.

4: Here we discuss the case for N=1, a path of one point. This will be used in the first few weeks to allow us to produce cod e for a simple path, to ensure everything works as we'd expect. This happens also to be the classical case for a harmonic oscillator.

5: The Metropolis Monte Carlo method (MMC) which we shall use to produce changes to our paths. If the code is correct, it sho uld give us an image like 6, on which we should be able to do some analysis to check the mean value of x and its variance.

A few comments:

The main goal of the project is to be able to model the quantum harmonic oscillator by producing random paths for the PI meth od and comparing results to expected ones from usual analysis of the system. To do this we need long paths, that is if the number of points N tends to infinity, our results should agree with known ones for the harmonic oscillator. For our code we shall choose a large N value.

The equations to the right of the board are:
$\epsilon = \frac{\beta}{N}$ where $\beta = \frac{1}{k_b T} = \frac{1}{T}$ if we set $k_b = 1$

These values will be used in our code (often set to 1), and will be used to allow comparison to known results.

The MMC is one that allows us to 'update' a point on our path, by adding a random distance to an x value. By doing this, if w e reduce the difference between the action of the original points and the new point ΔS then that new point is allowed. If the action increases, the new point is only allowed if $\exp(-\Delta S) < r$ where r is a random number between 0 and 1. If the new point is allowed,it is used to update the whole path.
The probability of the path given in 3 is proportional to the new action calculated for the new path. We want a small S to be the most likely - as shown by the probability of a path being a negative exponential.

Useful textbooks:

1. Yeomans, J 1992, Statistical Mechanics of Phase Transitions, Oxford University Press, Oxford. Chapter 7 Sections 1,2.

Will be a useful reference to look into the MMC and helpful when we apply it to the harmonic oscillator.

2. Feynman, R.P., *Space-Time Approach to Non-Relativistic Quantum Mechanics.* Reviews of Modern Physics, 1948. **20**(2): p. 367-387.
3. S. C. Bloch, *Introduction to Classical and Quantum Harmonic Oscillators*. (John Wiley & Sons, Incorporated, Hoboken, UNITED STATES, 1997). Chapters 11,12.

Post Supervisor meeting:

Outline: 1. Project Plan 2. Tasks for the week

After our supervisor meeting, **we finalised our project plan** - which is at the front of this notebook. This was then uploaded to Moodle.

We also spoke about our work for the coming week and decided we should **both** produce code for the MMC, as this would produce greater understanding of the method and allow for checking if one of us were to go wrong. We hope this code should be finishe d by our next supervisor meeting.

(TW) Metropolis Monte Carlo Method and Code (05-08/10/2022)

05 October 2022

Metropolis Monte Carlo Method (MMC) :

NB: As mentioned in the 'Weekly Plan and...' section of 'Week 3', both of us will produce a MMC method.

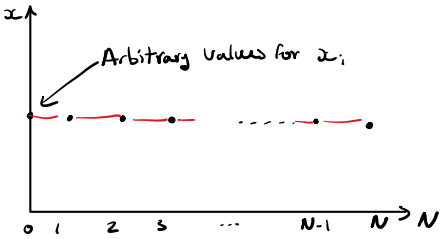
Research:
(05/10/2022): Some background information was covered in our Supervisor meeting - where we spoke about how we would apply the MMC to our project. The details of this are in the 'Weekly Plan and Supervisor Meeting' section of 'Week 3'. Our Supervisor mentioned a textbook to look into the method, where it was applied to another model. I plan to read this tomorrow before starting my code to ensure I understand the method fully.

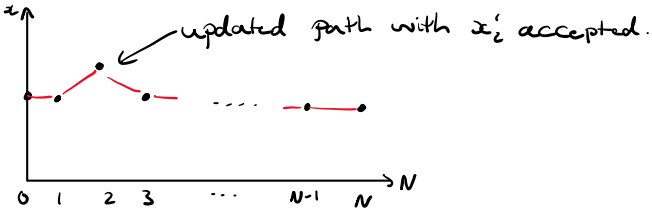
The recommended textbook:
Yeomans J, 1992, Statistical Mechanics of Phase Transitions, Oxford University Press, Oxford. Chapter 7 Sections 1,2.

(06/10/2022): Section 7.1 and 7.2 of the above textbook provided an example of the MMC method when applied to the Ising model. As this model isn't related to our project, the details of the Ising model need not be known. However, it was important that we could recognise where the MMC was used in this model. We could also relate this to our model for QMC. The MMC method was very similarly used here, expect instead of changing a position of a point by a random value, the spin value of a lattice site was changed. This was a useful guide to help formulate our own MMC method for our problem.

Algorithm plan: (06/10/2022)
Before starting the write the MMC code, I wanted to sketch out a rough plan to follow. This plan is for a path with N points, using the boundary conditions that $x_0 = x_N$. This will be restricted to a path of one point for this week to ensure that the algorithm is working correctly.

1. We first choose arbitrary values for our points x_i
2. Randomly choose an x_i to update.
3. We update x_i as follows: $x_i' = x_i + \delta$ where δ is a random number (uniformly generated) between two values, most likely to be [-1,1]
4. We calculate the action of both the original x_i and the updated value. This will allow us to find the difference in action.
   The action is:
   $$S = \epsilon \sum_{i=0}^{N-1} \left(\frac{m}{2} \left(\frac{(x_{i+1} - x_i)}{\epsilon} \right)^2 + \frac{1}{2} k x_i^2 \right)$$
   While the difference in action is: $\Delta S = S_{updated} - S_{original}$
5. If $\Delta S \leq 0$ then we accepted the updated path, storing both the new action and the path array.
   If $\Delta S > 0$ then we only accept the new path if $r = \exp(-\Delta S) < p$ where p is a random number (uniformly generated) between (0,1). Otherwise the new path is rejected.
6. The process repeats from step 2.

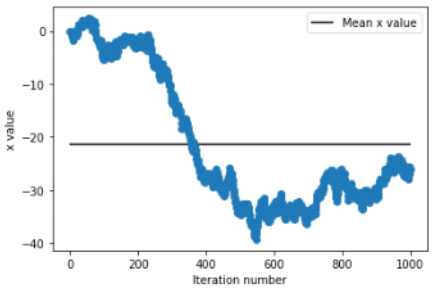


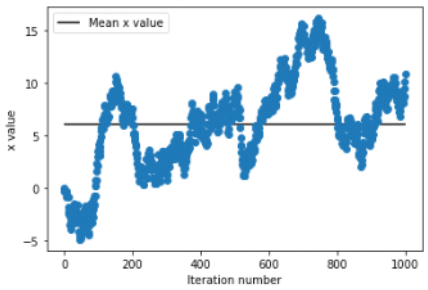


The **plan for this week** is to create the general algorithm for only one point. Then to produce a plot of our x value v. iteration number. The plot should be symmetrical about the x-axis. The mean value for x and its variance should also be found. (Should look like figure 1 area 6 in the supervisor meeting notes.)

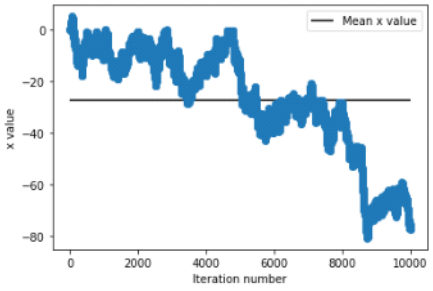
Code: (06/10/2022)
Using the algorithm plan, I created a function in python to attempt to follow the procedure. Apart from a few issues involving numpy arrays - noting that they were not copying correctly until np.copy() was used - and some issues regarding index numbers, the code followed the algorithm for the case N = 1. Most of the variables needed were set to 1 also.
Once the small number of bugs were fixed, it created a few plots noting that the figures created did not follow the symmetrical nature that we expect. I played around with some of the variables - setting them to roughly the correct order of magnitude - but this did not make too much of a difference, this was even the case for a large number of iterations. However, upon changing the range of values of delta, when updating a point, the graphs started to look roughly correct by eye, though still not as I would have expected - the average value for x was not near to its initial value as we would expect. I decided to come back to the code another day, and see if I could find any errors in the code - or if the values needed to be tweaked more.
Below are some screen shots of the graphs I produced varying the number of iterations, keeping variables to 1 and with delta between [-1,1].

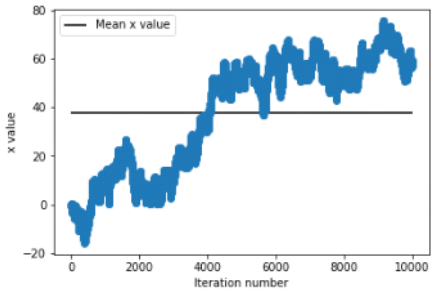
The plan for the rest of the week is to find out why I'm not getting the expected plot - and once resolved produce plots with data for the mean and variance of x. This will then be added to the notebook, with a conclusion.





All these plots were produced with N=1, and variables also equal to 1, with delta between [-1,1]





The following is the code at this stage, with only the number of iterations being changed:

In [5]:
```
#Created on Thu Oct  6 10:18:44 2022
#@author: WatsonTJ
```

In [5]:

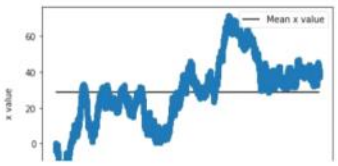
```
#Created on Thu Oct  6 10:18:44 2022
#@author: WatsonTJ

import random
import numpy as np
import matplotlib.pyplot as plt

def MMC(N, beta, m, k, iterations):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    eps = beta/N #created epslion from beta and N provided
    su1 = 0 #needed for the sum for the action
    paths = [x] #a list to store all paths
    for j in range(N):
        su1 =+ (m/2)*((x[j+1]-x[j])/eps)**2 + 1/2*k*(x[j])**2
        #the sum in the action calculation
    S_1 = eps*su1 #action of the original path
    for i in range(iterations):
        #This is the MMC method
        #We first choose a random position to alter from x_0 to x_n-1
        pos = random.randint(0,N-1)
        #the delta is a random number between two end points to alter the chosen x by
        delta = random.uniform(-1, 1) #I believe this is the issue with the plots
        #creates a copy of the current path to alter
        xtemp = np.copy(x)
        #we set a the chosen point in the copied array to its new point y_2
        y_1 = xtemp[pos]
        y_2 = y_1 + delta
        xtemp[pos] = y_2
        su2 = 0

        for j in range(N):
            su2 =+ (m/2)*((xtemp[j+1]-xtemp[j])/eps)**2 + 1/2*k*(xtemp[j])**2
        S_2 = eps*su2 #we calculate the action of this new path with y_2
        DS = S_2-S_1 #we calculate the change in action between the old and new paths
        #this ensures that the boundary condition holds,
        #that is that the start and end of the path must be the same value
        xtemp[N] = xtemp[0]
        #here we apply step 5 of the algorithm plan to see if we accept or reject the update
        #if accepted, our path x becomes new path xtemp
        #else, our path remains as it was before the attempted update
        if DS <= 0:
            x = np.copy(xtemp)
        else:
            r = np.exp(-DS)
            p = random.uniform(0,1)
            if r < p:
                x = np.copy(xtemp)
        #the path x, update or the original depending on the above calculation, is added to the list of paths
        paths.append(x)
    #once the number of iterations is reached, the function produces an array of paths
    return np.asarray(paths)


#This is used to test the function by plotting the graph with most variables set to 1
#When the function is working fully, it can be imported into other files
N = 1
beta = 1
m = 1
k = 1
it = 10000
paths1 = MMC(N, beta, m, k, it)

x_bar = np.mean(paths1[:,0])
xis = np.arange(it+1)
yis = paths1[:,0]
plt.plot(xis,yis,"o")
plt.hlines(x_bar, 0, it, "black", label="Mean x value")
plt.xlabel("Iteration number")
plt.ylabel("x value")
plt.legend()
```

Note(1): I believe the condition that the start and end points always need to be the same value, and this was included. Want to double check this is correct at our next meeting, as could cause problems later down the line.

Note(2): It could also be using 1 as the value for the physical quantities produces errors. Try setting them to the correct order of magnitudes.

Out[5]:

```
<matplotlib.legend.Legend at 0x28b50bb79a0>
```



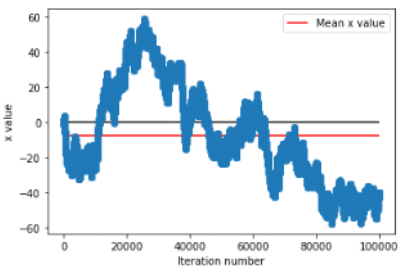
(08/10/2022): I set all variables to 1, and the number of iterations also to 1, and checked that my formulae for calculating the action were correct - and checked to see if the correct updates were allowed (and vice versa). This seemed to be correct.

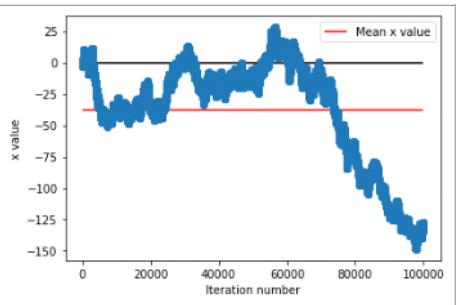
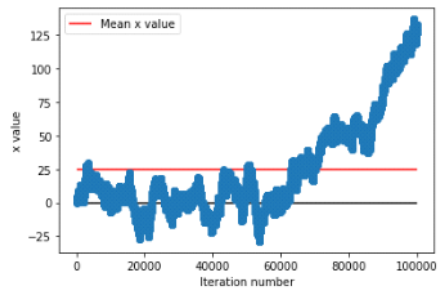
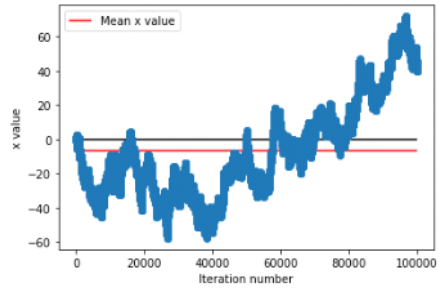
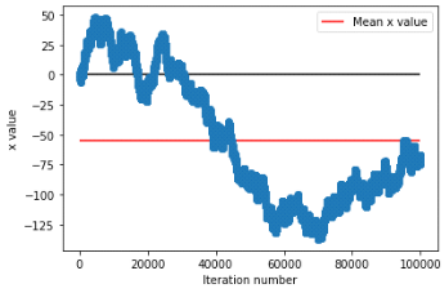
I changed the delta value to a few different options: (-0.5,0.5), (-0.1,0.1), (-10,10). This had an effect on how much the graph 'ran away' - that being the distance from 0. As expected the higher the range, the more likely the graph would move farther from zero. I settled on (-0.5,0.5) range as this seemed sensible and gave plots that followed closer to what I had expected.

Combining this with using the correct orders of magnitude for the physical variables gave more plots that looked as how I wanted them to - additionally iterating more times give better figures.

The final variable values were in the correct orders of magnitude:
beta = 1e20
m = 1e-31
k = 100

Below are some plots generated for 100000 iterations, with the mean value of x marked - the x=0 axis also explicitly drawn in to show where the axis of symmetry was expected to be. A table was also produced.
Results:

Run Number	Mean value of x (3sf)	Variance of x (3sf)	Standard deviation of x (3sf)	Plot
1	-7.67	794	28.2	

2	-37.9	1770	42.0	
3	24.8	1290	35.9	
4	-6.92	798	28.3	
5	-54.6	2980	54.6	

Remarks:
Runs 1 and 4 give a plot and data that is expected, while the others are not ideal. I produced many more plots, given that this is a statistical model each run will be different, and followed a similar pattern of every 3 or 4 being close to what I expected. I have decided to leave the code as it is; I will discuss with Toby when he has completed his code to see if he has similar problems - and I'll also discuss this with my supervisor to see if there are any obvious places where I have gone wrong - or if these plots are typical.

I noticed that the majority of the plots gave oscillatory plots - even if the mean value of x was higher than expected. A few did give plots that 'ran-away' either to x values with a large modulus. This happened at both small and high iteration numbers - is this due to the model being random or was there an error somewhere?

Conclusion:(08/10/2022)

I am mostly happy with my code and believe there are not any obvious errors. I am unsure whether my plots are actually correct, or there is an issue somewhere - possibly still due to the choice of delta. However, I produced the code that I wanted to for this week - and this can lead nicely onto the next step on the plan. If there are any issues, I do not believe they will take too long to fix. The next step will be to use this function to produce approximations of the behaviour of a classical harmonic oscillator and compare with known values. The case of N=1 is equivalent to the classical harmonic oscillator.

Questions for follow up:

- Are there any issues with my code?
- Do any values need to be changed that may solve this problem – if there is a problem.
- Should the start and end points of the path always be the same – even if the start point is updated? If this is the case when do we set them equal - before or after the ΔS calculation.
- Why do some of my graphs not give the expected oscillatory behaviour – no matter the mean value?
- Is the number of iterations a factor?
- Are the S and ΔS calculations correct?

(TK) Metropolis Monte Carlo

Sunday, October 9, 2022 11:11 AM

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct  9 17:01:25 2022

@author: tobyk
"""

"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot

"""values"""
k = 100
beta = 1e-2

def S_1(x):
    return 0.5*k*beta*x**2

"""Pick arbitrary value for x"""
x = np.array([0])

for i in range(1000):
    """Update x and append"""
    delta = random.uniform(-1,1)
    x = np.append(x, x[-1]+delta)

    """Test update"""
    if S_1(x[-1]) <= S_1(x[-2]):
        pass
    else:
        prop = np.exp(S_1(x[-2])-S_1(x[-1]))
        if random.random() < prop:
            pass
        else:
            x = np.delete(x, -1)
            x = np.append(x, x[-1])


"""axis"""
imag_time = np.array(range(len(x)))

"""plot"""
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(imag_time, x)
ax.set_xlabel('Imaginary Time')
ax.set_ylabel('x')
plt.title('Metropolis Monte Carlo')

print("Average: "+str(np.mean(x)))
```

After meeting 12/10/2022, changed these values both to 1, to make the data points fit better

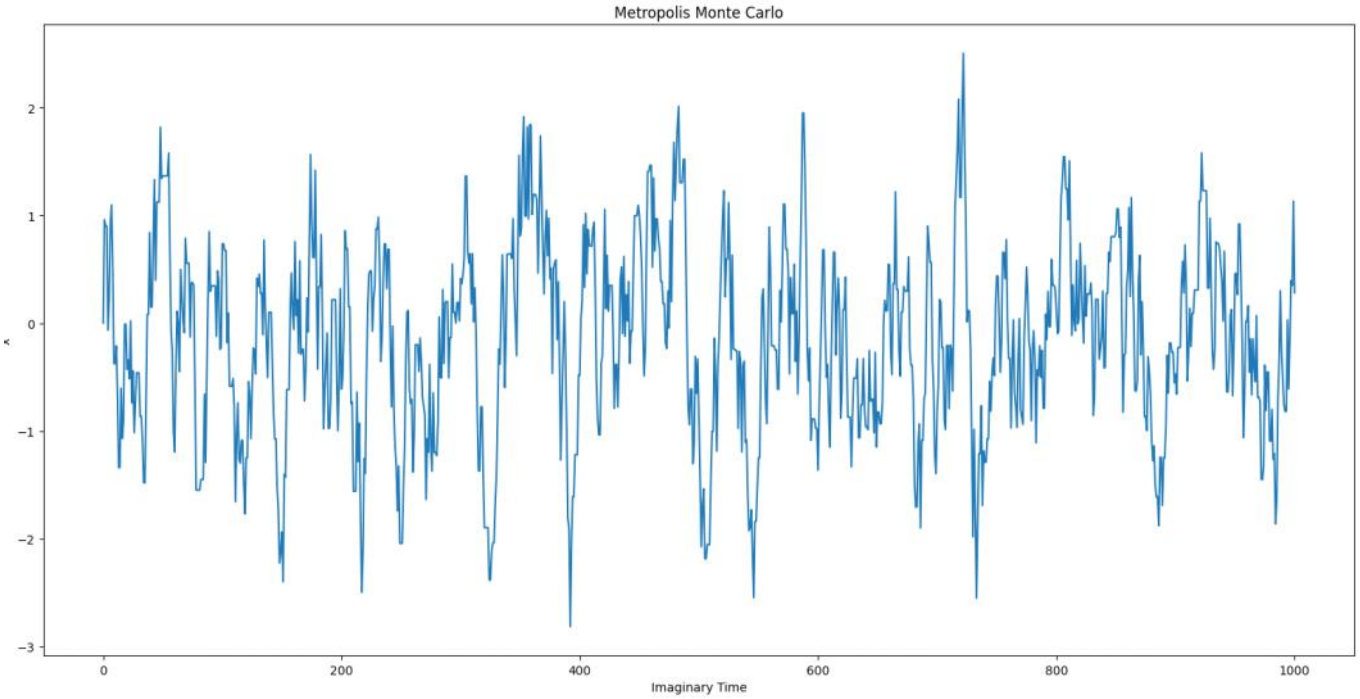
Need to find delta which produces the most reliable data

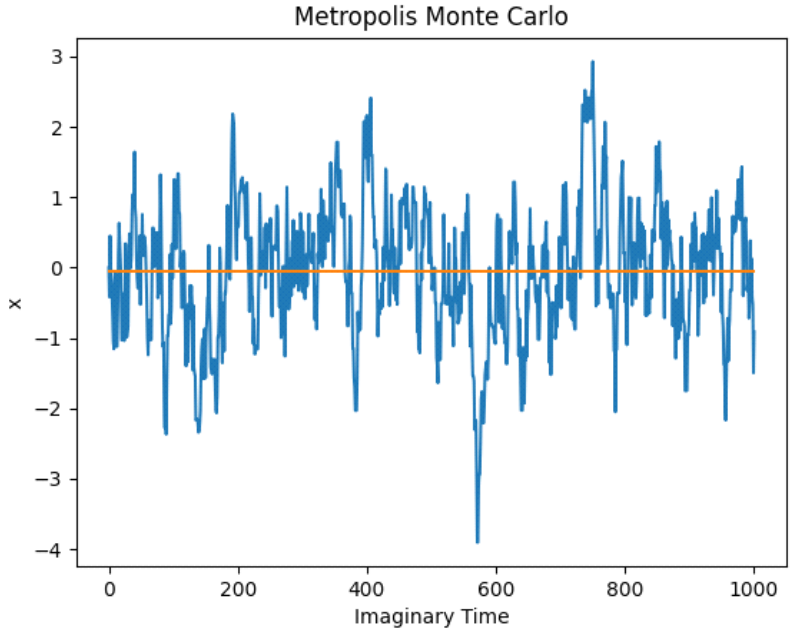
Initially was using x[-2]-x[-1] , which skewed the data greatly

Should change to prob , as confusing

x - axis

Results





Average is roughly zero, which agrees with analytical results, will check in more detail after supervisor meeting next week.

Week 4

Meeting: 10/10/2022

10 October 2022

Meeting Notes:

We met to discuss our progress on our individual MMC methods. Points in bold are action items:

- Agreed on the organisation of the notebook.
- Both of us were having slight issues.
- We had gone down slightly different routes to produce our algorithms, but the overall effect was the same.
- After playing around with both of our code, we noticed two main issues:
    a. That the value of the physical variables made a significant difference to our plots: we are both unsure about the 'size' of x and its physical interpretation. **To be discussed at our next supervisor meeting.**
    b. (TW) worked out that the issue with his plots was that every point was being accepted - this was probably due to the size of physical variables making certain values 0. Related to a. **Will look at code again to see if any issues.**
- Will meet again on 12/10/2022 with supervisor to go over issues and continue with the plan - that is comparing our results to the classical harmonic oscillator - **need to discuss how to go about this with supervisor.**

(TW) Metropolis Monte Carlo Method Code (10-12/10/2022)

10 October 2022

Metropolis Monte Carlo Method (MMC) :

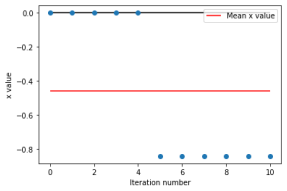
NB: Continued from week 3 after meeting with TK.

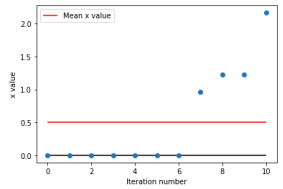
Post meeting plan: (10/10/2022)
Having compared my code to Toby's, we both knew there were issues with both our work. In doing some debugging, and comparing mine with his, I found that every updated point was being accepted - even for iterations of over 1,000. There was clearly an issue here.

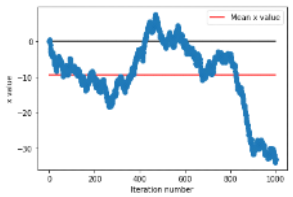
The plan was to look over my code and see why every update was being allowed, and fix this issue, to try to get plots closer to what was expected. We noted that the size of physical variables may be playing apart, so needed to clarify with our supervisor the physical meaning of x, its size, and the value of some of the other variables - including δ.

Debugging: (10/10/2022)

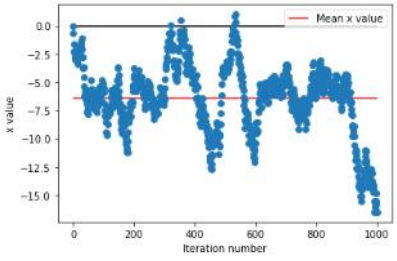
Change	Result	Plot

Set all variables to 1, with delta between [-1,1]:
`N = 1`
`beta =1#e20`
`m = 1#e-31`
`k = 1#00`
`it = 10`
`paths1 = MMC(N, beta, m, k, it)`
	Updates were rejected - meaning the code to reject them is working. Seems to confirm this is an issue of 'size' of physical variables.	

Changed the mass back to the correct order of magnitude, delta as above:
`N = 1`
`beta =1#e20`
`m = 1#e-31`
`k = 1#00`
`it = 10`
`paths1 = MMC(N, beta, m, k, it)`
	Updates still rejected	

Changed the k value and the β value. First just k, second just β, thirdly both. Increased the number of iterations to 1000:
`N = 1`
`beta =1e20`
`m = 1e-31`
`k = 100`
`it = 1000`
`paths1 = MMC(N, beta, m, k, it)`
`data = paths1[:,0]`
	Update no longer rejected. I also printed the value of $r = \exp(-\Delta S)$: which was always equal to zero (ie ΔS very large). This means that there is: **an issue with the delta S calculation. Or an issue due to relative sizes.**	

Set the number of iterations to 1, and printed the values for the updated x, ΔS, r, and the random probabilty p - to check if all was correct.	Code produced: x = -0.9456, and a $\Delta S = 4.47 * 10^{21}$, which I agreed with. This gives an r roughly equal to zero, which will always be accepted. **This leads me to think it is an issue of the size of variables.**	
I changed when the last and first point were set to be the same - before the calculation of ΔS.	This didn't have any effect - I think due to the size of beta. **But I do believe they should be set to be the same before calculating ΔS so I have left it like this**	

I changed the value of β. Setting $K_b = 1$ to try $\beta = 0.1$ The number of iterations was 1000. Delta was [-1,1]:
`N = 1`
`beta =0.1`
`m = 1e-31`
`k = 100`
`it = 1000`
`paths1 = MMC(N, beta, m, k, it)`
	Updates were not being rejected - but the overall pattern still wasn't as I expected. Not many were being rejected - 10 of the 1000.	

Changelog: (10/10/2022)
Changed where the first and last point were set to be equal - before the action calculation. **Double check this was the correct thing to do.**

```
33          #we set a the chosen point in the copied array to its new point y_2
34          y_1 = xtemp[pos]
35          y_2 = y_1 + delta
36          xtemp[pos] = y_2
37          su2 = 0
38          #this ensures that the boundary condition holds,
39          #that is that the start and end of the path must be the same value
40          xtemp[N] = xtemp[0]
41
42          for j in range(N):
43              su2 =+ (m/2)*((xtemp[j+1]-xtemp[j])/eps)**2 + 1/2*k*(xtemp[j])**2
44          S_2 = eps*su2 #we calculate the action of this new path with y_2
```

Note: This line was moved up a few places, to before the calculation of the new action

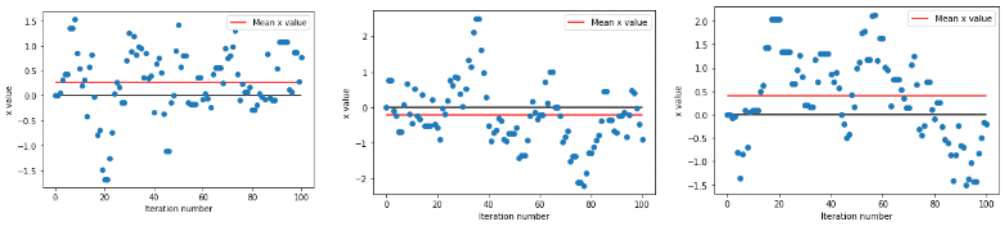
Conclusion: (10/10/2022)
I believe this is due to a value problem and getting the relative sizes correct. I shall raise this at our next supervisor meeting to see how best to solve this issue.

Supervisor meeting comments and corrections: (12/10/2022)

In our meeting we discussed each other's codes and tried to resolve the issues in mine. The issues were not due to the 'relative sizes' of the physical variables. In fact, setting most to be **equal to 1** will produce the necessary analysis we need.
There were three main issues that we very straight forward to fix:

1. The sum for calculating the action had '=+' instead of '+=' - this wouldn't cause an issue until N>1
2. The initial action for the ΔS calculation was never updated - this is what lead to very large ΔS values. This was fixed by setting S_1 to the previous S_2 if a point was accepted.
3. The variables r and p were the wrong way around. Now we have:
   $p = \exp(-\Delta S)$
   r = random number between (0,1)

These three changes together, with the physical variables and δ equal to 1 produced the plots we would expect. Below are a few examples.



Conclusions:
After these fixes that code as producing the correct plots. The next step was to calculate the statistical values for the plots - the mean value and error bars - to see if the plot were giving the values expected. The process of this, and the explanation, is covered in other sections.
I was correct to change where setting the first and last point to be equal to each other was.
I also simplified calculating the original action S_1 at the start - this will always be zero as every point begins as zero. So it was inefficient to calculate the sum.

New Code:

In [1]:

```
"""
Created on Thu Oct  6 10:18:44 2022

@author: WatsonTJ

"""
import random
import numpy as np
import matplotlib.pyplot as plt


def MMC(N, beta, m, k, iterations):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    eps = beta/N #created epslion from beta and N provided
    S_1 = 0 #action will always start as 0 for x=0
    paths = [x] #a list to store all paths

    for i in range(iterations):
        #This is the MMC method
        #We first choose a random position to alter from x_0 to x_n-1
        pos = random.randint(0,N-1)
        #the delta is a random number between two end points to alter the chosen x by
        delta = random.uniform(-1, 1)
        #creates a copy of the current path to alter
        xtemp = np.copy(x)
        #we set a the chosen point in the copied array to its new point y_2
        y_1 = xtemp[pos]
        y_2 = y_1 + delta
        xtemp[pos] = y_2
        su2 = 0
        #this ensures that the boundary condition holds,
        #that is that the start and end of the path must be the same value
        xtemp[N] = xtemp[0]

        for j in range(N):
            su2 += (m/2)*((xtemp[j+1]-xtemp[j])/eps)**2 + 1/2*k*(xtemp[j])**2
        S_2 = eps*su2 #we calculate the action of this new path with y_2
        DS = S_2-S_1 #we calculate the change in action between the old and new paths
        #here we apply step 5 of the algorithm plan to see if we accept or reject the update
        #if accepted, our path x becomes new path xtemp
        #else, our path remains as it was before the attempted update
        #The new action is set to be S_1 if the update is accepted

        if DS <= 0:
            x = np.copy(xtemp)
            S_1 = S_2
        else:
            p = np.exp(-DS)
            r = random.uniform(0,1)
            if r < p:
                x = np.copy(xtemp)
                S_1 = S_2
        #the path x, update or the original depending on the above calculation, is added to the list of paths
        paths.append(x)
    #once the number of iterations is reached, the function produces an array of paths
    return np.asarray(paths)


N = 1
beta =1
m = 1
k = 1
it = 1000
paths1 = MMC(N, beta, m, k, it)
data = paths1[:,0]

x_bar = np.mean(data)
x_std = np.std(data)
plt.plot(np.arange(it+1),data,"o")
plt.hlines(x_bar, 0, it, "red", label="Mean x value")
plt.hlines(0, 0, it, "black")
plt.xlabel("Iteration number")
plt.ylabel("x value")
plt.legend()

print(x_bar)
```

Note: This will be changed to iterations-1 - as is explained in the 'error data code' section of 'week 4'

```
0.07958159497977577
```

Meeting: 12/10/2022

Wednesday, October 12, 2022 9:44 PM

Supervisor meeting outline:

- Discussed the details of finding the mean and standard errors of data
- Set an updated time on Wednesday meeting times to 1100

Tasks for the week:

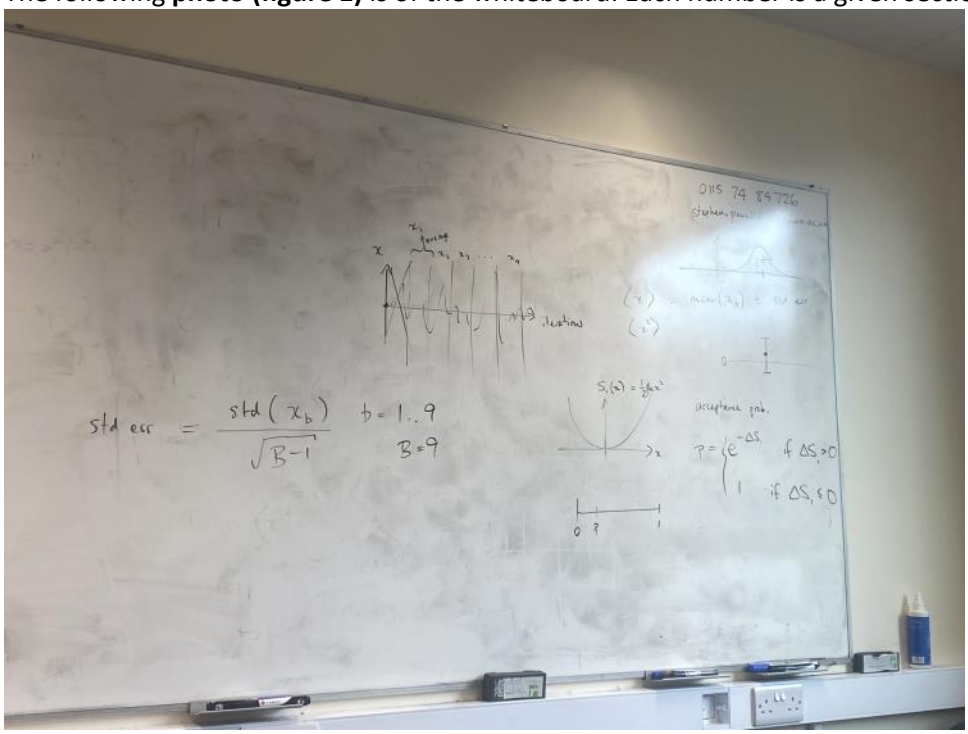
- Create a function for displaying the mean and standard error for data
- Call this function and pass the variable for the values of x values and x^2 values
- Check if values make sense (e.g. mean of x^2 values should be roughly 1)

Distribution of tasks:

- Function: both will create a function for the mean and standard errors
- We will then compare our values for our mean and standard errors to make sure both codes are producing
  the correct code and adjust otherwise (incorporating pieces of each other's code)

Project details - as discussed in the meeting:

The following **photo (figure 1)** is of the whiteboard: Each number is a given section.



1: Here we looked at correcting the acceptance probability in our code, and making sure that it was accepting and
rejecting in the correct proportions

2: Formula for standard error

3: Successive measurements, which are the same are not statistically independent, and the standard error formula

only works with statistically independent data points. We avoid this by splitting the data into tenths from the
number of iterations. The first tenth iterations are immediately dismissed, due to the time to get away from zero (equilibration time). The other 9 blocks remaining are taken to find the means are used to calculate the standard error.

4: This shows the qualitative explanation of the mean and the standard error in terms of a Gaussian.

A few comments:

Statistically independent data is achieved through splitting the data into tenths, as the tenths are statistically independent from one another.

(TW) Error Data code (12-13/10/2022)

12 October 2022

Error Data Code:

After fixing the issues with the MMC method, I was creating plots that looked correct. We now needed to find the average value of x over all the iterations to see, within the an error, if this average was zero - as we would expect. This will not happen in all cases - due to the gaussian nature of the expectation of x, $\langle x \rangle$. We would expect around 70% of all $\langle x \rangle$ to be within an error of zero.

Background:(12/10/2022)
As mentioned in the 'Meeting (12/10/2022)' page, we needed to create a function that would return the mean value over all iterations and the standard error. x_b is the set of all x points.The usual formula for the standard error, using numpy's built in standard deviation function, is:

$$\text{error} = \frac{np.std(x_b)}{\sqrt{B-1}}, b = 1, \dots, B$$

This assumes that our data is independent - this is not the case as sometimes an update will be rejected and so the next point is dependent on that un-updated point. If we didn't account for this, we would get a standard error that was too small.
To account for this, we would first section our data into 10 chunks: x_b. By taking the standard deviation between chunks, we ensure, if the number of points between the chunks is large enough, that the data is independent.
We set B=9.
Also we remove the 1st chunk: this allows the MMC method to produce enough points such that 'it has forgotten about its starting point x=0' - this is known as the equilibrium time, and also helps to reduce the number of dependent points.
From this we can find the error and mean value for a given path:
$\langle x \rangle = mean(x_b) \pm error$. Within this error bar, we'd expect $\langle x \rangle = 0$.

Plan:(12/10/2022)

- Produce the error data code to return the mean and error for a given array of x values - this will be, for example, all the x_0 values in all the paths produced by the MMC function.
- Run this function over several iterations of the MMC function (the MMC function itself iterating enough to ensure independent sections x_b)
- Run the error data function on each set of paths for each iteration of the MMC function to find the mean of x_0 and its error for a given run of the MMC function (a set of paths)
- Once this has been done a number of times, produce a plot of the mean value of a set of paths against the index of said set of paths, with error bars included. The error bars should include x=0 in their length - roughly 70%.
- If this works, do the same but for $\langle x^2 \rangle$, which should be within an error of 1.

Algorithm: (12/10/2022)

1. Separate the array of points into 10 chunks.
2. Discard the first chunk.
3. Find the standard deviation of each of the remaining chunks.
4. Find the standard error of the data set, by using the standard deviation of the 9 chunks.
5. Find the mean of the whole data set - including the 1st chunk.
6. Return both the mean and error as a list.

Code: (13/10/2022)
The code was relatively simple to implement - a function (errordata) was produced as outlined above. Once done I produced plots for $\langle x \rangle$ and $\langle x^2 \rangle$. Also I **changed** the MMC function slightly to iterate: 'iterations'-1 times - to ensure we had a number of points = iterations. (as the first path of all zeros counted).

MMC change:

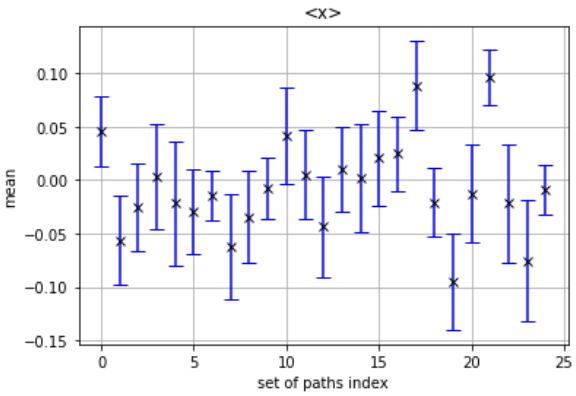
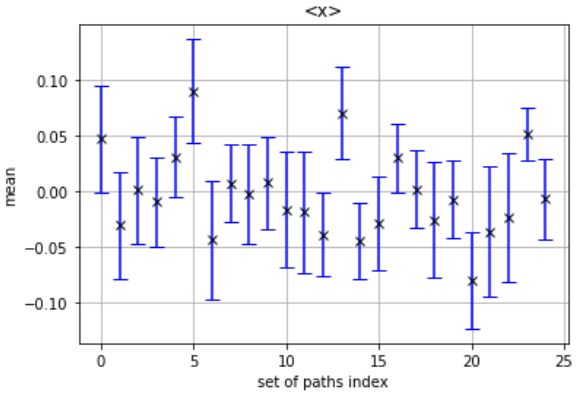
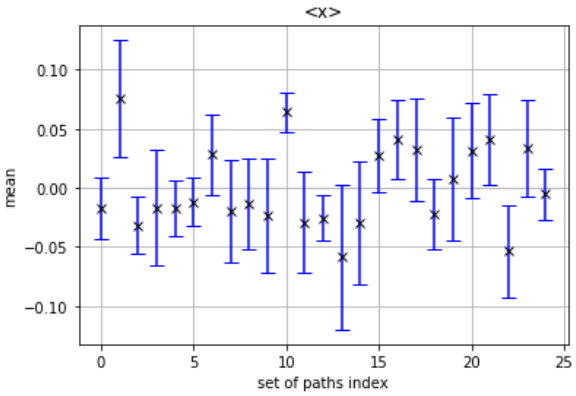
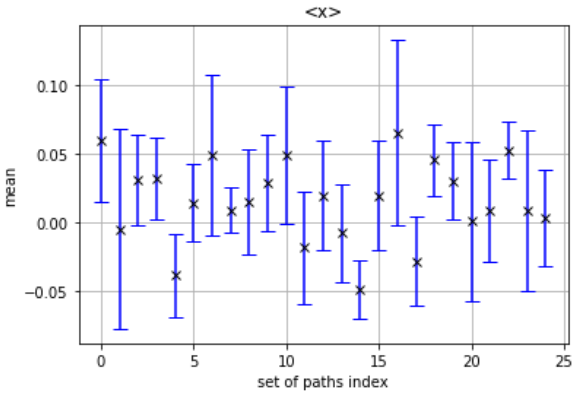
```
        #iterations-1 to ensure we have the number of paths = 'iterations' (otherwise would be iterations+1 due to starting path)
        for i in range(iterations-1):
```

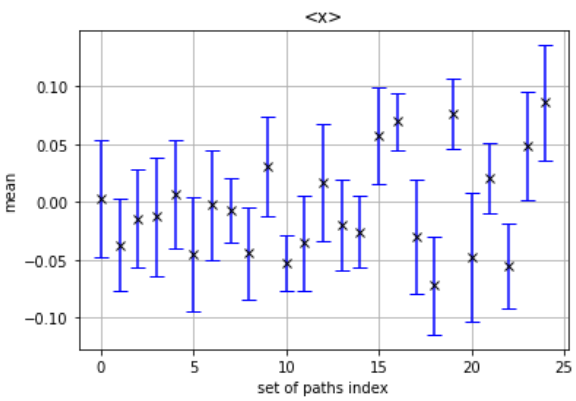
Errordata function:

```
    def errordata(xarr):
        #creates a function that find the mean and standard error for a given array of points
        #usually for an array of all values x_i takes for the number iterations in the MMC method
        sects = np.split(xarr, 10) #Splits array into 10 sections, on which we will calculate means and std deviations
        means = []
        for i in range(8):
            #for each section - except the first - we find the mean of each section
            section = sects[i+1]
            means.append(np.mean(section))
        #calculates the std error for the 9 sections we are interested in
        error = np.std(means)/np.sqrt(8)
        #calculates the total mean for all points - including the first section
        totalmean = np.mean(xarr)
        #returns a tuple of the mean and std error for that set of x values
        return [totalmean, error]

```

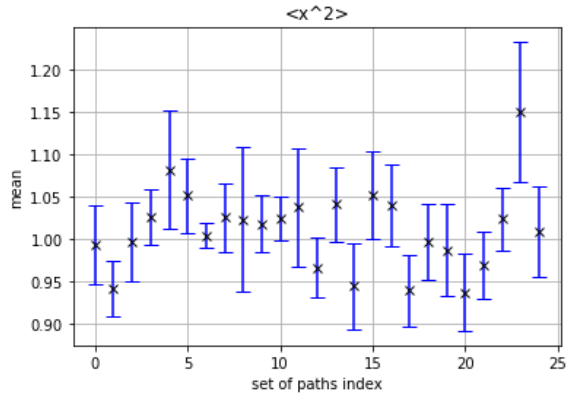
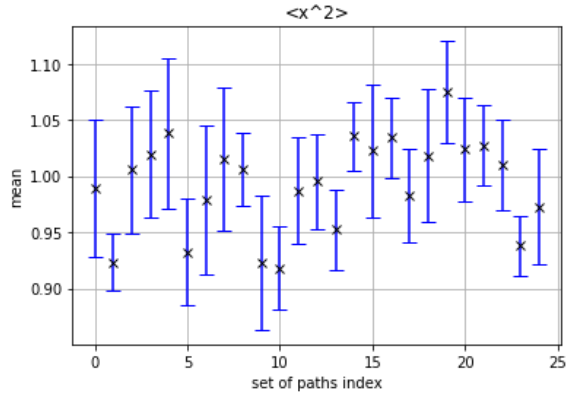
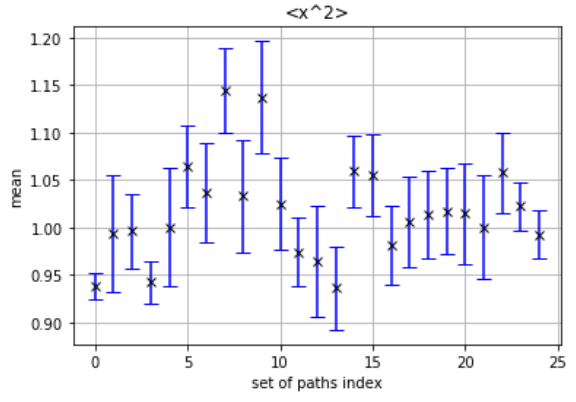
Results:(13/10/2022)

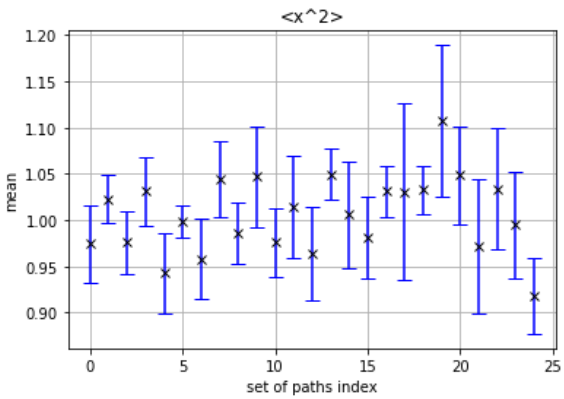
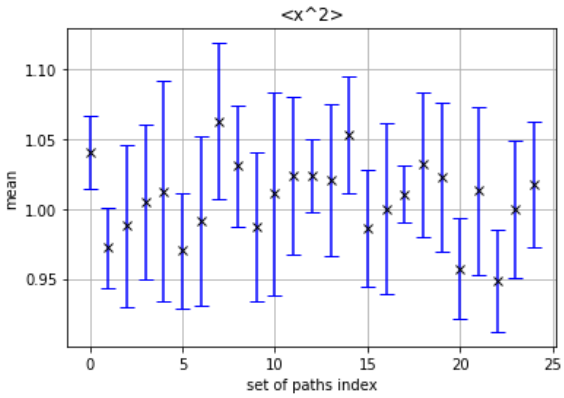
The following graphs are for $\langle x \rangle$. This should be within an error of 0 for roughly 70% of the points. I produced 25 different sets of paths, with the number of paths in a single set being 10000.
The percentages were correct, so I was happy with my code.

Run Number	Plot of $\langle x \rangle$	Percentage within an error of 0
1		72%
2		80%
3		72%
4		72%

5		68%

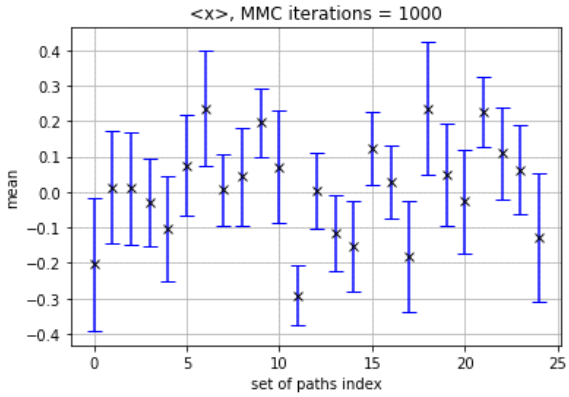
The following graphs are for $\langle x^2 \rangle$. The values used above were kept. We we expect $\langle x^2 \rangle$ to be within an error of 1.

Run Number	Plot of $\langle x^2 \rangle$	Percentage within an error of 1
1		72%
2		68%
3		64%

4		72%
5		80%

Again I was happy with the percentages - meaning my MMC method is working correctly.

Remarks: (13/10/2022)
Changing the number of iterations for the MMC method did effect the result of the graph. The mean values of x had a greater range, and the percentages were less. This is due to the independence problem mentioned above - the higher the number of iterations the closer to the expected result - as you would expect.
Below is a graph at 1000 iterations on the MMC, for $\langle x \rangle$:



This has a percentage of points within zero of 64%, and a noticeable increase in range (-0.4 to 0.4) compared to (-0.1 to 0.1) for an iteration number of 10000.

Code to produce the plots (for x^2, all the path array values were squared):

```
#Here we call the MMC function n times,each time taking the list of all the x_0 coordinates
#the mean and std error are calculated using the errordata function, with the respective results stored in two lists
#we then plot the mean value of x for a particular run against the run number, with error bars present
N = 1
beta =1
m = 1
k = 1
it = 10000
n = 25

means = []
errors = []
for i in range(n):
    paths = ((MMC(N, beta, m, k, it)[:,0]))
    data = errordata(paths)
    means.append(data[0])
    errors.append(data[1])

x = np.arange(n)
y = np.asarray(means)

plt.errorbar(x, y, xerr = None, yerr = np.asarray(errors), marker='x', color='black', ecolor='blue', linestyle = 'none', capsize = 5 )
plt.ylabel('mean')
plt.xlabel('set of paths index')
plt.grid()
plt.title('<x>')
```

Conclusion:(13/10/2022)
With the MMC issues fixed, the error data plots show the function creates correct paths that line up with the statistics we would expect from a path of length N=1 - that is analogous to the classical harmonic oscillator. The next steps will be to compare my data with Toby's before moving onto the more complex case for longer paths and comparing them to the quantum harmonic oscillator.

(TK) Errors

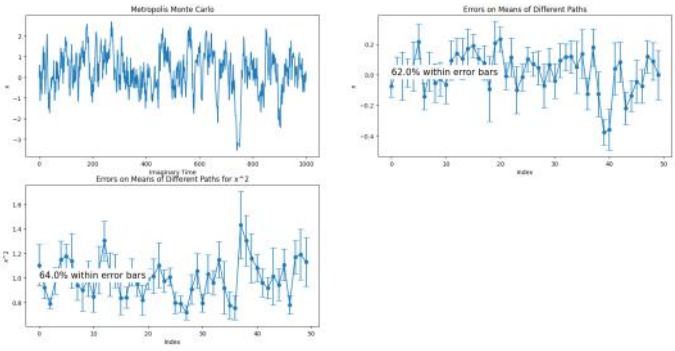
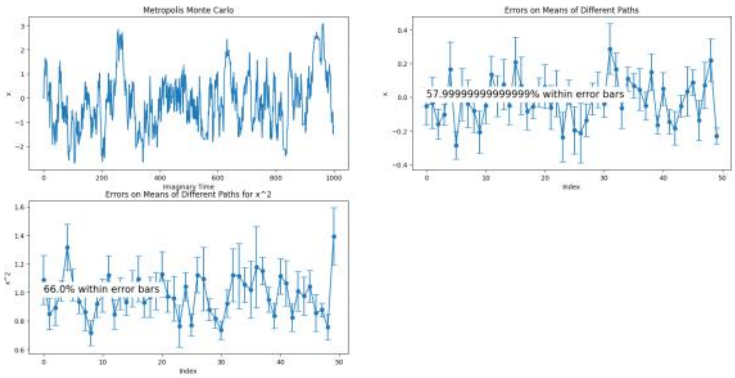
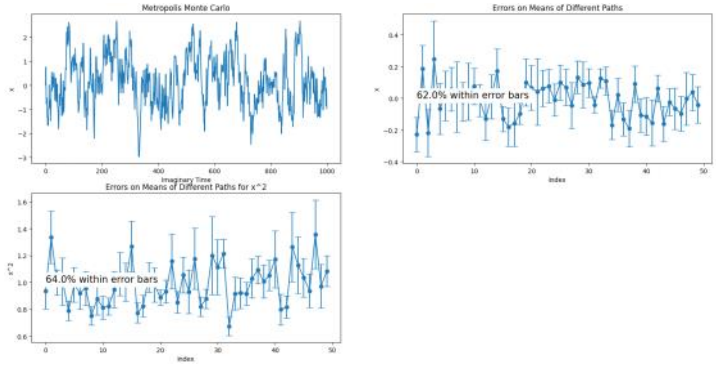
Wednesday, October 26, 2022 1:15 AM

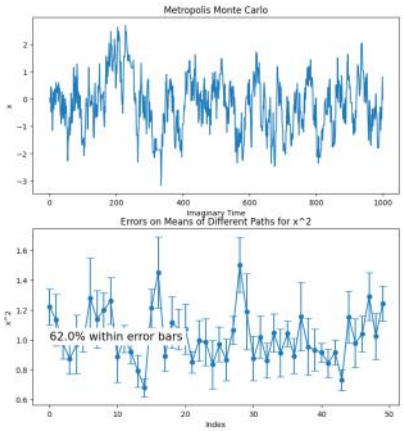
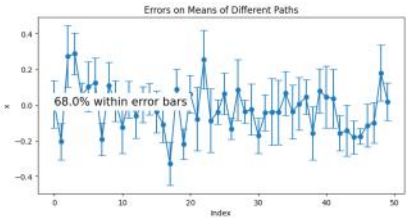
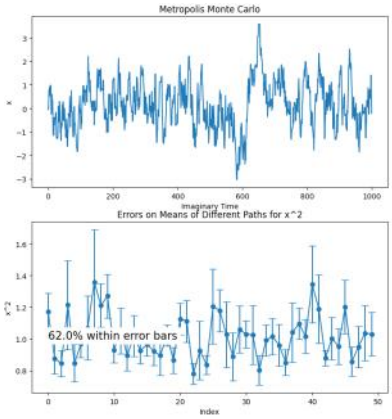
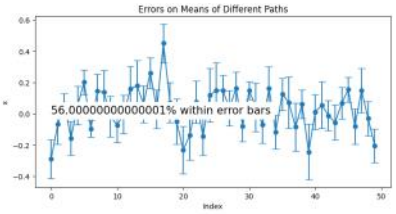
Aim:
To find the expectation values (<x> & <x^2>) from the means and standard errors of my paths.

Sanity Checks:
<x> should roughly be equal to 0 within error bars
<x^2> should roughly be equal to 1 within error bars
Expect to find roughly 68.27% of data points to be equal to the true value within error bars

Steps:
1) Arrays of mean and standard error values from different paths
2) Plot mean and errors of different paths
3) Number within error bars
4) Compare the percentages within error bars compared to 68.27%

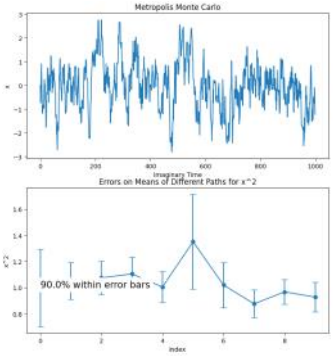
Comparisons:

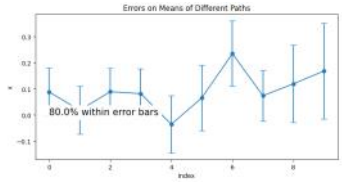
Index	Graphs	Analysis
1		Both lower than expected
2		Both lower than expected
3		Both lower than expected

4			Graph for <x> is about right, but graph for <x^2> is low
5			Both lower than expected

Reflection:

When a large number of paths are taken into account the number within error bars is shown to be low, however I initially did not realise this while testing the code as I was only using 10 paths and getting results of roughly 70%, as shown below:





Going Forward:

I will need to look at why the number within error bars is slightly low

More importantly, it shows how I need to check my code intermittently during testing with large volumes of data points, to avoid this issue from happening again

Testing Cells:

```
#%% Within Error Bars
a = np.array([1,2,3,0,0.3,100,0.3])
b = np.array([1,1,1,1,1,1,10])
tot = 0
for i in range(len(a)):
    if a[i]-b[i]<=0 and a[i]+b[i]>=0:
        tot += 1
print(tot)
```

New Code:

```
"""averages function"""
def meanError(x):
    x_split = np.split(x,10)
    x_split = np.delete(x_split,0,0)
    x_split = np.split(x_split,9)
    x_split_mean = np.array([])
    for i in range(0,len(x_split)):
        x_split_mean = np.append(x_split_mean,np.mean(x_split[i]))
    overall_mean = np.mean(x_split_mean)
    std_err = np.std(x_split_mean)/np.sqrt(len(x_split_mean))
    #print("Overall mean: "+str(overall_mean)+"; Standard Error: "+str(std_err))
    return overall_mean,std_err

"""arrays of mean and standard error values from different paths"""
means = np.array([])
std_errs = np.array([])
for i in range(0,10):
    meanErrors = meanError(updateX())
    means = np.append(means,meanErrors[0])
    std_errs = np.append(std_errs,meanErrors[1])

error_index = np.array([i for i in range(0,len(means))])

"""plot mean and errors of different paths"""
ax2 = fig.add_subplot(222)
ax2.scatter(error_index,means)
ax2.errorbar(error_index,means, yerr=[std_errs,std_errs], capsize=5)
ax2.set_xlabel('Index')
ax2.set_ylabel('x')
ax2.title.set_text('Errors on Means of Different Paths')

"""0 within error bars"""
inErrorBars = 0
for i in range(len(means)):
    if means[i]-std_errs[i]<=0 and means[i]+std_errs[i]>=0:
        inErrorBars += 1
inErrorsPercentage = inErrorBars/len(means)*100
ax2.text(0,0,str(inErrorsPercentage)+'% within error bars',fontsize=15,backgroundcolor='w')

"""same but with x^2"""
means_2 = np.array([])
std_errs_2 = np.array([])
for i in range(0,10):
    meanErrors_2 = meanError(updateX()**2)
    means_2 = np.append(means_2,meanErrors_2[0])
    std_errs_2 = np.append(std_errs_2,meanErrors_2[1])

error_index_2 = np.array([i for i in range(0,len(means_2))])

ax3 = fig.add_subplot(223)
ax3.scatter(error_index_2,means_2)
ax3.errorbar(error_index_2,means_2, yerr=[std_errs_2,std_errs_2], capsize=5)
ax3.set_xlabel('Index')
ax3.set_ylabel('x^2')
ax3.title.set_text('Errors on Means of Different Paths for x^2')

inErrorBars_2 = 0
for i in range(len(means_2)):
    if means_2[i]-std_errs_2[i]<=1 and means_2[i]+std_errs_2[i]>=1:
        inErrorBars_2 += 1
inErrorsPercentage_2 = inErrorBars_2/len(means_2)*100
ax3.text(0,1,str(inErrorsPercentage_2)+'% within error bars',fontsize=15,backgroundcolor='w')
```

Week 5

Supervisor Meeting: 19/10/2022

19 October 2022

Supervisor meeting outline:

- Discussed our results to last week's task.
- Discussed this week's work - the graphs we needed to produce and the theory.
- Discussed what was to come - looking at the N=2 case and eventually N large.

Tasks for the week:

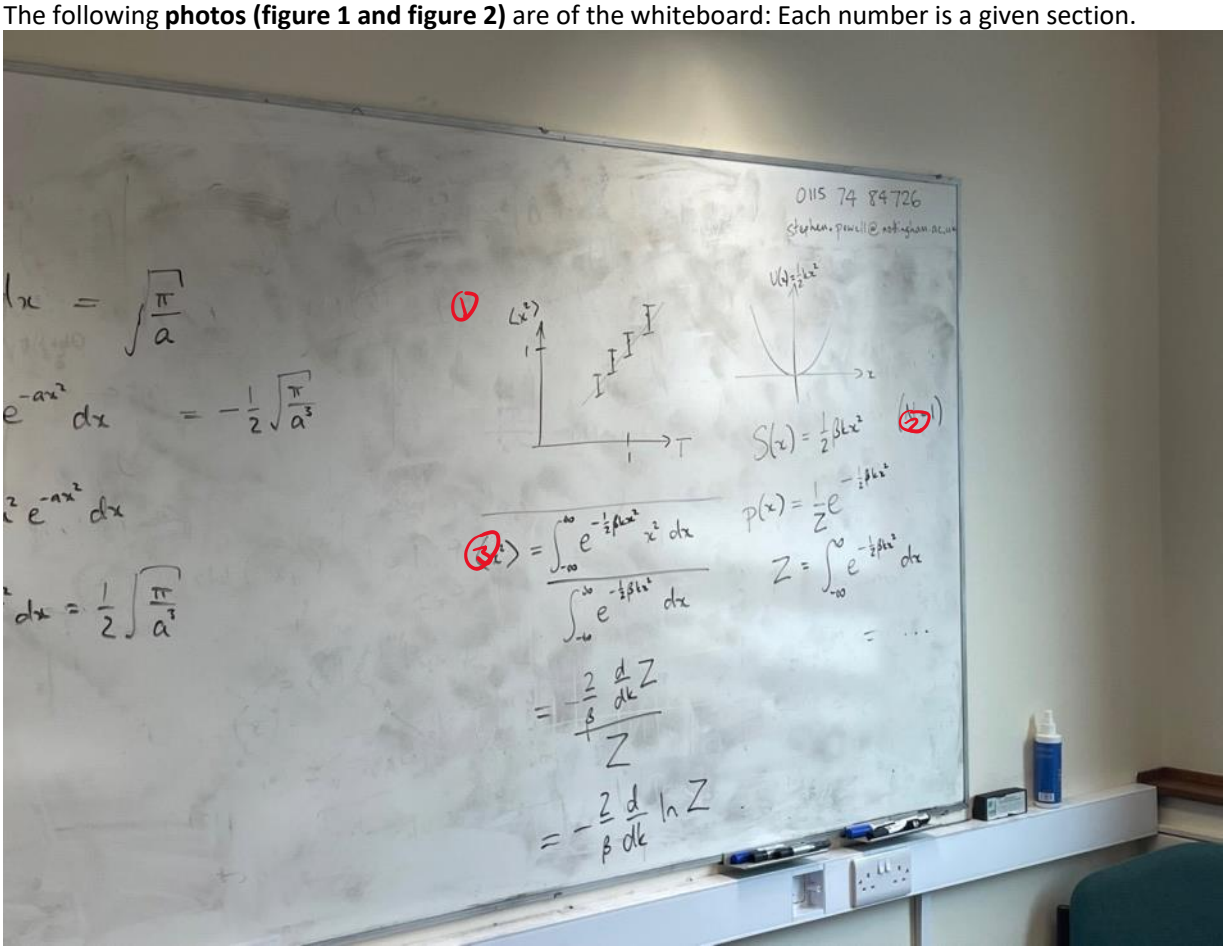
- Produce plots for $\langle x^2 \rangle$ for a range of $\beta = \frac{1}{\mathrm{T}}$ values $T = \{0.2, \ldots., 2\}$
- Compare plots with the expected curve for $\langle x^2 \rangle$ against $\beta = \frac{1}{\mathrm{T}}$

Distribution of tasks:

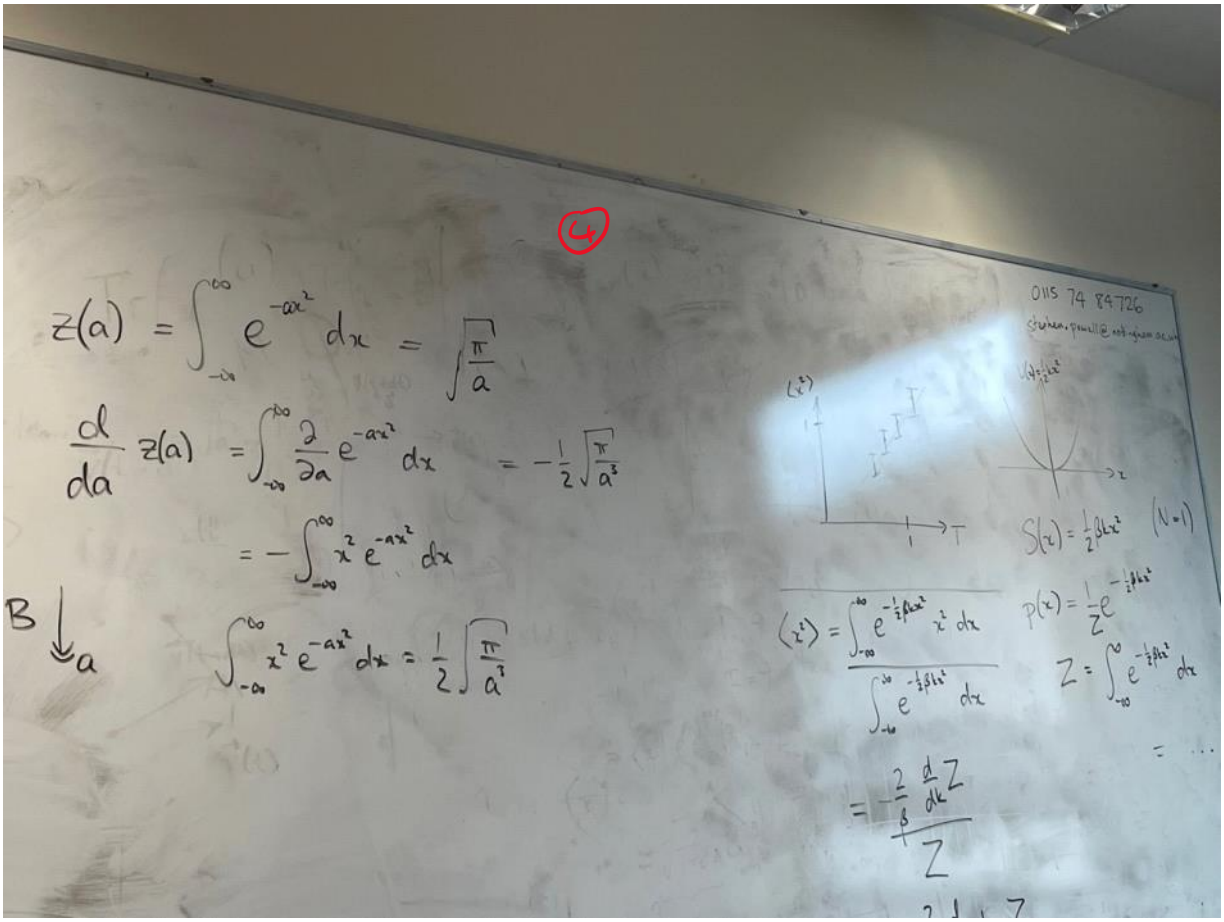
- Both would produce graphs and data using our codes to confirm everything was working.
- We would then compare our results to see if any issues arose.

Project details - as discussed in the meeting:

The following **photos (figure 1 and figure 2)** are of the whiteboard: Each number is a given section.



(Fig 1)



(Fig 2)

1: This was the plot we were to create - $\langle x^2 \rangle$ against T (where $\beta = \frac{1}{T}$). This was to be produced with error bars overlayed with the expected result.

2: Here we spoke about the probability density function. For the simple classical case of N=1, the probability of an accepted point was: $p(x) = \frac{1}{Z}\exp(-S(x)) = \frac{1}{Z}\exp\left(-\frac{1}{2}k\beta x^2\right)$ where $Z=\int_{-\infty}^{\infty}\exp\left(-\frac{1}{2}\beta x^2\right)dx$ ie the parition function that normalises our distribution. This we would be able to work out ananlytically for our plot curve.

3: This calculation looks over how to calculate $\langle x^2 \rangle$ using our probability function. Using the rule of

$$\langle f(x) \rangle = \frac{\int_{-\infty}^{\infty} f(x)\exp\left(-\frac{1}{2}\beta x^2\right)dx}{\int_{-\infty}^{\infty}\exp\left(-\frac{1}{2}\beta x^2\right)dx}$$

Using the chain-rule and results from 4, this formula could be easily rearrange to $\langle x^2 \rangle = -\frac{2}{\beta}\frac{d}{dk}(\ln Z)$

4: This talks about Gaussian integrals, with a refresher on how to calculate more complex ones starting with the known result $\int_{-\infty}^{\infty}\exp(-ax^2)\,dx = \sqrt{\frac{\pi}{a}}$

A few comments:

If $\langle x^2 \rangle$ is worked out for $\beta = \frac{1}{T} = 1$, we get $\langle x^2 \rangle = 1$; the result we got last week.
We expect a percentage within an error of the expected value of around 70% as before - for the same reasons.

(TW) N=1 Statistical Results (19/10/2022)

19 October 2022

$\langle x^2 \rangle \; vs. \beta \; = \frac{1}{T}$ for N=1:

I used the code created last week to produce plots of $\langle x^2 \rangle$ against β and T. The first task was to find the expected curve and then to plot my data on top.

Background:(19/10/2022)
A lot of the theory behind this week's work is very similar to the statistics covered in the Year 2 Module: 'Thermal and Statistical Physics'. That being the partition function and probability densities.

Useful results for Gaussian Integrals - covered in our supervisor meeting and the above module - are:

$\int_{-\infty}^{\infty} \exp(-ax^2)\,dx = \sqrt{\frac{\pi}{a}}$

$\int_{-\infty}^{\infty} x^2 \exp(-ax^2)\,dx = \frac{1}{2}\sqrt{\frac{\pi}{a^3}}$ - This may be worked out by taking the derivative wrt to a of the above integral.

Plan:(19/10/2022)

- Find analytically the curve for $\langle x^2 \rangle$
- Produce plots for $\langle x^2 \rangle$ against T to see if they follow the expected line.
- We expect around 70% of the model's values to be within an error bar of the expected value.

Analytical Result:(19/10/2022)

As we have seen:

$$\langle x^2 \rangle = \frac{-2}{\beta}\frac{d}{dk}(\ln z), \quad z = \int_{-\infty}^{\infty} e^{-\frac{1}{2}k\beta x^2}\,dx = \int_{-\infty}^{\infty} e^{-ax^2}\,dx = \sqrt{\frac{\pi}{a}}$$

where $a = \frac{1}{2}k\beta$

$$\Rightarrow \ln z = \ln\left(\sqrt{\frac{\pi}{\frac{1}{2}k\beta}}\right) = \frac{1}{2}\ln\left(\frac{2\pi}{k\beta}\right)$$

$$= \frac{1}{2}\left(\ln(2\pi) - \ln(k\beta)\right)$$

$$\Rightarrow \frac{d}{dk}(\ln z) = \frac{-1}{2}\cdot\frac{1}{k\beta}\cdot\beta$$

So: $\langle x^2 \rangle = \frac{-2}{\beta}\cdot\frac{-1}{2}\cdot\frac{1}{k\beta}\cdot\beta$

$$= \frac{1}{k\beta}$$

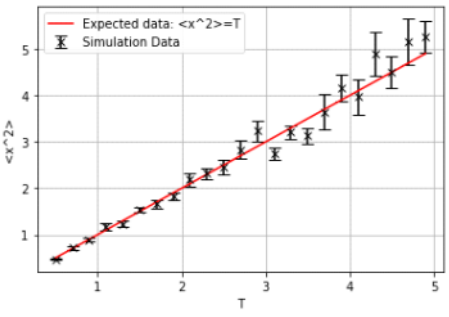
or $\langle x^2 \rangle = \frac{T}{k}$ using $\beta = \frac{1}{k_b T}$ with $k_b = 1$

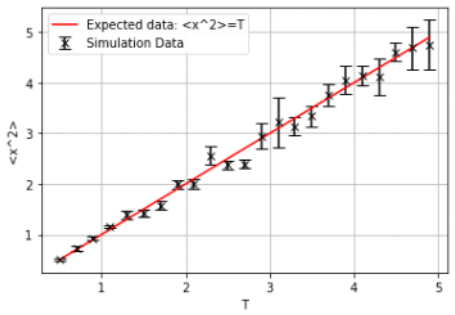
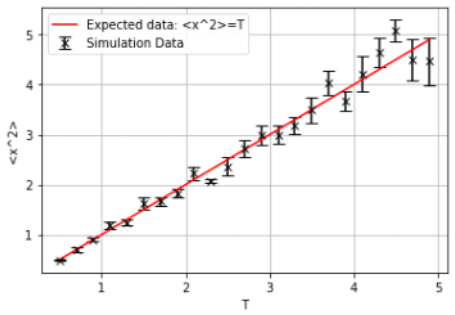
So the **expected value curve is:**

$\langle x^2 \rangle = \frac{1}{k\beta} = \frac{T}{k}$

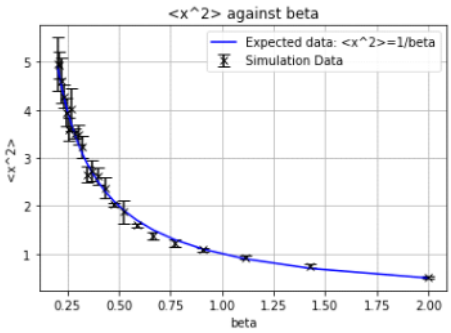
Results:(19/10/2022)

The following graphs are for $\langle x^2 \rangle$ against T. T takes values {0.2,....,2}. The MMC method iterates 10,000 times with all other physical values set to 1.

Run Number	Plot of $\langle x^2 \rangle$	Percentage within an error of the expected value
1		83%
2		

		78%
3		78%

The above graphs follow what we would expect.
Below is also a plot of $\langle x^2 \rangle$ against β - with the same values used as above. This will produce the same plot - just with a graph that goes like $\frac{1}{x}$:

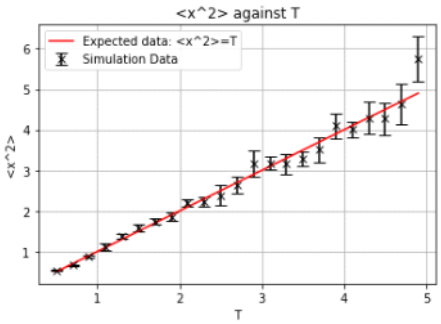


Remarks: (19/10/2022)
It is useful to remark that the value of $\langle x^2 \rangle = 1$ for $\beta = 1$. This is what we observed last time, but were unsure of the reason why.

Conclusions: (19/10/2022)
The points produced by the MMC method followed the expected data for the simple case of N=1 - with upwards of 70% of data within an error bar of the expected value. This meant the code was working correctly - and when expanded to larger N - should give us some useful results to compare to the Quantum Harmonic Oscillator.

Code: (19/10/2022)
The code to produce the plots for $\langle x^2 \rangle$ against T. For plots against beta, the values of the axis were simply changed:

```
N = 1
m = 1
k = 1
it = 10000
T = np.arange(0.5,5,0.2)
beta = 1/T
means = []
errors = []

for i in range(np.size(beta)):
    paths = ((MMC(N, beta[i], m, k, it)[:,0]))**2
    data = errordata(paths)
    means.append(data[0])
    errors.append(data[1])

y = np.asarray(means)

plt.errorbar(T, y, xerr = None, yerr = np.asarray(errors), marker='x', color='black', ecolor='black', linestyle = 'none', capsize = 5, label='Simulation Data' )
plt.ylabel('<x^2>')
plt.xlabel('T')
plt.plot(T, T, color='red', label='Expected data: <x^2>=T')
plt.grid()
plt.legend()
plt.title('<x^2> against T')

```


<x^2> against T
Expected data: <x^2>=T
Simulation Data
<x^2>
T
1
2
3
4
5
6

(TK) N=1: Statistical Results

Wednesday, October 26, 2022 1:45 AM

Aim:
Plot a graph of <x^2> against Temperature

Expected Results:
A plot of points with the line <x^2>=T/k within error bars

Analytical Reasoning:

As shown in reading:

$$\langle x^2 \rangle = \frac{-\frac{2}{\beta}\frac{d}{dk}Z}{Z} = -\frac{2}{\beta}\frac{d}{dk}\ln Z$$

$$\& \; Z = \int_{-\infty}^{\infty} e^{-\frac{1}{2}\beta k x^2}\, dx = \sqrt{\frac{\pi}{a}}, \text{ where } a = \frac{1}{2}\beta k$$

$$= \sqrt{\frac{2\pi}{\beta k}}$$

$$\ln Z = \ln\left(\sqrt{\frac{2\pi}{\beta k}}\right) = \frac{1}{2}\left[\ln(2\pi) - \ln(\beta k)\right]$$

$$\frac{d\ln Z}{dk} = \frac{d\left[\frac{1}{2}\left[\ln(2\pi) - \ln(\beta k)\right]\right]}{dk}$$

$$= \frac{d\left[-\frac{1}{2}\ln(\beta k)\right]}{dk}$$

$$= \frac{-\frac{1}{2}\beta}{\beta k}$$

$$= -\frac{1}{2k}$$

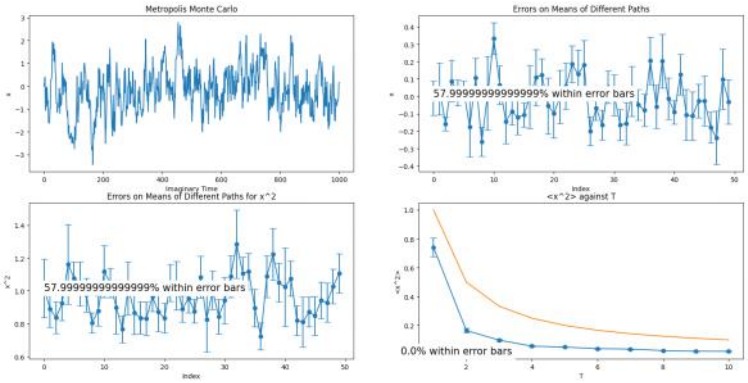
$$\Rightarrow \langle x^2 \rangle = \left(-\frac{2}{\beta}\right)\left(-\frac{1}{2k}\right)$$

$$= \frac{1}{\beta k}$$

$$\& \text{ have } \beta = \frac{1}{T}$$

$$\therefore \langle x^2 \rangle = \frac{T}{k}$$

Plots:



A clear error in the code can be seen here so I will need to go back and correct it in the original function to calculate the means and standard errors.

Potential Errors:

1) Mean and standard error calculation for a path

```
#%% Means
import numpy as np

a = np.array([2,3,4,5,6,7,8,9,10,11])
print(a)
a_split = np.split(a,10)
print(a_split)
a_split = np.delete(a_split,0,0)
print(a_split)
a_split = np.split(a_split,9)
print(a_split)
a_split_mean = np.array([])
for i in range(0,len(a_split)):
    a_split_mean = np.append(a_split_mean,np.mean(a_split[i]))
overall_mean = np.mean(a_split_mean)
std_err = np.std(a_split_mean)/np.sqrt(len(a_split_mean))
print("Overall mean: "+str(overall_mean)+"; Standard Error: "+str(std_err))
```

```
Overall mean: 7.0; Standard Error: 0.8606629658238704
```

Therefore error is not due to this function

2) updateX function:

```
#%% update
def updateX(beta):
    """Pick arbitrary value for x"""
    x = np.array([0])
    for i in range(9):
        """Update x and append"""
        delta = random.uniform(-1,1)
        x = np.append(x, x[-1]+delta)

        """Test update"""
        if S_1(x[-1],beta) <= S_1(x[-2],beta):
            pass
        else:
            prop = np.exp(S_1(x[-2],beta)-S_1(x[-1],beta))
            if random.random() < prop:
                pass
            else:
                x = np.delete(x, -1)
                x = np.append(x, x[-1])
    return x

print(updateX(1))
```

Makes data points fit better

3) Action Function

```
#%% Action Function
"""x array test"""
x = np.array([0.3])

"""values"""
k = 1
beta = 1

"""functions"""
def S_1(x,beta):
    return 0.5*k*beta*x**2

print(S_1(np.array([0.3]),1)) #0.045 as expected
print(S_1(np.array([0.3]),2)) #0.09 as expected
print(S_1(np.array([0.4]),4)) #0.32 as expected
k=2
print(S_1(np.array([0.4]),4)) #0.64 as expected
```

All values return as expected

4) Mean and standard error calculation function

```
#%% Mean & Standard Error
"""averages function"""
def meanError(x):
    x_split = np.split(x,10)
    x_split = np.delete(x_split,0,0)
    x_split = np.split(x_split,9)
    x_split_mean = np.array([])
    for i in range(0,len(x_split)):
        x_split_mean = np.append(x_split_mean,np.mean(x_split[i]))
    overall_mean = np.mean(x_split_mean)
    std_err = np.std(x_split_mean)/(np.sqrt(len(x_split_mean))-1)
    #print("Overall mean: "+str(overall_mean)+"; Standard Error: "+str(std_err))
    return overall_mean,std_err
```

add in

Here I forgot to add in the -1 term for working out the standard error, this makes the code give the expected results

Week 6

Supervisor Meeting: 26/10/2022

26 October 2022

Supervisor meeting outline:

- Discussed our results to last week's task.
- Discussed this week's work:
  - looking at longer paths of N=2,3
  - how to work out the expectation values for longer paths both analytically and using our models
- Also spoke about longer paths such as N=20 and how these would tend to the quantum harmonic oscillator.

Tasks for the week:

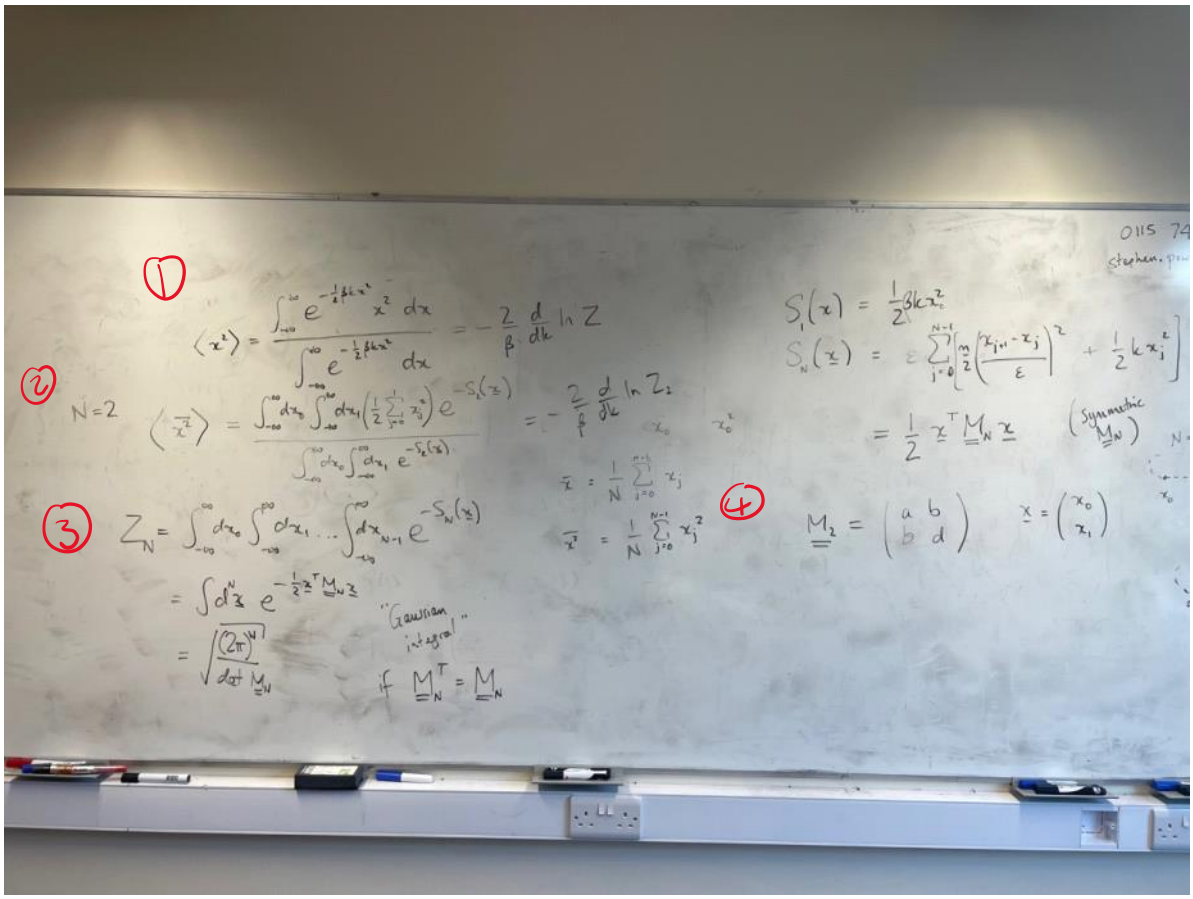
- Ensure that last week's code was working properly, giving the expected results.
- Create a function to find the average x value for a path of N>1 - giving an array of $\bar{x}$ or $\overline{x^2}$
- Use this function to plot $\left\langle \overline{x^2} \right\rangle$ for a range of $\beta = \frac{1}{T}$ values $T = \{0.2, \ldots, 2\}$ for $N = 2,3$
- Compare this plot with the expected analytical results for both cases - using Gaussian Integrals in high dimensions.
- To get the analytical results, check the derivation of $\left\langle \overline{x^2} \right\rangle$ from the Gaussian Integral is correct.

Distribution of tasks:

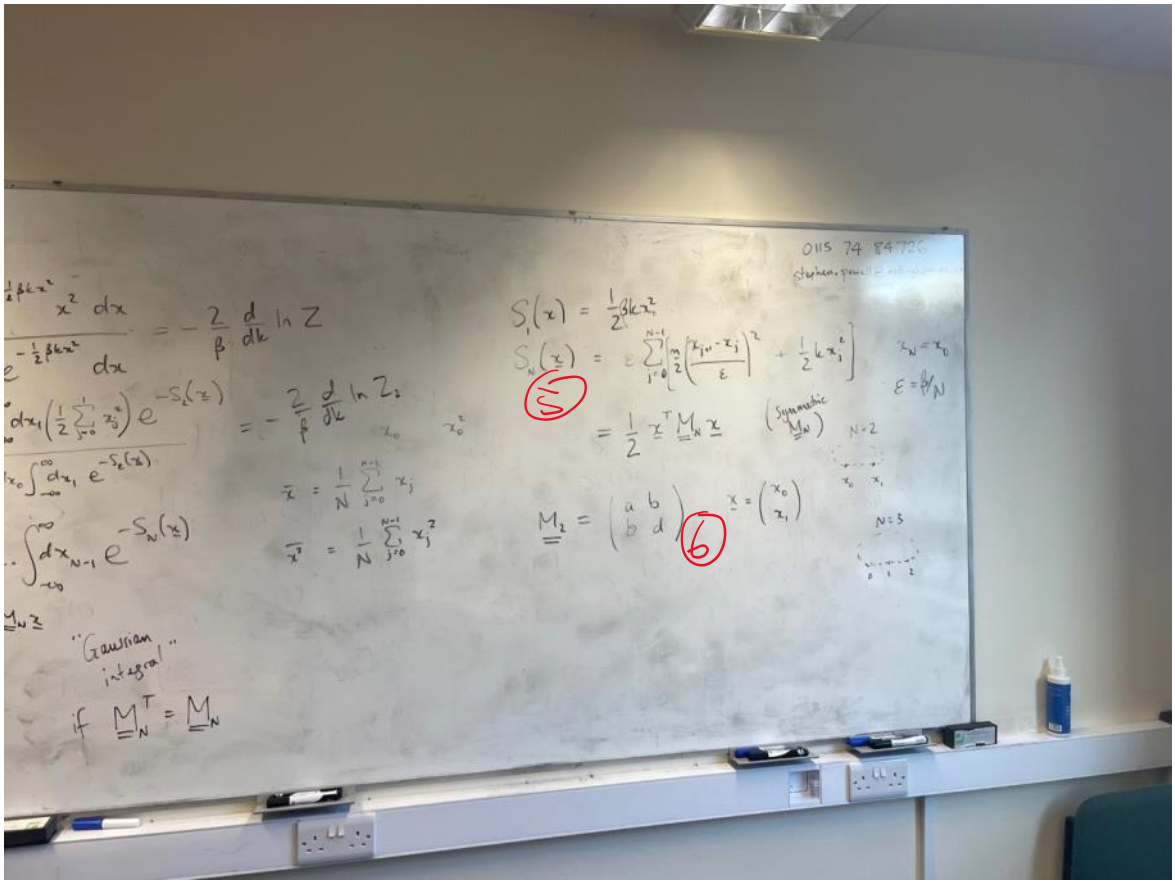
- Both would produce graphs and data using our codes to confirm everything was working.
- We would then compare our results to see if any issues arose - meeting before our next supervisor meeting to see if we can find any errors.

Project details - as discussed in the meeting:

The following **photos (figures 1 & 2)** are of the whiteboard: Each number is a given section.



(Fig 1)



(Fig 2)

1: We recalled the result for $\langle x^2 \rangle$ from last week.

2: Here we looked at calculating the expected value for
$\langle \overline{x^2} \rangle$, using a similar trick to last time to use the partition fucntion. This needed to be checked.

3: This looks at the general definition for the partition function for N dimensions, and the expected result of the corresponding Gaussian Integral. We do not need to know the derivation of this integral, only the result. Some useful background reading (and sources) can be found here:
https://en.wikipedia.org/wiki/Gaussian_integral#Generalizations
The formula only works for symmetrical matrices - but given we are working with symmetrical matrices this isn't a problem.

4: We spoke about the mean value of x, $\bar{x}$ and why we are using it. It is difficult to calculate the standard deviation and statistical values for each point along our path for each path. Instead we you nicer statistical results by finding the average value of x along a given path and creating an array of path averages. This can then be used to find the standard deviation and 'mean of means' as before (using the errordata method). We will use $\overline{x^2}$ for our expectation value analysis.

5: Looks at the general value of the action and how it may be written as a matrix equation (similar to the N dimension Hamiltonian). This matrix can be used in the general Gaussian Integral. We require the matrix to be symmetrical - this can be easily seen when you explicitly work it out.

6: For the case of N=2, we get this matrix.

A few comments:

Working out the matrices for N=2,3 and the analytical results will be enough to confirm our code is working (if the data agrees with these results). As N gets larger, the system tends to the quantum results - it will be more efficient to work out the expectation value using standard quantum methods over the matrix method - the determinant of a 20x20 matrix (and finding the coefficients) would be very difficult.

We mentioned that working out the change in action can be computation heavy; it will be fine for small cases but not for large ones. We discussed how we can streamline this process - we know only the 'potential term' of a given update, the two 'kinetic energy' terms involving that point, will be affected by the update - there is no need to redo the rest of the sum, as it won't have changed.

(TW) N=2 and N=3 Statistical Results and Changing How DS is Calculated (26-27/10/2022)

26 October 2022

$\langle \overline{x^2} \rangle$ vs. $\beta = \frac{1}{T}$:

Using the code created in the preceding weeks, I needed to produce plots for N=2 and N=3 to see if the code produced data that was expected analytically.

Background:(26/10/2022)
Most of the theory was the same as last week, apart from the generalised Gaussian Integral:

$$\int_{-\infty}^{\infty} \exp\left(-\frac{1}{2}x^{\mathrm{T}}Ax\right) d^n x = \sqrt{\frac{(2\pi)^n}{\det A}}$$

Working out the matrix form of the action is very similar to working out the Hamiltonian for N particles, which we did in the module 'Introduction to Mathematical Physics' last year (2022).
The discussion as to why $\langle x^2 \rangle$ was used instead of $\langle x^2 \rangle$ can be found in the 'supervisor meeting' notes section.

Plan:(26/10/2022)

- Ensure the calculation for finding $\langle \overline{x^2} \rangle$ as discussed is correct, regarding Z_n
- Produce an average method to find the average value of x for a given path and test.
- Use this function to find $\langle \overline{x^2} \rangle$ for the case N=2 and N=3. This can be done in a very similar way to last week.
- Find analytically the expected curves for N=2 and N=3.
- Using these curves plot the experimental data error bars with the expected value - roughly 70% of the points should agree as before (for a large enough number of iterations).
- Streamline the calculation for ΔS

Partition function Results:(26/10/2022)

$\langle \overline{x^2} \rangle = -\frac{2}{\beta}\frac{d}{dk}\ln Z_2$ for N = 2, where $\epsilon = \frac{\beta}{N_\beta} = \frac{\beta}{2}$

We know: $\langle \overline{x^2} \rangle = \dfrac{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \left(\frac{1}{2}\sum_{j=0}^{1} x_j^2\right) e^{-S_2(x_j)}}{\left\{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 e^{-S_2(x_j)}\right\} Z_2}$

Want to confirm: $\langle \overline{x^2} \rangle = \frac{-2}{\beta}\frac{d}{dk}\ln Z_2 = \frac{-2}{\beta}\frac{\frac{d}{dk}Z_2}{Z_2}$

$$\frac{d}{dk}Z_2 = \int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \frac{d}{dk}\left(e^{-S_2(x_j)}\right)$$

$$= \int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \, (-1)\cdot\frac{d}{dk}\left(S_2(x_j)\right) e^{-S_2(x_j)}$$

Given $S_2(x_j) = \varepsilon\left[\frac{m}{2}\left(\frac{x_1 - x_0}{\varepsilon}\right)^2 + \frac{1}{2}kx_0^2 + \frac{m}{2}\left(\frac{x_2 - x_1}{\varepsilon}\right)^2 + \frac{1}{2}kx_1^2\right]$ ($x_2 = x_0$)

$$\Rightarrow \frac{d}{dk}S_2(x_j) = \frac{\varepsilon}{2}x_0^2 + \frac{\varepsilon}{2}x_1^2 = \frac{\varepsilon}{2}\left(\sum_{j=0}^{1} x_j^2\right)$$

$$\Rightarrow \frac{d}{dk}Z_2 = \int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \, -\frac{\varepsilon}{2}\left(\sum_{j=0}^{1} x_j^2\right) e^{-S_2(x_j)} \quad \text{NB: } \varepsilon = \frac{\beta}{2}$$

So: $\dfrac{\frac{-2}{\beta}\frac{d}{dk}Z_2}{Z_2} = \dfrac{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \left(\frac{1}{2}\sum_{j=0}^{1} x_j^2\right) e^{-S_2(x_j)}}{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 e^{-S_2(x_j)}} \cdot \frac{-2}{\beta} = \langle \overline{x^2} \rangle.$

So the assumption is correct.

The same works for the case of N=3, and it can be seen that it will work for N>3 also:

$$\langle \overline{x^2} \rangle = \frac{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 \, \frac{1}{3}\left(\sum_{j=0}^{2} x_j^2\right) e^{-S_3(x_j)}}{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 \, e^{-S_3(x_j)}}$$

Assuming the same result:

$$\frac{d}{dk}Z_3 = \int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 \, e^{-S_3(x_j)} \cdot (-1)\frac{d}{dk}\left(S_3(x_j)\right)$$

$$S_3(x_j) = \varepsilon\left[\frac{m}{2}\left(\frac{x_1 - x_0}{\varepsilon}\right)^2 + \frac{1}{2}kx_0^2 + \frac{m}{2}\left(\frac{x_2 - x_1}{\varepsilon}\right)^2 + \frac{1}{2}kx_1^2 + \frac{m}{2}\left(\frac{x_3 - x_2}{\varepsilon}\right)^2 + \frac{1}{2}kx_2^2\right] \quad (x_3 = x_0)$$

$$\frac{d}{dk}S_3(x_j) = \frac{\varepsilon}{2}\left(x_0^2 + x_1^2 + x_2^2\right) = \frac{\varepsilon}{2}\sum_{j=0}^{2} x_j^2$$

So $\frac{d}{dk}Z_3 = \int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 \, e^{-S_3(x_j)} \frac{-\varepsilon}{2}\sum_{j=0}^{2} x_j^2$

with $\varepsilon = \frac{\beta}{3}$

So $\dfrac{\frac{-2}{\beta}\frac{d}{dk}Z_3}{Z_3} = \dfrac{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 \, e^{-S_3(x_j)} \frac{1}{2}\cdot\frac{\beta}{3}\sum_{j=0}^{2} x_j^2}{\int_{-\infty}^{\infty} dx_0 \int_{-\infty}^{\infty} dx_1 \int_{-\infty}^{\infty} dx_2 \, e^{-S_3(x_j)}} \cdot \frac{-2}{\beta} = \langle \overline{x^2} \rangle$

Average Function Plan and Code:(26/10/2022)
This is a simple function that takes an mxn array, taking the average value of a given row (that being one path) and returning a 1xn array of the average value of x for a given path.

```python
def pathavg(xarr):
    #This function takes a array of paths, and find the average x position along each path
    #And returns an array of averages - each entry is the average of the corresponding path index number.
    means = []
    for i in range(np.shape(paths)[0]):
        #This finds the number of rows, and interates each one in tern
        s = 0
        for j in range(np.shape(paths)[1]):
            #This finds the number of columns, and iterates each one in tern
            s += paths[i][j]
        means.append(s/np.shape(paths)[1])
    return np.asarray(means)
```

I tested this array on a path of length 2 and 3, calculating the average value of x after a selected few updates to see if I agreed with the code. I did, so I was happy to continue.

The case of N=2:(26/10/2022)

I first focused on the case of N=2, finding the analytical value for $\langle \bar{x^2} \rangle$ and producing a few plots against T, similar to last week.

Analytical solution:

$$S_2(\underline{x}) = \varepsilon \left[\frac{m}{2}\left(\frac{x_1 - x_0}{\varepsilon}\right)^2 + \frac{1}{2}kx_0^2 + \frac{m}{2}\left(\frac{\overbrace{x_2}^{x_2 = x_0} - x_1}{\varepsilon}\right)^2 + \frac{1}{2}kx_1^2 \right]$$

$$= \frac{m}{2\varepsilon}\left(x_1^2 - 2x_1x_0 + x_0^2\right) + \frac{m}{2\varepsilon}\left(x_0^2 - 2x_1x_0 + x_1^2\right) + \frac{\varepsilon k}{2}\left(x_0^2 + x_1^2\right)$$

$$= x_0^2\left(\frac{m}{\varepsilon} + \frac{\varepsilon k}{2}\right) + 2x_1x_0\left(\frac{-m}{\varepsilon}\right) + x_1^2\left(\frac{m}{\varepsilon} + \frac{\varepsilon k}{2}\right)$$

$$= \frac{1}{2}\left[x_0^2\left(\frac{2m}{\varepsilon} + \varepsilon k\right) + 2x_1x_0\left(\frac{-2m}{\varepsilon}\right) + x_1^2\left(\frac{2m}{\varepsilon} + \varepsilon k\right)\right]$$

We let $S_2(\underline{x}) = \frac{1}{2}(x_0, x_1)\begin{bmatrix} a & b \\ b & c \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$

$$= \frac{1}{2}\left(ax_0^2 + 2bx_1x_0 + cx_1^2\right)$$

$$\Rightarrow a = \frac{2m}{\varepsilon} + \varepsilon k = c$$

$$b = \frac{-2m}{\varepsilon}$$

ie: $\underline{\underline{M}}_2 = \begin{bmatrix} \frac{2m}{\varepsilon} + \varepsilon k & \frac{-2m}{\varepsilon} \\ \frac{-2m}{\varepsilon} & \frac{2m}{\varepsilon} + \varepsilon k \end{bmatrix}$

$$\Rightarrow \det \underline{\underline{M}}_2 = \left(\frac{2m}{\varepsilon} + \varepsilon k\right)^2 - \frac{4m^2}{\varepsilon^2}$$

$$= 4mk + \varepsilon^2 k^2$$

- Using $\langle \bar{x}^2 \rangle = \frac{-2}{\beta}\frac{d}{dk}\ln Z_N$; for N=2 :

$$Z_2 = \int_{-\infty}^{\infty} d^2\underline{x}\, e^{-\frac{1}{2}\underline{x}^T \underline{\underline{M}}_2 \underline{x}} = \sqrt{\frac{(2\pi)^2}{\det \underline{\underline{M}}_2}} = \frac{2\pi}{\sqrt{4mk + \varepsilon^2k^2}}$$

$$\Rightarrow \ln Z_2 = \ln 2\pi - \frac{1}{2}\ln\left(4mk + \varepsilon^2k^2\right)$$

$$\Rightarrow \frac{d}{dk}\ln Z_2 = \frac{-1}{2}\cdot\frac{1}{4mk + \varepsilon^2k^2}\cdot(4m + 2\varepsilon^2 k)$$

- So $\langle \bar{x}^2 \rangle = \frac{-2}{\beta}\cdot\frac{-1}{2}\cdot\frac{4m + 2\varepsilon^2 k}{4mk + \varepsilon^2k^2}$; using $\varepsilon = \frac{\beta}{2}$:

$$\Rightarrow \langle \bar{x}^2 \rangle = \frac{1}{\beta}\cdot\frac{4m + \frac{\beta^2 k}{2}}{4mk + \frac{\beta^2k^2}{4}} = \frac{4m + \frac{\beta^2 k}{2}}{4mk\beta + \frac{\beta^3k^2}{4}}$$

$$= \frac{16m + 2\beta^2 k}{16mk\beta + \beta^3k^2}.$$ Setting $m = k = 1$, $\beta = \frac{1}{k_B T} = \frac{1}{T}$ we have:

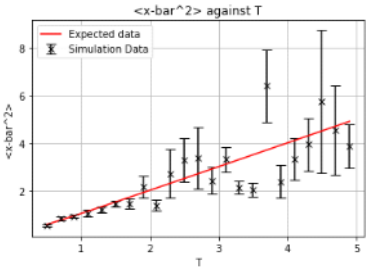
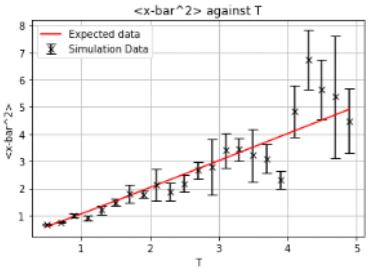
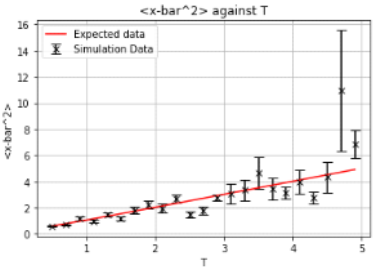
$$\langle \bar{x}^2 \rangle = \frac{16 + 2\beta^2}{16\beta + \beta^3} = \frac{16 + \frac{2}{T^2}}{\frac{16}{T} + \frac{1}{T^3}} = \frac{16T^3 + 2T}{16T^2 + 1} \quad //$$

With this result I could plot, in the same way as last week, $\langle \bar{x^2} \rangle$ against T along with the expected curve $\langle \bar{x^2} \rangle = \frac{16T^3+2T}{16T^2+1}$

Results:

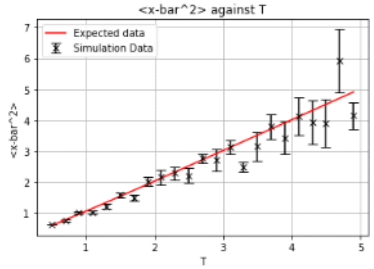
The following graphs are for $\langle \bar{x^2} \rangle$ against T. T takes values {0.2,….,2}. The MMC method iterates 10,000 times with all other physical values set to 1, and N=2.

Run Number	Plot of $\langle \bar{x^2} \rangle$ against T for N=2	Percentage within an error of the expected value
1		74%

		
2		70%
3		74%

Conclusion:
The simulation data followed the analytical result within the range expected, for N=2. This makes it more likely that the simulation is working correctly for large N, but will first check with N=3 before going for large N. The next steps are to produce similar data for N=3, but first I will streamline the ΔS calculation to help run-time.

NB by increasing the number of iterations, the points got closer to the expected curve as expected:



This plot is for 50,000 iterations in the MMC method - it takes a lot longer to produce. The points are ~80% within the expected data.

Code:
The code for these plots is very similar to last week, just using the pathsavg function before calculating any data.

```
N = 2
m = 1
k = 1
it = 50000
T = np.arange(0.5,5,0.2)
beta = 1/T
means = []
errors = []

for i in range(np.size(beta)):
    paths = ((MMC(N, beta[i], m, k, it))**2)
    pathsm = pathavg(paths)
    data = errordata(pathsm)
    means.append(data[0])
    errors.append(data[1])

y = np.asarray(means)
plt.errorbar(T, y, xerr = None, yerr = np.asarray(errors), marker='x', color='black', ecolor='black', linestyle = 'none', capsize = 5, label='Simulation Data' )
plt.ylabel('<x-bar^2>')
plt.xlabel('T')
plt.plot(T, (16*T**3+2*T)/(16*T**2+1), color='red', label='Expected data')
plt.grid()
plt.legend()
plt.title('<x-bar^2> against T')
```

The case of N=3:(26/10/2022)
The procedure was similar to the case above.

Analytical solution:

$$S_3(\beta) = \varepsilon \sum_{i=0}^{\varepsilon} \left(\frac{m}{2} \left(\frac{x_{i+1} - x_i}{\varepsilon} \right)^2 + \frac{1}{2} k x_i^2 \right)$$

$$= \varepsilon \left[\frac{m}{2} \left(\frac{x_1 - x_0}{\varepsilon} \right)^2 + \frac{1}{2} k x_0^2 + \frac{m}{2} \left(\frac{x_2 - x_1}{\varepsilon} \right)^2 + \frac{1}{2} k x_1^2 + \frac{m}{2} \left(\frac{x_3 - x_2}{\varepsilon} \right)^2 + \frac{1}{2} k x_2^2 \right]$$

$x_3 = x_0$.

$$\left[\frac{m}{2\varepsilon}\left(\frac{x_3-x_0}{\varepsilon}\right)^2+\frac{1}{2}kx_0^2\right]$$

$x_3 = x_0$.

$$= \frac{m}{2\varepsilon}\left[x_1^2 - 2x_1x_0 + x_0^2 + x_2^2 - 2x_2x_1 + x_1^2 + x_0^2 - 2x_2x_0 + x_0^2\right]$$

$$+ \frac{\varepsilon k}{2}\left[x_0^2 + x_1^2 + x_2^2\right]$$

$$= x_0^2\left(\frac{m}{\varepsilon} + \frac{\varepsilon k}{2}\right) + x_1^2\left(\frac{m}{\varepsilon} + \frac{\varepsilon k}{2}\right) + x_2^2\left(\frac{m}{\varepsilon} + \frac{\varepsilon k}{2}\right) + 2x_1x_0\left(\frac{-m}{\varepsilon^2}\right)$$

$$+ 2x_2x_1\left(\frac{-m}{\varepsilon^2}\right) + 2x_1x_0\left(\frac{-m}{\varepsilon^2}\right)$$

$$= \frac{1}{2}\left[x_0^2\left(\frac{2m}{\varepsilon} + \varepsilon k\right) + x_1^2\left(\frac{2m}{\varepsilon} + \varepsilon k\right) + x_2^2\left(\frac{2m}{\varepsilon} + \varepsilon k\right) + 2x_1x_0\left(\frac{-m}{\varepsilon}\right)\right.$$

$$\left. + 2x_2x_1\left(\frac{-m}{\varepsilon}\right) + 2x_1x_0\left(\frac{-m}{\varepsilon}\right)\right]$$

Also: $\langle \bar{x}^2 \rangle = \frac{1}{2}(x_0, x_1, x_2)\begin{bmatrix} a & b & c \\ b & e & f \\ c & f & i \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$

NB: $\underline{\underline{M}}_3$ is symmetrical.

$$= \frac{1}{2}(ax_0^2 + ex_1^2 + ix_2^2 + 2bx_1x_0 + 2cx_2x_0 + 2fx_2x_1)$$

$$\Rightarrow a = e = i = \frac{2m}{\varepsilon} + \varepsilon k$$

$$b = c = f = \frac{-m}{\varepsilon}$$

$$\Rightarrow \underline{\underline{M}}_3 = \begin{bmatrix} \frac{2m}{\varepsilon} + \varepsilon k & \frac{-m}{\varepsilon} & \frac{-m}{\varepsilon} \\ \frac{-m}{\varepsilon} & \frac{2m}{\varepsilon} + \varepsilon k & \frac{-m}{\varepsilon} \\ \frac{-m}{\varepsilon} & \frac{-m}{\varepsilon} & \frac{2m}{\varepsilon} + \varepsilon k \end{bmatrix}$$

$$\Rightarrow \det \underline{\underline{M}}_3 = \left(\frac{2m}{\varepsilon} + \varepsilon k\right)\begin{vmatrix} \frac{2m}{\varepsilon} + \varepsilon k & \frac{-m}{\varepsilon} \\ \frac{-m}{\varepsilon} & \frac{2m}{\varepsilon} + \varepsilon k \end{vmatrix} + \frac{m}{\varepsilon}\begin{vmatrix} \frac{-m}{\varepsilon} & \frac{-m}{\varepsilon} \\ \frac{-m}{\varepsilon} & \frac{2m}{\varepsilon} + \varepsilon k \end{vmatrix}$$

$$- \frac{m}{\varepsilon}\begin{vmatrix} \frac{-m}{\varepsilon} & \frac{2m}{\varepsilon} + \varepsilon k \\ \frac{-m}{\varepsilon} & \frac{-m}{\varepsilon} \end{vmatrix}$$

Which may be shown to be:

$$\det \underline{\underline{M}}_3 = \frac{9m^2k}{\varepsilon} + 6m\varepsilon k^2 + \varepsilon^3 k^3 \quad \text{(checked for } \varepsilon = m = k = 1\text{, correct)}.$$

$$\Rightarrow Z_3 = \sqrt{\frac{(2\pi)^3}{\frac{9m^2k}{\varepsilon} + 6m\varepsilon k^2 + \varepsilon^3 k^3}} = \int_{-\infty}^{\infty} d^3\underline{x}\, e^{-\frac{1}{2}\underline{x}^T \underline{\underline{M}}_3 \underline{x}}$$

$$\Rightarrow \ln Z_3 = \frac{1}{2}\left(3\ln 2\pi - \ln\left(\frac{9m^2k}{\varepsilon} + 6m\varepsilon k^2 + \varepsilon^3 k^3\right)\right)$$

$$\Rightarrow \frac{d}{dk}\ln Z_3 = \frac{-1}{2} \cdot \frac{1}{\frac{9m^2k}{\varepsilon} + 6m\varepsilon k^2 + \varepsilon^3 k^3}\left(\frac{9m^2}{\varepsilon} + 12m\varepsilon k + 3\varepsilon^3 k^2\right)$$

$$\therefore \langle \bar{x}^2 \rangle = \frac{-2}{\beta}\frac{d}{dk}\ln Z_3 = \frac{1}{\beta}\,\frac{\frac{9m^2}{\varepsilon} + 12m\varepsilon k + 3\varepsilon^3 k^2}{\frac{9m^2k}{\varepsilon} + 6m\varepsilon k^2 + \varepsilon^3 k^3}$$

Which may be shown to be: using $\varepsilon = \frac{\beta}{3}$

$$\langle \bar{x}^2 \rangle = \frac{\frac{27m^2}{\beta} + 4m\beta k + \frac{\beta^3 k^2}{9}}{27m^2k + 2m\beta^2 k^2 + \frac{\beta^4 k^3}{27}}$$

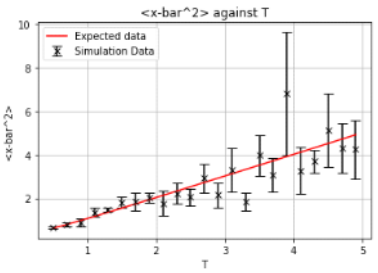
using $\beta = \frac{1}{T}$ and $m = k = 1$:

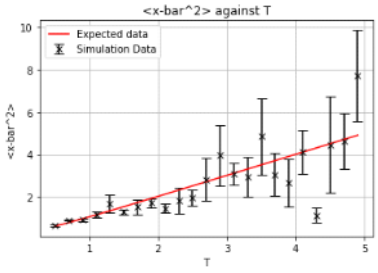
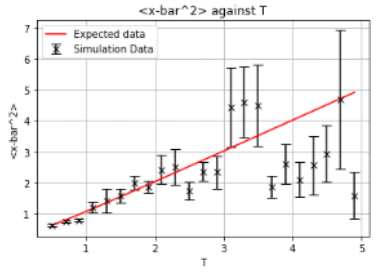
$$\langle \bar{x}^2 \rangle = \frac{729T^5 + 108T^3 + 3T}{729T^4 + 54T^2 + 1}$$ (after some algebra). //

With this result I could plot, in the same way as last week, $\langle \overline{x^2} \rangle$ against T along with the expected curve $\langle \overline{x^2} \rangle = \frac{729T^5+108T^3+3T}{729T^4+54T^2+1}$

Results:

The following graphs are for $\langle \overline{x^2} \rangle$ against T. T takes values {0.2,….,2}. The MMC method iterates 20,000 (more iterations were needed to give cleaner results due to longer paths) times with all other physical values set to 1, and N=3.

Run Number	Plot of $\langle \overline{x^2} \rangle$ against T for N=3	Percentage within an error of the expected value
1		83%
2		

		74%
3		61%

Code:
The code for these plots is the same as above but with a different expected function, and N=3:

```
N = 3
m = 1
k = 1
it = 20000
T = np.arange(0.5,5,0.2)
beta = 1/T
means = []
errors = []

for i in range(np.size(beta)):
    paths = ((MMC(N, beta[i], m, k, it))**2)
    pathsm = pathavg(paths)
    data = errordata(pathsm)
    means.append(data[0])
    errors.append(data[1])

y = np.asarray(means)
plt.errorbar(T, y, xerr = None, yerr = np.asarray(errors), marker='x', color='black', ecolor='black', linestyle = 'none', capsize = 5, label='Simulation Data' )
plt.ylabel('<x-bar^2>')
plt.xlabel('T')
plt.plot(T, (729*T**5+108*T**3+3*T)/(729*T**4+54*T**2+1), color='red', label='Expected data')
plt.grid()
plt.legend()
plt.title('<x-bar^2> against T')
```

Conclusion:
The simulation data followed the analytical result within the range expected for N=3.

Overall Conclusions (26/10/2022)**:**
Given the plots for N=1, N=2 and N=3 agree with their respective curves, I am happy that my Monte Carlo method produces the paths correctly. The next step will be to increase N to simulate quantum systems and use known expectation values of the harmonic oscillator to compare our model against. I am also going to streamline my ΔS calculation first, as this will speed up the algorithm for larger N.

Tasks for next time/week:
- Streamline the ΔS calculation (see below - 27/10/2022).
- Begin looking at the case for N large, including finding the theoretical expected value.

Streamlining the ΔS calculation: (27/10/2022)

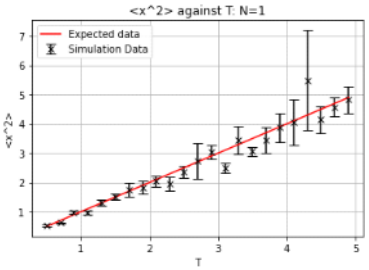
As mentioned above, the calculation for ΔS is more intensive than it needs to be. As only one point is updated at a time, this creating a new path, the change in action between the old and updated paths is just the 'kinetic energy terms' for which the new point is involved, and the 'potential energy term' of the updated point. So only three quanities need to be found rather than a sum over many - this is will make the MMC method more efficient - especially for large N. The code for the update is below, and was tested with the old code to ensure the calculations were the same.

```python
def MMC(N, beta, m, k, iterations):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    eps = beta/N #created epslion from beta and N provided
    S_1 = 0 #action will always start as 0 for x=0
    paths = [x] #a list to store all paths

    #iterations-1 to ensure we have the number of paths = 'iterations' (otherwise would be iterations+1 due to starting path)
    for i in range(iterations-1):
        #This is the MMC method
        #We first choose a random position to alter from x_0 to x_n-1
        pos = random.randint(0,N-1)
        #the delta is a random number between two end points to alter the chosen x by
        delta = random.uniform(-1, 1)
        #creates a copy of the current path to alter
        xtemp = np.copy(x)
        #we set a the chosen point in the copied array to its new point y_2
        y_1 = xtemp[pos]
        y_2 = y_1 + delta
        xtemp[pos] = y_2
        #this ensures that the boundary condition holds,
        #that is that the start and end of the path must be the same value
        xtemp[N] = xtemp[0]

        #The calculation for DS has been streamlined, all that needs to be calculated are the three terms that change due to the update
        #This means the claculation will be a lot quicker for large N. The rest of the method is unchanged.
        #DS is worked out first, and from that we can find the new action S_2, which is stored in case the updated path is accepted.
        DS = eps*((m/2)*((xtemp[pos+1]-xtemp[pos])/eps)**2 + (m/2)*((xtemp[pos]-xtemp[pos-1])/eps)**2 + 1/2*k*(xtemp[pos])**2)
        S_2 = S_1+DS

        if DS <= 0:
            x = np.copy(xtemp)
            S_1 = S_2
        else:
            p = np.exp(-DS)
            r = random.uniform(0,1)
            if r < p:
                x = np.copy(xtemp)
                S_1 = S_2
        #the path x, update or the original depending on the above calculation, is added to the list of paths
        paths.append(x)
    #once the number of iterations is reached, the function produces an array of paths
    return np.asarray(paths)
```

Change for DS calculation

16/11/2022 There is an error here, mentioned and fixed in week 8/9.

I produced a simple plot for $\langle x^2 \rangle$ against T for N=1 (the same as last week, 19/10/2022), to confirm the results were the same for this updated method. They were (83% within an error bar of the expected value) so I was happy with the change:



TK

Saturday, October 29, 2022 1:44 PM

Please expand tab for full weekly report

Theory

Saturday, October 29, 2022 1:50 PM

Aim:

To produce plots for N=2 and N=3 for $< \overline{x^2} > against\ \beta = \frac{1}{T}$, and compare these results against the expected curves, which are calculated below.

Background:

Action in matrix form for N points: $S_N(\boldsymbol{x}) = \varepsilon \sum_{j=0}^{N-1} \left[\frac{m}{2} \left(\frac{x_{j+1} - x_j}{\varepsilon} \right)^2 + \frac{1}{2} k x_j^2 \right] = \frac{1}{2} \boldsymbol{x}^T \boldsymbol{M}_N \boldsymbol{x}$, $\boldsymbol{M}_N$ is symmetric

Partition function for N points: $Z_N = \sqrt{\frac{(2\pi)^N}{\det \boldsymbol{M}_N}}$, if $\boldsymbol{M}_N^T = \boldsymbol{M}_N$

$$< \overline{x^2} > = \frac{\int_{-\infty}^{\infty} e^{-\frac{1}{2}\beta k x^2} x^2 \, dx}{\int_{-\infty}^{\infty} e^{-\frac{1}{2}\beta k x^2} \, dx} = -\frac{2}{\beta} \left(\frac{d}{dk} \right) \ln Z_N$$

Plan:

- Find $\boldsymbol{M}_N$ for N=2 & N=3
- Find Z_N for N=2 & N=3
- Derive an equation for $< \overline{x^2} >$ with respect to temperature for N=2 & N=3.
- Produce a plot for N=2 & N=3 with the expected curve for $< \overline{x^2} >$ with respect to temperature, and check percentage of points within error bars
    - I expect the percentage of points within error bars to the expected curve to be >68.2%

Method:

N=2

$$S_2 = \varepsilon \left[\frac{m}{2} \left(\frac{x_1 - x_0}{\varepsilon} \right)^2 + \frac{1}{2} k x_0^2 + \frac{m}{2} \left(\frac{x_0 - x_1}{\varepsilon} \right)^2 + \frac{1}{2} k x_1^2 \right]$$

$$= \varepsilon \left[\frac{m}{2\varepsilon^2} \left((x_1 - x_0)^2 + (x_0 - x_1)^2 \right) + \frac{1}{2} k (x_0^2 + x_1^2) \right]$$

$$= \varepsilon \left[\frac{m}{2\varepsilon^2} \left(x_1^2 - 2x_0 x_1 + x_0^2 + x_0^2 - 2x_0 x_1 + x_1^2 \right) + \frac{1}{2} k (x_0^2 + x_1^2) \right]$$

$$= \varepsilon \left[\frac{m}{2\varepsilon^2} \left(2x_1^2 + 2x_0^2 - 4x_0 x_1 \right) + \frac{1}{2} k (x_0^2 + x_1^2) \right]$$

$$= x_0^2 \left(\frac{m}{\varepsilon} + \frac{1}{2} k \varepsilon \right) + x_1^2 \left(\frac{m}{\varepsilon} + \frac{1}{2} k \varepsilon \right) + x_0 x_1 \left(\frac{-2m}{\varepsilon} \right)$$

$$S_2(\underline{x}) = \frac{1}{2} (x_0, x_1) \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$= \frac{1}{2} (x_0, x_1) \begin{pmatrix} a x_0 + b x_1 \\ c x_0 + d x_1 \end{pmatrix}$$

$$= \frac{1}{2} \left(a x_0^2 + b x_0 x_1 + c x_0 x_1 + d x_1^2 \right)$$

$$= \frac{1}{2} \left(a x_0^2 + (b + c) x_0 x_1 + d x_1^2 \right)$$

$\therefore a = d = \frac{2m}{\varepsilon} + k\varepsilon$

& $b + c = \frac{-4m}{\varepsilon}$

& $\underline{\underline{M}}_2$ is symmetric $\Rightarrow b = c = \frac{-2m}{\varepsilon}$

$$\therefore \underline{\underline{M}}_2 = \begin{pmatrix} \frac{2m}{\varepsilon} + k\varepsilon & \frac{-2m}{\varepsilon} \\ \frac{-2m}{\varepsilon} & \frac{2m}{\varepsilon} + k\varepsilon \end{pmatrix}$$

$$\det \underline{\underline{M}}_2 = \left(\frac{2m}{\varepsilon} + k\varepsilon \right)^2 - \left(\frac{-2m}{\varepsilon} \right)^2$$

$$= \frac{4m^2}{\varepsilon^2} + 4mk + k^2\varepsilon^2 - \frac{4m^2}{\varepsilon^2}$$

$$= 4mk + k^2\varepsilon^2$$

$$Z_2 = \frac{(2\pi)^2}{\sqrt{\det \underline{\underline{M}}_2}}$$

$$= \sqrt{\frac{4\pi^2}{4mk + k^2\varepsilon^2}}$$

$$\langle \overline{x^2} \rangle = -\frac{2}{\beta}\frac{1}{dk}\ln\left(\frac{2\pi}{(4mk+k^2\varepsilon^2)^{1/2}}\right)$$

$$= -\frac{2}{\beta}\left(\frac{(4mk+k^2\varepsilon^2)^{1/2}}{2\pi}\right)\frac{d}{dk}\left(2\pi(4mk+k^2\varepsilon^2)^{-1/2}\right)$$

$$= \frac{2}{\beta}\left(\frac{(4mk+k^2\varepsilon^2)^{1/2}}{2\pi}\right)(2\pi)\left(\tfrac{1}{2}(4m+2k\varepsilon^2)(4mk+k^2\varepsilon^2)^{-3/2}\right)$$

$$= \left(\frac{1}{\beta}\right)\frac{(4m+2k\varepsilon^2)(4mk+k^2\varepsilon^2)^{1/2}}{(4mk+k^2\varepsilon^2)^{3/2}}$$

$$= \left(\frac{1}{\beta}\right)\frac{4m+2k\varepsilon^2}{(4mk+k^2\varepsilon^2)}$$

$$= \frac{2(2m+k\varepsilon^2)}{\beta k(4m+k\varepsilon^2)}$$

& $\varepsilon = \frac{\beta}{N} = \frac{\beta}{2}$

$$\therefore \langle \overline{x^2} \rangle = \frac{2(2m+\frac{k\beta^2}{4})}{\beta k(4m+\frac{k\beta^2}{4})} = \frac{2(8m+k\beta^2)}{\beta k(16m+k\beta^2)} = \frac{16m+2k\beta^2}{16mk\beta + k^2\beta^3}$$

& $\beta = \frac{1}{T}$

$$\therefore \langle \overline{x^2} \rangle = \frac{(16m+\frac{2k}{T^2})}{(\frac{16mk}{T}+\frac{k^2}{T^3})} \times \frac{T^3}{T^3} = \frac{16mT^3+2kT}{16mkT^2+k^2} = \frac{2T(8mT^2+k)}{16mkT^2+k^2}$$

For our case: $m=k=1$

$$\therefore \langle \overline{x^2} \rangle = \frac{16T^3+2T}{16T^2+1}$$

N=3

$$S_3(x) = \frac{1}{2}(x_0, x_1, x_2)\begin{pmatrix} a & b & c \\ b & a & d \\ c & d & a \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$= \frac{1}{2}(x_0, x_1, x_2)\begin{pmatrix} ax_0 + bx_1 + cx_2 \\ bx_0 + ax_1 + dx_2 \\ cx_0 + dx_1 + ax_2 \end{pmatrix}$$

$$= \frac{1}{2}\left(ax_0^2 + bx_0x_1 + cx_0x_2 + bx_0x_1 + ax_1^2 + dx_1x_2 + cx_0x_2 + dx_1x_2 + ax_2^2\right)$$

$$= \frac{1}{2}\left(a(x_0^2+x_1^2+x_2^2) + 2bx_0x_1 + 2cx_0x_2 + 2dx_1x_2\right)$$

$$= \frac{1}{2}a(x_0^2+x_1^2+x_2^2) + bx_0x_1 + cx_0x_2 + dx_1x_2$$

$$S_3(x) = \varepsilon\sum_{j=0}^{2}\left[\frac{m}{2}\left(\frac{x_{j+1}-x_j}{\varepsilon}\right)^2 + \frac{1}{2}kx_j^2\right]$$

$$= \varepsilon\left[\frac{m}{2}\left(\frac{x_1-x_0}{\varepsilon}\right)^2 + \frac{1}{2}kx_0^2 + \frac{m}{2}\left(\frac{x_2-x_1}{\varepsilon}\right)^2 + \frac{1}{2}kx_1^2 + \frac{m}{2}\left(\frac{x_0-x_2}{\varepsilon}\right)^2 + \frac{1}{2}kx_2^2\right]$$

$$= \varepsilon\left[\frac{m}{2\varepsilon^2}\left((x_1-x_0)^2 + (x_2-x_1)^2 + (x_0-x_2)^2\right) + \frac{1}{2}k(x_0^2+x_1^2+x_2^2)\right]$$

$$= \frac{m}{2\varepsilon}\left(x_1^2 - 2x_0x_1 + x_0^2 + x_2^2 - 2x_1x_2 + x_1^2 + x_0^2 - 2x_0x_2 + x_2^2\right) + \frac{1}{2}k\varepsilon\left(x_0^2 + x_1^2 + x_2^2\right)$$

$$= x_0^2\left(\frac{m}{\varepsilon} + \frac{1}{2}k\varepsilon\right) + x_1^2\left(\frac{m}{\varepsilon} + \frac{1}{2}k\varepsilon\right) + x_2^2\left(\frac{m}{\varepsilon} + \frac{1}{2}k\varepsilon\right) + x_0x_1\left(-\frac{m}{\varepsilon}\right) + x_0x_2\left(-\frac{m}{\varepsilon}\right) + x_1x_2\left(-\frac{m}{\varepsilon}\right)$$

$$\Rightarrow a = \frac{2m}{\varepsilon} + k\varepsilon$$

& $b = c = d = -\frac{m}{\varepsilon}$

$$\therefore \underline{\underline{M}}_3 = \begin{pmatrix} \frac{2m}{\varepsilon}+k\varepsilon & -\frac{m}{\varepsilon} & -\frac{m}{\varepsilon} \\ -\frac{m}{\varepsilon} & \frac{2m}{\varepsilon}+k\varepsilon & -\frac{m}{\varepsilon} \\ -\frac{m}{\varepsilon} & -\frac{m}{\varepsilon} & \frac{2m}{\varepsilon}+k\varepsilon \end{pmatrix}$$

$$\therefore \underline{\underline{M}}_3 = \begin{pmatrix} \frac{2m}{\varepsilon}+k\varepsilon & \cdot & \cdot \\ -\frac{m}{\varepsilon} & \frac{2m}{\varepsilon}+k\varepsilon & -\frac{m}{\varepsilon} \\ -\frac{m}{\varepsilon} & -\frac{m}{\varepsilon} & \frac{2m}{\varepsilon}+k\varepsilon \end{pmatrix}$$

$$\det \underline{\underline{M}}_3 = \left(\frac{2m}{\varepsilon}+k\varepsilon\right)\left[\left(\frac{2m}{\varepsilon}+k\varepsilon\right)^2-\left(-\frac{m}{\varepsilon}\right)^2\right]-\left(-\frac{m}{\varepsilon}\right)\left[\left(-\frac{m}{\varepsilon}\right)\left(\frac{2m}{\varepsilon}+k\varepsilon\right)-\left(-\frac{m}{\varepsilon}\right)^2\right]+\left(-\frac{m}{\varepsilon}\right)\left[\left(-\frac{m}{\varepsilon}\right)^2-\left(\frac{2m}{\varepsilon}+k\varepsilon\right)\left(-\frac{m}{\varepsilon}\right)\right]$$

$$= \left(\frac{2m}{\varepsilon}+k\varepsilon\right)\left[\frac{4m^2}{\varepsilon^2}+4mk+k^2\varepsilon^2-\frac{m^2}{\varepsilon^2}\right]+\frac{m}{\varepsilon}\left[-\frac{2m^2}{\varepsilon^2}-km-\frac{m^2}{\varepsilon^2}\right]-\frac{m}{\varepsilon}\left[\frac{m^2}{\varepsilon^2}+\frac{2m^2}{\varepsilon^2}+km\right]$$

$$= \left(\frac{2m}{\varepsilon}+k\varepsilon\right)\left[\frac{3m^2}{\varepsilon^2}+4mk+k^2\varepsilon^2\right]+\frac{m}{\varepsilon}\left[-\frac{3m^2}{\varepsilon^2}-km\right]-\frac{m}{\varepsilon}\left[\frac{3m^2}{\varepsilon^2}+km\right]$$

$$= \left[\frac{6m^3}{\varepsilon^3}+\frac{8m^2k}{\varepsilon}+2mk^2\varepsilon+\frac{3km^2}{\varepsilon}+4mk^2\varepsilon+k^3\varepsilon^3\right]+\left[-\frac{3m^3}{\varepsilon^3}-\frac{km^2}{\varepsilon}\right]-\left[\frac{3m^3}{\varepsilon^3}+\frac{km^2}{\varepsilon}\right]$$

$$= \frac{9m^2k}{\varepsilon}+6mk^2\varepsilon+k^3\varepsilon^3$$

$$= \frac{9m^2k+6mk^2\varepsilon^2+k^3\varepsilon^4}{\varepsilon}$$

$$Z_3 = \sqrt{\frac{(2\pi\varepsilon)^3}{\det \underline{\underline{M}}_3}}$$

$$= \sqrt{\frac{8\pi^3\varepsilon}{9m^2k+6mk^2\varepsilon^2+k^3\varepsilon^4}}$$

$$\langle \overline{x^2} \rangle = -\frac{2}{\beta}\frac{d}{dk}\ln Z_3$$

$$= \frac{1}{\beta}\frac{(9m^2k+6mk^2\varepsilon^2+k^3\varepsilon^4)^{1/2}}{(8\pi^3\varepsilon)^{1/2}}(8\pi^3\varepsilon)^{1/2}\left(\frac{1}{2}\right)(9m^2+12mk\varepsilon^2+3k^2\varepsilon^4)(9m^2k+6mk^2\varepsilon^2+k^3\varepsilon^4)^{-3/2}$$

$$= \frac{1}{\beta}\frac{9m^2+12mk\varepsilon^2+3k^2\varepsilon^4}{9m^2k+6mk^2\varepsilon^2+k^3\varepsilon^4}$$

$$\& \ \varepsilon = \frac{\beta}{N} = \frac{\beta}{3}$$

$$\therefore \langle \overline{x^2} \rangle = \frac{1}{\beta}\frac{9m^2+12mk\left(\frac{\beta}{3}\right)^2+3k^2\left(\frac{\beta}{3}\right)^4}{9m^2k+6mk^2\left(\frac{\beta}{3}\right)^2+k^3\left(\frac{\beta}{3}\right)^4}$$

$$\& \ \beta = \frac{1}{T}$$

$$\therefore \langle \overline{x^2} \rangle = T\,\frac{9m^2+\frac{12}{9}mk\left(\frac{1}{T^2}\right)+\frac{3}{81}k^2\left(\frac{1}{T^4}\right)}{9m^2k+\frac{2}{3}4mk^2\left(\frac{1}{T^2}\right)+\frac{1}{81}k^3\left(\frac{1}{T^4}\right)} \times \frac{81T^4}{81T^4}$$

$$= \frac{729m^2T^5+108mkT^3+3k^2T}{729m^2kT^4+54mk^2T^2+k^3}$$

For our case $m=k=1$:

$$\therefore \langle \overline{x^2} \rangle = \frac{729T^5+108T^3+3T}{729T^4+54T^2+1}$$

Therefore,

For N=2: $< \overline{x^2} > = \frac{16T^3+2T}{16T^2+1}$

For N=3: $< \overline{x^2} > = \frac{729T^5+108T^3+3T}{729T^4+54T^2+1}$

Results:

Please check analysis section (possible hyperlink)

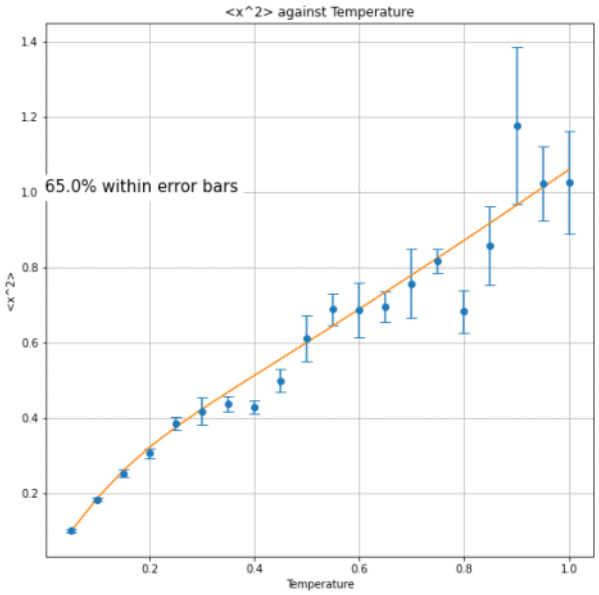
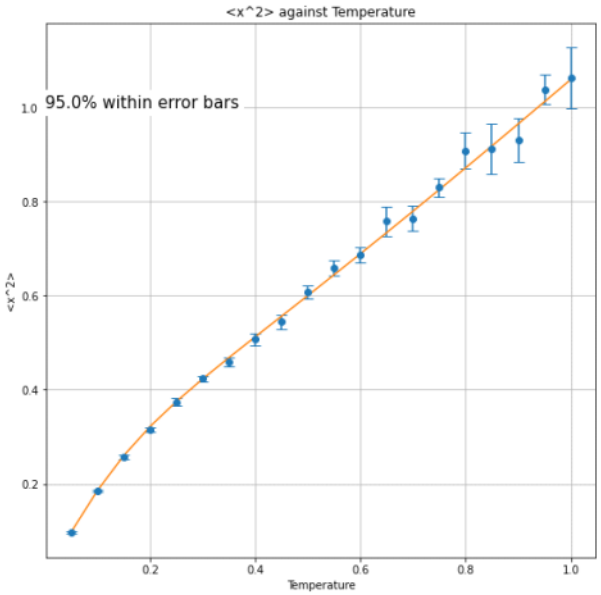
Analysis:

Conclusion:

Results

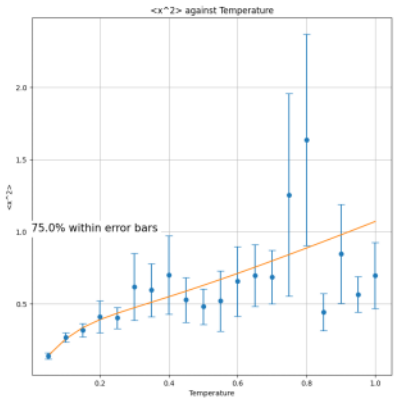
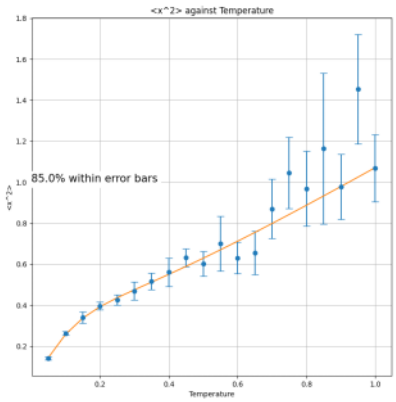
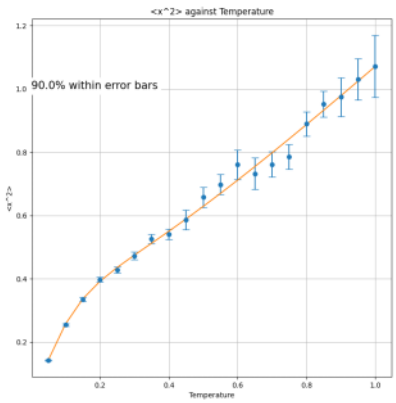
Sunday, October 30, 2022 12:54 PM

N=2:

Details	Graph	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=2 Iterations =9998		65%	Elapsed time = 10.125877857208252	
N=2 Iterations =99998		95%	Elapsed time = 354.942414522171	

N=3:

Details	Graph	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments

N=3 Iterations= 998		75%	Elapsed time = 2.2798657417297363	
N=3 Iterations= 9998		85%	Elapsed time = 7.176348924636841	
N=3 Iterations= 99998		90%	Elapsed time = 1717.7559640407562	

Code

Sunday, October 30, 2022 1:44 PM

```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #time how long code runs for

plt.close('all')

"""values"""
k = 1
beta = 1
m = 1
N = 3

"""functions"""
def action(x,beta,N,m,k):
    epsilon = beta/N
    S_sum = 0
    x = np.append(x,x[0])
    for j in range(0,N):
        S_sum += m/2*((x[j+1]-x[j])/epsilon)**2 + 0.5*k*x[j]**2
    S_N = epsilon * S_sum
    return S_N


def updateX(beta,N,m,k):
    """Pick arbitrary value for x"""
    x_values = np.zeros([N])
    x_values = np.append(x_values,x_values)
    x_values = np.reshape(x_values, (2,N))
    for i in range(99998):
        """duplicate array"""
        x_values = np.append(x_values,x_values[-1])
        x_values = np.reshape(x_values, (i+3,N))

        """Choose which x value to update"""
        point_selected = random.randint(0,N-1)
        x = x_values[-1, point_selected]

        """Update x and append"""
        delta = random.uniform(-1,1)
        x_updated = x+delta

        path = x_values[-1]
        path[point_selected] = x_updated
```

```
        """Test update"""
        if action(path,beta,N,m,k) <= action(x_values[-2],beta,N,m,k):
            pass
        else:
            prop = np.exp(action(x_values[-2],beta,N,m,k)-action(path,beta,N,m,k))
            if random.random() <= prop:
                pass
            else:
                x_values[-1, point_selected] = x
    #x_values = np.delete(x_values,0)
    return x_values

"""averages function"""
def meanError(x_values):
    x_split = np.split(x_values,10)
    x_split = np.delete(x_split,0,0)
    x_split = np.split(x_split,9)
    sum = 0
    x_split_mean = np.array([])
    for i in range(0,len(x_split)):
        sum += np.mean(x_split[i])
        x_split_mean = np.append(x_split_mean,np.mean(x_split[i]))
    mean = sum/len(x_split)
    std_err = np.std(x_split_mean)/(np.sqrt(len(x_split_mean))-1)
    return mean, std_err

"""plot figure"""
fig = plt.figure(figsize=(9,9))

"""plot <x^2> against Temperature"""
means = np.array([])
std_errs = np.array([])
temperature_values = np.array([i for i in range(1,21)])/20

start_time = time.time()
for i in range(0,20):
    meanErrors = meanError(updateX(1/temperature_values[i],N,m,k)**2)
    means = np.append(means,meanErrors[0])
    std_errs = np.append(std_errs,meanErrors[1])
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))

ax5 = fig.add_subplot(111)
ax5.grid()
ax5.errorbar(temperature_values,means, yerr=[std_errs,std_errs], capsize=5, fmt="o")
ax5.set_xlabel('Temperature')
ax5.set_ylabel('<x^2>')
ax5.title.set_text('<x^2> against Temperature')

"""plot expected curve"""
expected_curve2 = (16*temperature_values**3+2*temperature_values)/(16
*temperature_values**2+1)
expected_curve3 = (729*temperature_values**5+108*temperature_values**3+3
*temperature_values)/(729*temperature_values**4+54*temperature_values**2+1)
ax5.plot(temperature_values,expected_curve3)
```

```
inErrorBars = 0
for i in range(len(means)):
    if means[i]-std_errs[i]<=expected_curve3[i] and means[i]+std_errs[i]>=expected_curve3[i]:
        inErrorBars += 1
inErrorsPercentage = inErrorBars/len(means)*100
ax5.text(0,1,str(inErrorsPercentage)+'% within error bars',fontsize=15,backgroundcolor='w')
```

Week 7

Supervisor Meeting: 02/11/2022

Thursday, November 3, 2022 10:32 AM

Outline:

- Checked that units were correct for previously calculated results
- Looked at results for m=0
- Discussed the Hamiltonian and using the annihilation and creation operators to get to the ground state energy of a system
- Using high N, to find the <x^2> for the ground state of the quantum harmonic oscillator

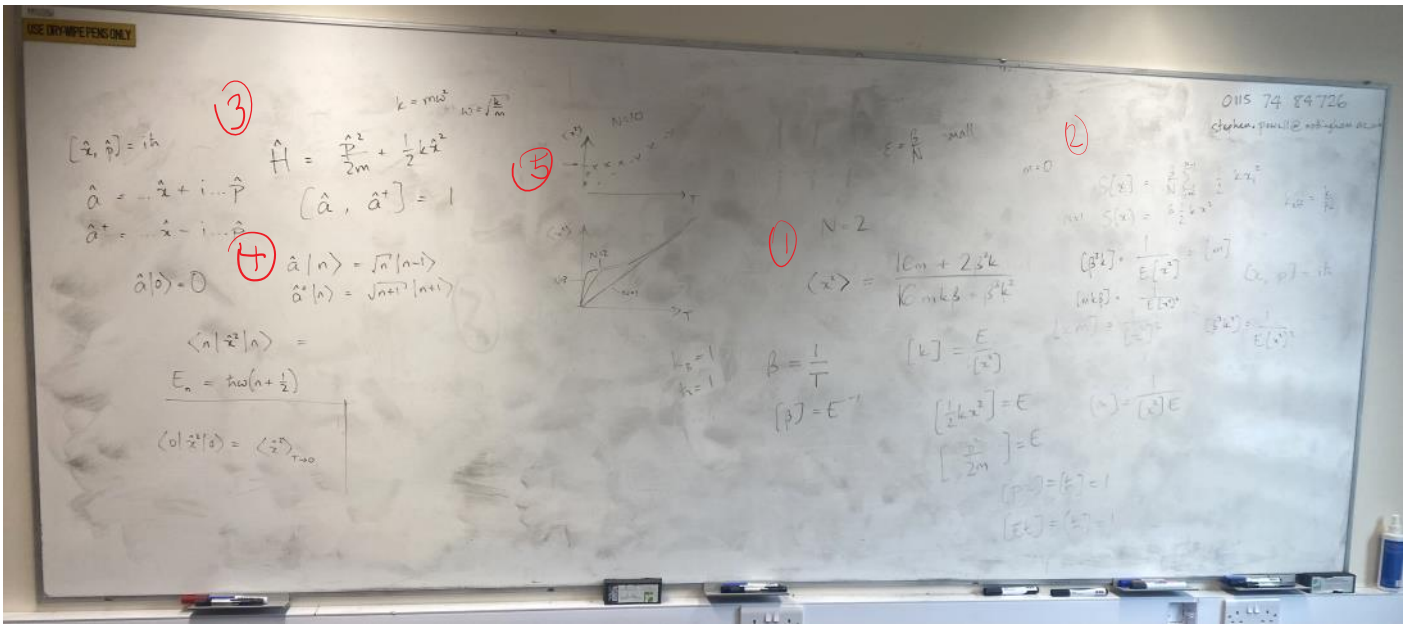
Tasks:

- Check that last week's code gives the correct results with the correct units as shown in meeting
- Repeat our code for large values of N (e.g: N=10,20 and no larger than N=100)
- Check that these values agree with our analytical results

Distribution of Tasks:

- Both would produce graphs and data using our codes to confirm everything was working.
- We would then compare our results to see if any issues arose - meeting before our next supervisor meeting to see if we can find any errors.

Project Details:



1) Here we checked to see if the units matched on both sides of the given equation, to ensure that our previously calculated results were correct.
2) Here outlines the case where m=0, and the equations for the actions. By comparing this to the case where N=1, we can say that the effective k=k/N
3) Here outlines the definitions of the Hamiltonian for the case of the QHO, and the definitions of the annihilation and the creation operators.
4) This shows what happens when the operators act on a certain eigenstate of the wavefunction.
5) These graphs show how the results for <x^2> are expected to change with respect to temperature.
    a. For large T, the graphs all tend to the same values for all N
    b. For all N, <x^2> tend to zero
        i. This is an error due to the fact that we made an initial assumption that epsilon=beta/N was small. However, for large beta (small T), this assumption no longer holds. The corrected version can be seen in the graph above.

Textbooks about the Quantum Harmonic Oscillator:

1) D.J.Griffiths (1995). Introduction to Quantum Mechanics, Prentice-Hall International (UK) Limited. Chapter 2.6
2) F.Schwabl (1992). Quantum Mechanics, Springer-Verlag. Chapter 3.1

(TW) Checking Results for N=2 and N=3

02 November 2022

Checking Results for N=2,3: (02/11/2022)

In our discussions in our supervisor meeting we discussed our results from last week and checked the dimensions of the theoretical result for N=2. The dimensions agreed with that value I found, and made some physical sense.

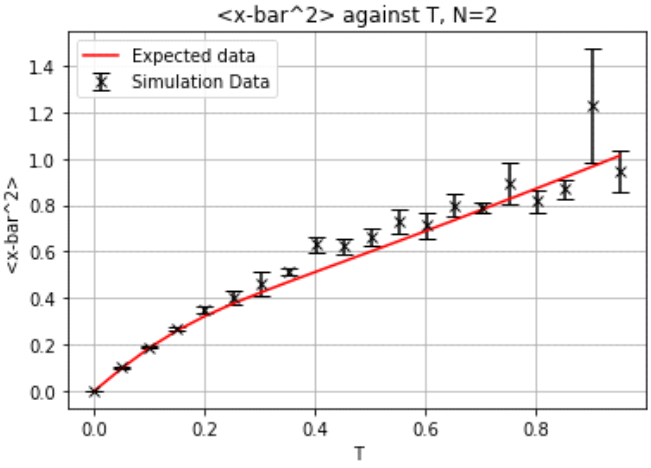
N=2:

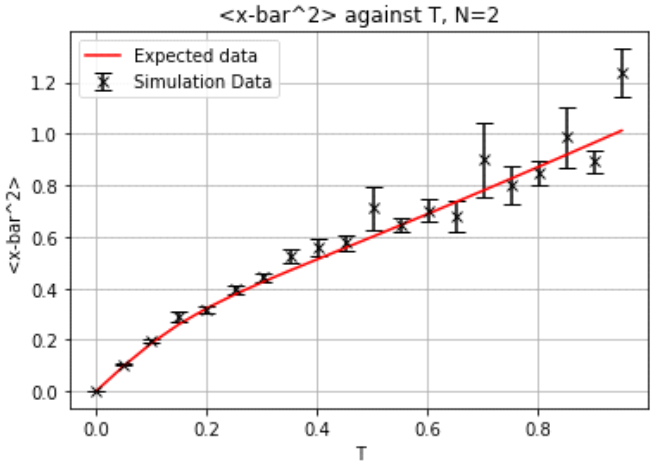
- The dimensions of $\langle x^2 \rangle = \frac{16m + 2\beta^2 k}{16mk\beta + \beta^3 k^2}$ are L^2 as expected. This was shown in our meeting.
- The case of m=0 will give back an uncoupled system - so the value of $\langle x^2 \rangle$ should be similar to the case of N=1 - with a constant due to N:

for $m=0$ $\langle x^2 \rangle = \frac{2\beta^2 k}{\beta^3 k^2} = \frac{2}{k\beta}$

for $N=1$: $\langle x^2 \rangle = \frac{1}{k\beta}$. This factor of 2 comes from the action of $S_2 = \frac{1}{2}kx_0^2 + \frac{1}{2}kx_1^2$

This gives another implication that the value of $\langle x^2 \rangle$ for N=2 is correct (that being my algebra was correct). I also decided to plot this case again, with a closer look for low T, as this is where the more interesting behaviour occurs (not roughly a y=x line):
Again the simulation data agreed with the expected plot. But here we see the more interesting behaviour for small T, which is not a simple linear dependence.

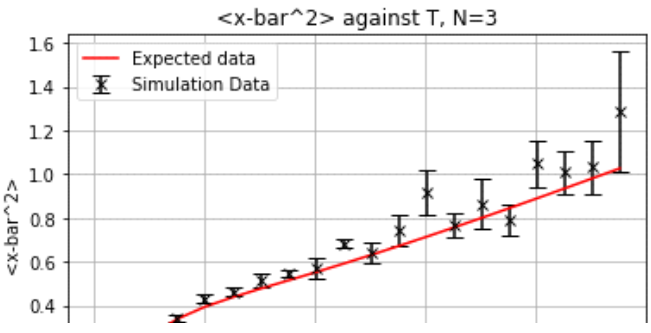


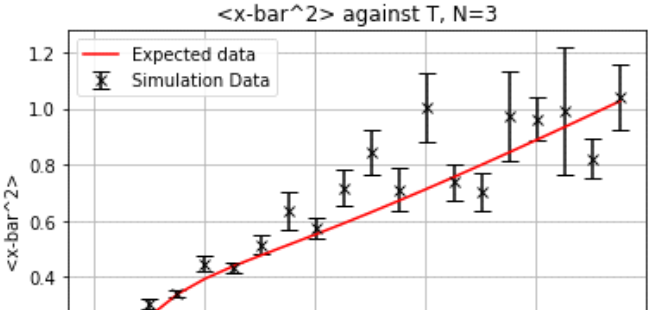


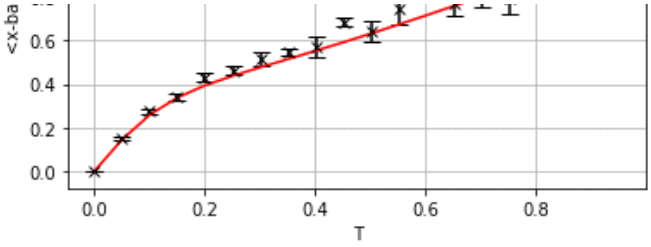
N=3:

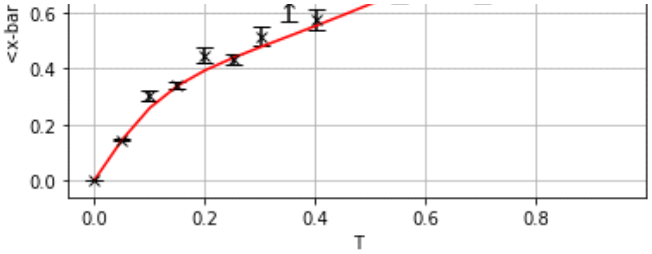
- The dimensions of $\langle x^2 \rangle = \frac{\frac{27m^2}{\beta} + 9m\beta k + \frac{\beta^3 k^2}{9}}{27m^2 k + 2m\beta^2 k^2 + \frac{\beta^4 k^3}{27}}$ are L^2 as expected.
- The case where m=0 gives $\langle x^2 \rangle = \frac{3}{\beta K}$ as expected.

This again gives me confidence that my algebra was correct for the N=3 case. Like above I plotted the graph again but for a lower temperature range to see the more interesting behaviour:









Conclusion:

Overall I am happy with my results from last week, and my algebra seems to be correct after a few sanity checks. The graphs above a more interesting than the ones produced last time as show the behaviour for smaller T. For large T, both graphs tend to the N=1 case, which isn't as interesting.

(TW) Statistical Results for Large N - Part 1 (02/11/2022)

02 November 2022

Statistical Results for Large N: (02/11/2022)

Now that we are happy with the code, we shall set N to be large enough to simulate the Quantum Harmonic Oscillator (QHO). In the QMC method, a larger N leads to more accurate results. This week I shall set N=10,20 and possibly higher to try to simulate the QHO. This will be compared to known results for the QHO - this week we will look at $\langle \hat{x}^2 \rangle$ for the ground-state in a QHO.

This result should agree with our simulation for low T and large N.

As before I need to find the expected value of $\langle \hat{x}^2 \rangle$ before running the simulation:

Expectation value for $\hat{x}^2$ in the QHO:

$\hat{a} = \alpha \hat{x} + i\beta \hat{p}$ $\quad \alpha = \sqrt{\frac{m\omega}{2\hbar}}$ $\quad \omega = \sqrt{\frac{k}{m}}$

$\hat{a}^\dagger = \alpha \hat{x} - i\beta \hat{p}$ $\quad \beta = \frac{1}{\sqrt{2mk\omega}}$

$\hat{a}^\dagger + \hat{a} = 2\alpha \hat{x} \Rightarrow \hat{x} = \frac{1}{2\alpha}(\hat{a} + \hat{a}^\dagger)$

$\Rightarrow \hat{x}^2 = \frac{1}{4\alpha^2}(\hat{a}\hat{a} + \hat{a}\hat{a}^\dagger + \hat{a}^\dagger\hat{a} + \hat{a}^\dagger\hat{a}^\dagger)$

Want to find $\langle n|\hat{x}^2|n\rangle$.

NB: $\hat{a}|n\rangle = \sqrt{n}\,|n-1\rangle$ and $\langle n|m\rangle = \delta_{nm}$

$\hat{a}^\dagger|n\rangle = \sqrt{n+1}\,|n+1\rangle$

$\langle n|\hat{x}^2|n\rangle = \frac{1}{4\alpha^2}(\langle n|\hat{a}\hat{a}|n\rangle + \langle n|\hat{a}\hat{a}^\dagger|n\rangle + \langle n|\hat{a}^\dagger\hat{a}|n\rangle + \langle n|\hat{a}^\dagger\hat{a}^\dagger|n\rangle)$

$= \frac{1}{4\alpha^2}(\sqrt{n}\langle n|\hat{a}|n-1\rangle + \sqrt{n+1}\langle n|\hat{a}|n+1\rangle + \sqrt{n}\langle n|\hat{a}^\dagger|n-1\rangle + \sqrt{n+1}\langle n|\hat{a}^\dagger|n+1\rangle)$

$= \frac{1}{4\alpha^2}(\sqrt{n}\sqrt{n-1}\langle n|n-2\rangle + \sqrt{n+1}\sqrt{n+1}\langle n|n\rangle + \sqrt{n}\sqrt{n}\langle n|n\rangle + \sqrt{n+1}\sqrt{n+2}\langle n|n+2\rangle)$

$= \frac{1}{4\alpha^2}(n+1+n) = \frac{2n+1}{4\alpha^2} = \langle \hat{x}^2 \rangle_n$

$\langle \hat{x}^2 \rangle_n = \frac{2n+1}{4} \cdot \left(\frac{2\hbar}{m\omega}\right) = \frac{\hbar}{m\omega}(n+\tfrac{1}{2})$.

for ease $\hbar=1, m=1, k=1 \Rightarrow \omega = \sqrt{\frac{k}{m}} = 1$

for n=0 (ground state):

$\langle \hat{x}^2 \rangle_0 = \langle 0|\hat{x}^2|0\rangle = \tfrac{1}{2}$.

So we see that in the ground-state (that is n=0) we have $\langle \hat{x}^2 \rangle = \frac{1}{2}$. This requires T to be small, and uses are values of $\hbar = m = k = 1$.

Simulation vs. Expected value for $\langle \hat{x}^2 \rangle$ for the QHO:

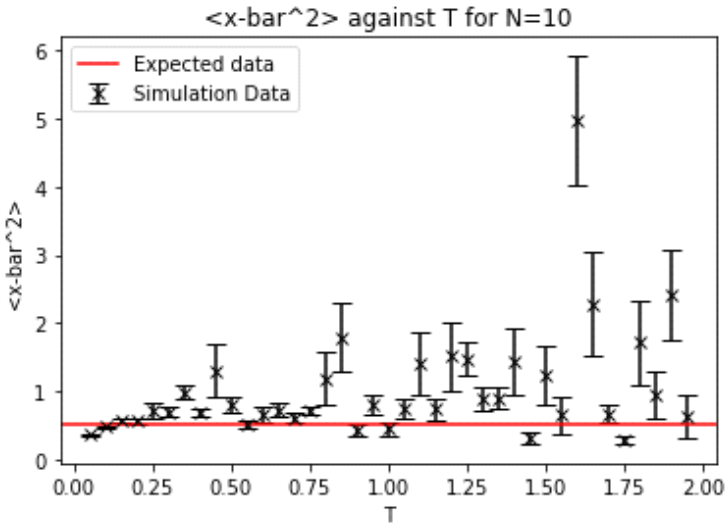
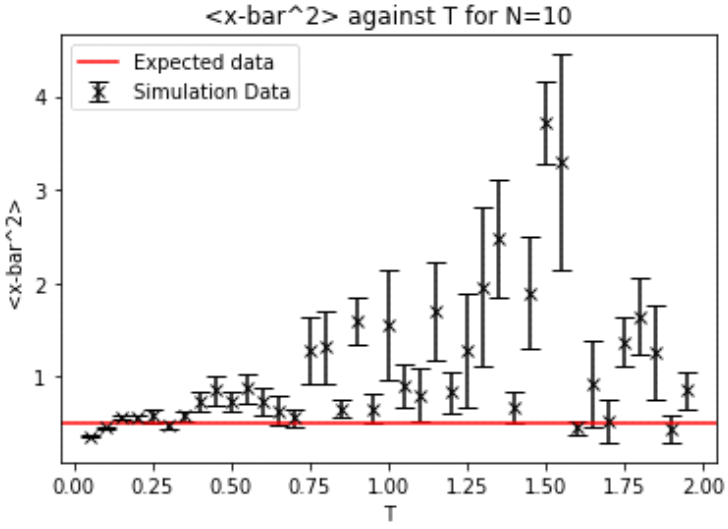
We use the same code as last week, just with setting N=10 and larger. This will produce a plot, and for low T we

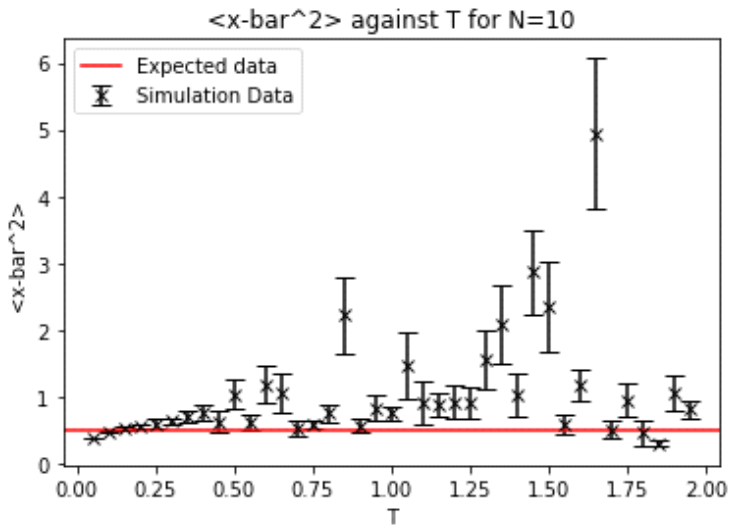
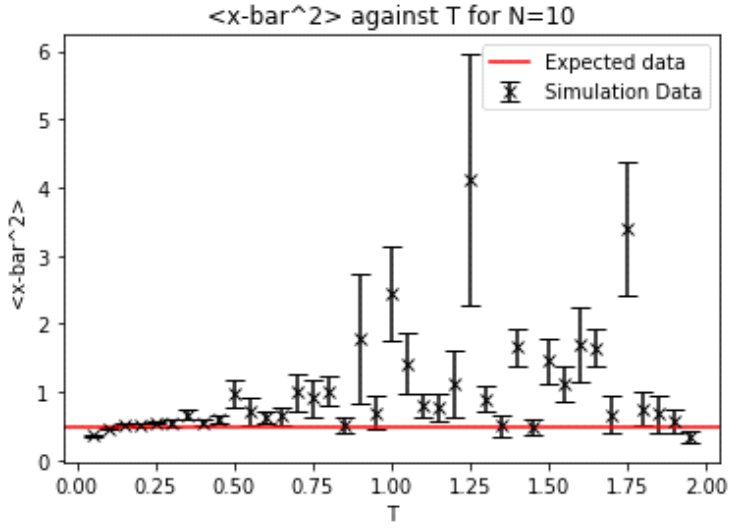
expect our values to be within an error bar of 0.5.

Note: The following variable values were used, with only N changed where stated:

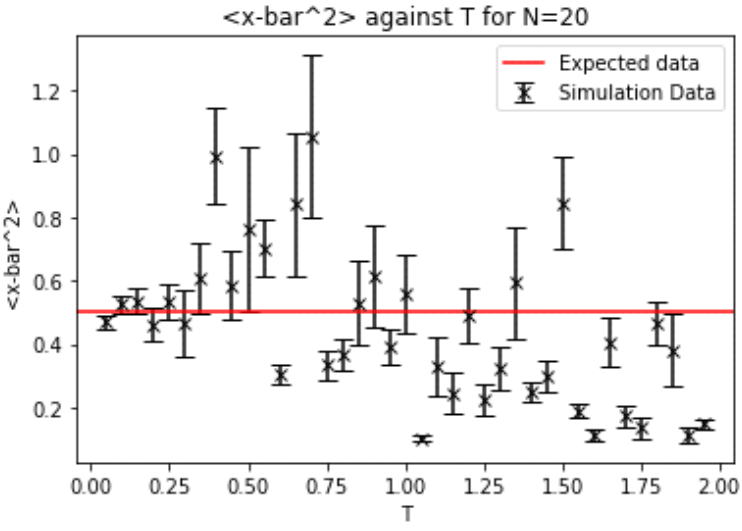
```
N = 20
m = 1
k = 1
it = 50000
T = np.arange(0.05,2,0.05)
beta = 1/T
```

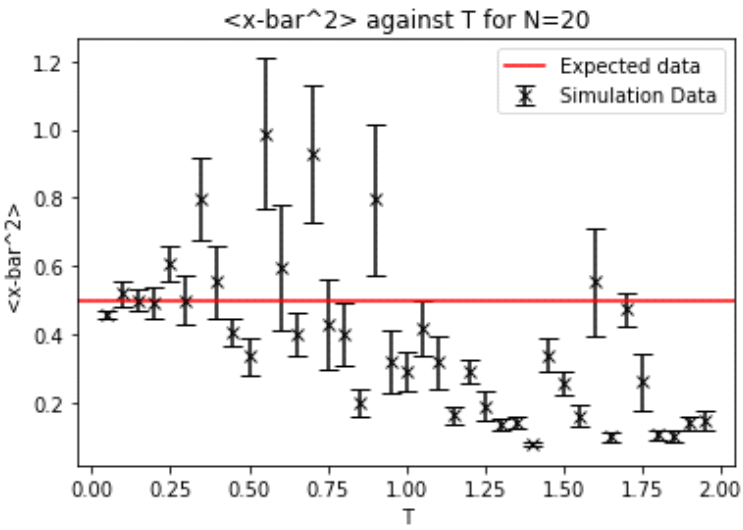
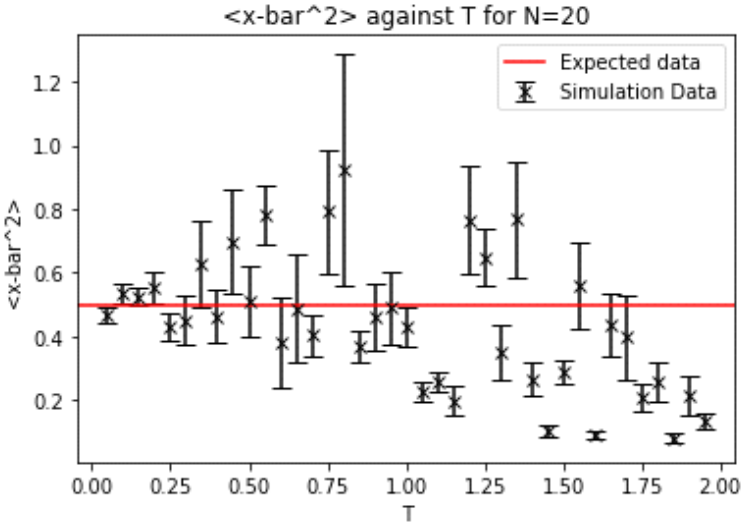
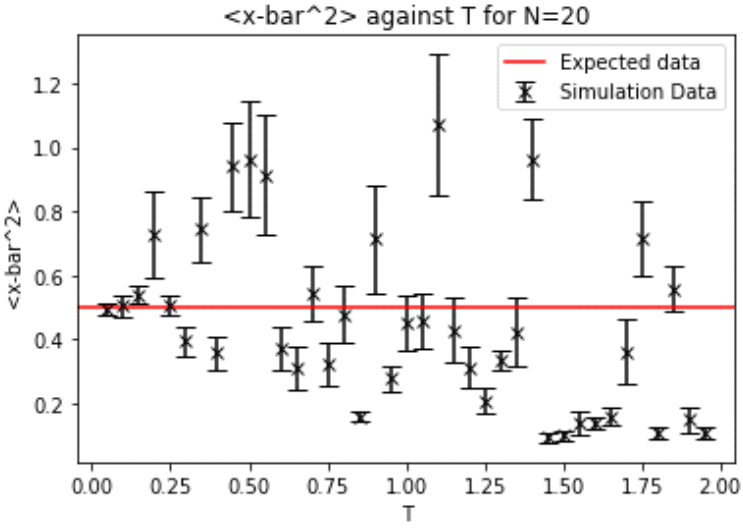
Results for N=10:

Run	Plot
1	
2	

3	
4	

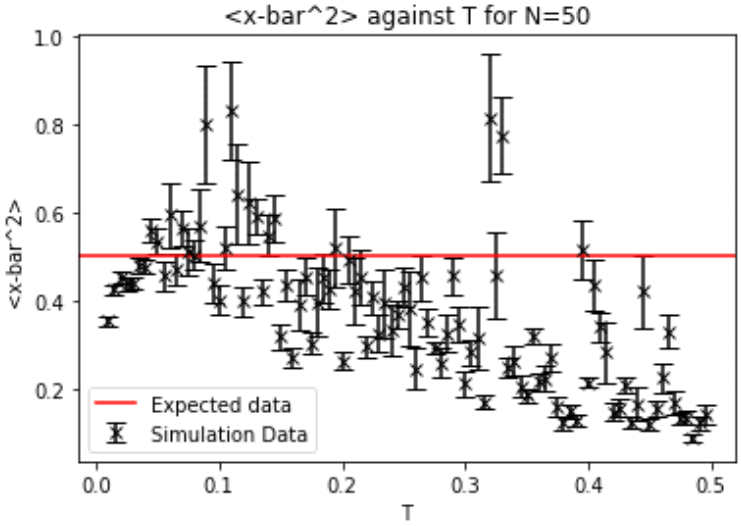
Results for N=20:

Run	Plot
1	

2	
3	
4	

The results for both cases vaguely follow what is expected. As this is a more rough estimate - only accounting for the ground-state, thus not representing the entire system - I do not want to make too many remarks. However, I am mostly happy as the simulation results are within an error bar of this value for very low temperatures.

I also tried **N=50, but over a smaller T range (0.01 to 0.5):**



Similar comments to above - it roughly agrees with this very rough estimate, but will have to do more analysis to be sure.

Conclusion:

Given how our expected value is a very rough estimate, I am quite happy with my results for low T and large N. Next week we will look at the thermal expectation value for the QHO which will give a better expected result to follow - and hopefully will provide more insight into whether or not the simulation is working correctly.

TK

Saturday, November 5, 2022 5:05 PM

Please expand tab for full weekly report

Theory

Saturday, November 5, 2022 1:10 PM

Outline:

- This week I want to find out what happens for large N, and ensure that the code that I have produced works correctly for this
- Check for where and why the simulation breaks down
- Investigate how the simulation changes for increasing N

Background:

$< 0|\hat{x}^2|0 > = < \hat{x}^2 >_{T \to 0}$, this means that the ground state motion will be what the simulation tends to as T tends to 0. However, this is affected by limitations in the model, because it is assumed that $\varepsilon = \frac{\beta}{N}$ is small. However, for very small temperatures epsilon, is no longer small. So the assumption no longer holds, and the simulation breaks down.

Plan:

- Calculate $< \hat{x}^2 >_0$
- Run code for large N=10 & N=20
  - Analyse plots
- Create and run new code, investigating where the assumption $\varepsilon = \frac{\beta}{N}$ is small holds
  - Guess range of temperature values, where assumption no longer holds (as seen from previous plots)
  - Run simulation for these temperature values with a step of 0.01

Method:

$\langle 0|\hat{x}^2|0\rangle = \langle \hat{x}^2 \rangle_{T\to 0}$

$\hat{x} = \sqrt{\frac{\hbar}{2m\omega}}\left(\hat{a}^\dagger + \hat{a}\right)$

$\hat{x}^2 = \frac{\hbar}{2m\omega}\left(\hat{a}^\dagger + \hat{a}\right)\left(\hat{a}^\dagger + \hat{a}\right)$

$\Rightarrow \langle n|\hat{x}^2|n\rangle = \langle n|\frac{\hbar}{2m\omega}\left(\hat{a}^\dagger + \hat{a}\right)\left(\hat{a}^\dagger + \hat{a}\right)|n\rangle$

$= \frac{\hbar}{2m\omega}\langle n|\hat{a}^\dagger\hat{a}^\dagger + \hat{a}^\dagger\hat{a} + \hat{a}\hat{a}^\dagger + \hat{a}\hat{a}|n\rangle$

$= \frac{\hbar}{2m\omega}\left[\langle n|\hat{a}^\dagger\hat{a}^\dagger|n\rangle + \langle n|\hat{a}^\dagger\hat{a}|n\rangle + \langle n|\hat{a}\hat{a}^\dagger|n\rangle + \langle n|\hat{a}\hat{a}|n\rangle\right]$

$= \frac{\hbar}{2m\omega}\left[\sqrt{n+1}\sqrt{n+2}\langle n|n+2\rangle + \sqrt{n}\sqrt{n}\langle n|n\rangle + \sqrt{n+1}\sqrt{n+1}\langle n|n\rangle + \sqrt{n}\sqrt{n-1}\langle n|n-2\rangle\right]$

& $\langle n_i|n_j\rangle = \delta_{ij}$

$\langle n|\hat{x}^2|n\rangle = \frac{\hbar}{2m\omega}\left[n + n + 1\right]$

$= \frac{\hbar}{2m\omega}\left(2n+1\right)$

$= \frac{\hbar}{m\omega}\left(n + \frac{1}{2}\right)$

& $\omega = \sqrt{\frac{k}{m}}$

$\therefore \langle n|\hat{x}^2|n\rangle = \frac{\hbar}{m\sqrt{\frac{m}{k}}}\left(n + \frac{1}{2}\right)$

$= \hbar\sqrt{\frac{1}{km}}\left(n + \frac{1}{2}\right)$

For our case: $\hbar = k = m = 1$

$\therefore \langle n|\hat{x}^2|n\rangle = \left(n + \frac{1}{2}\right)$

$\therefore \langle 0|\hat{x}^2|0\rangle = \langle \hat{x}^2 \rangle_{T\to 0} = \frac{1}{2}$

Therefore, while the initial assumption of the simulation holds, the simulation should tend to 0 as temperature tends to 0.

Results:

Please check analysis section (possible hyperlink)

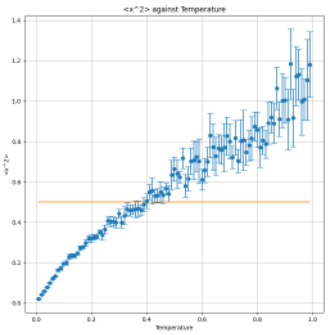
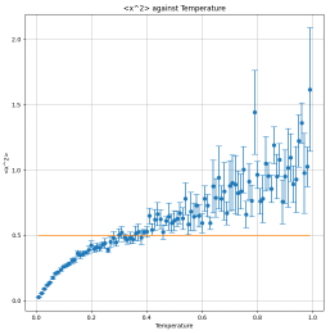
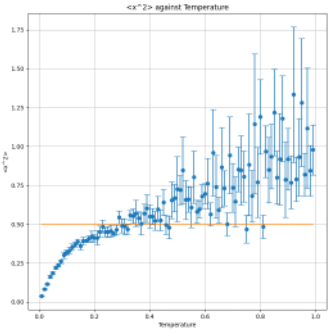
Analysis:

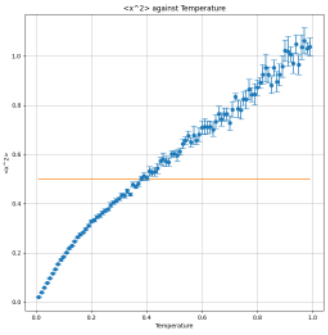
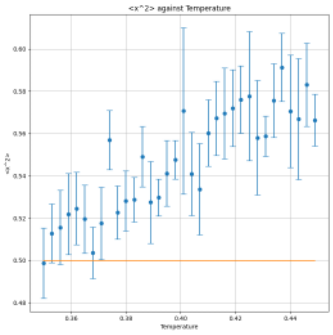
Conclusion:

Results

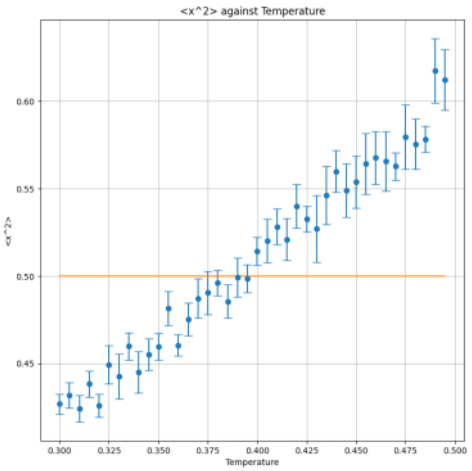
Sunday, November 6, 2022 7:05 PM

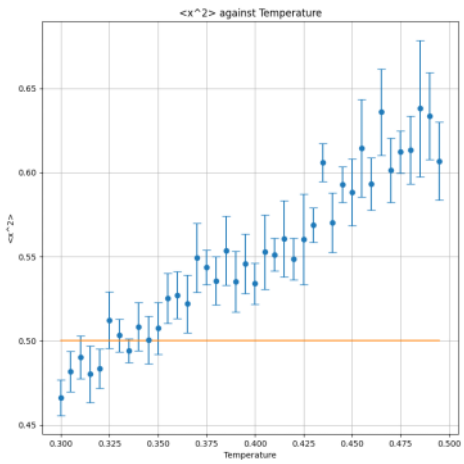
N = number of points on lattice
It = number of iterations
T = temperature values

Details	Graph	% in error bars	Average T in error bars for 0.5	Time Taken	Comments
N=2 It=9998 T=(0.1,1,0.1)			0.41333333333333333 average temperature in error bars for 0.5	Elapsed time = 90.08760285377502	
N=3 It=9998 T=(0.1,1,0.1)			0.35 average temperature in error bars for 0.5	Elapsed time = 125.33560371398926	
N=4 It=9998 T=(0.1,1,0.1)			0.48000000000000004 average temperature in error bars for 0.5 [0.23 0.3 0.31 0.35 0.36 0.37 0.4 0.41 0.42 0.44 0.46 0.47 0.55 0.62 0.68 0.71 0.75 0.81]	Elapsed time = 136.36992311477766	

N=2 It=99998 T=(0.01,1,0.01)			0.38999999999999996 average temperature in error bars for 0.5 [0.38 0.39 0.4]	Elapsed time = 4531.236029863375	
N=3 It=99998 T=(0.35,0.45,0.001)			0.35675 average temperature in error bars for 0.5 [0.35 0.353 0.356 0.368]	Elapsed time = 2601.826593875885	

Attempt 2:

Details	Graph	Average T in error bars for 0.5	Comments
N=2 It=99998 T=(0.3,0.5,0.005)		0.38500000000000006 average temperature in error bars for 0.5 [0.375 0.38 0.39 0.395]	

N=3 It=99998 T=(0.3,0.5, 0.005)		0.3335714285714286 average temperature in error bars for 0.5 [0.31 0.325 0.33 0.335 0.34 0.345 0.35]	Need to lower both the maximum and minimum temperatures by equal amounts for increasing N.

Need to try for much larger N, will attempt this in the coming weeks, once I show that the results are reliable for larger N.

At the moment it appears that the temperatures for which the ground state energy (0.5) is predicted is:

- For N=2:
  - T=0.38
- For N=3:
  - T=0.33

Code

Sunday, November 6, 2022 1:17 PM

```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #time how long code runs for

plt.close('all')

"""values"""
k = 1
beta = 1
m = 1
N = 3

"""functions"""
def action(x,beta,N,m,k):
    epsilon = beta/N
    S_sum = 0
    x = np.append(x,x[0])
    for j in range(0,N):
        S_sum += m/2*((x[j+1]-x[j])/epsilon)**2 + 0.5*k*x[j]**2
    S_N = epsilon * S_sum
    return S_N


def updateX(beta,N,m,k):
    """Pick arbitrary value for x"""
    x_values = np.zeros([N])
    x_values = np.append(x_values,x_values)
    x_values = np.reshape(x_values, (2,N))
    for i in range(99998): # change number of iterations
        """duplicate array"""
        x_values = np.append(x_values,x_values[-1])
        x_values = np.reshape(x_values, (i+3,N))

        """Choose which x value to update"""
        point_selected = random.randint(0,N-1)
        x = x_values[-1, point_selected]

        """Update x and append"""
        delta = random.uniform(-1,1)
        x_updated = x+delta

        path = x_values[-1]
        path[point_selected] = x_updated
```

```
        """Test update"""
        if action(path,beta,N,m,k) <= action(x_values[-2],beta,N,m,k):
            pass
        else:
            prop = np.exp(action(x_values[-2],beta,N,m,k)-action(path,beta,N,m,k))
            if random.random() <= prop:
                pass
            else:
                x_values[-1, point_selected] = x
    #x_values = np.delete(x_values,0)
    return x_values

"""averages function"""
def meanError(x_values):
    x_split = np.split(x_values,10)
    x_split = np.delete(x_split,0,0)
    x_split = np.split(x_split,9)
    sum = 0
    x_split_mean = np.array([])
    for i in range(0,len(x_split)):
        sum += np.mean(x_split[i])
        x_split_mean = np.append(x_split_mean,np.mean(x_split[i]))
    mean = sum/len(x_split)
    std_err = np.std(x_split_mean)/(np.sqrt(len(x_split_mean))-1)
    return mean, std_err

"""plot figure"""
fig = plt.figure(figsize=(9,9))

"""plot <x^2> against Temperature"""
means = np.array([])
std_errs = np.array([])
temperature_values = np.arange(0.2,0.6,0.01) # change temperature values being used

start_time = time.time()
for i in range(0,len(temperature_values)):
    meanErrors = meanError(updateX(1/temperature_values[i],N,m,k)**2)
    means = np.append(means,meanErrors[0])
    std_errs = np.append(std_errs,meanErrors[1])
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))

ax5 = fig.add_subplot(111)
ax5.grid()
ax5.errorbar(temperature_values,means, yerr=[std_errs,std_errs], capsize=5, fmt="o")
ax5.set_xlabel('Temperature')
ax5.set_ylabel('<x^2>')
ax5.title.set_text('<x^2> against Temperature')

"""plot expected curve"""
expected = np.ones(len(temperature_values))/2
ax5.plot(temperature_values,expected)

inErrorBars = 0
valsInErrorBars = np.array([])
for i in range(len(means)):
```

```
    if means[i]-std_errs[i]<=expected[i] and means[i]+std_errs[i]>=expected[i]:
        inErrorBars += 1
        valsInErrorBars = np.append(valsInErrorBars,temperature_values[i])
inErrorsPercentage = inErrorBars/len(means)*100
avInErrorBars = np.mean(valsInErrorBars)
#ax5.text(0,1,str(avInErrorBars)+'average temp in error bars for 0.5',fontsize=
15,backgroundcolor='w')
print(str(avInErrorBars)+' average temperature in error bars for 0.5')
print(valsInErrorBars)
```

Week 8

Supervisor Meeting: 09/11/2022

09 November 2022

Outline:

- Briefly spoke about our results from last week
- Looked at the thermal expectation value and how to calculate it.

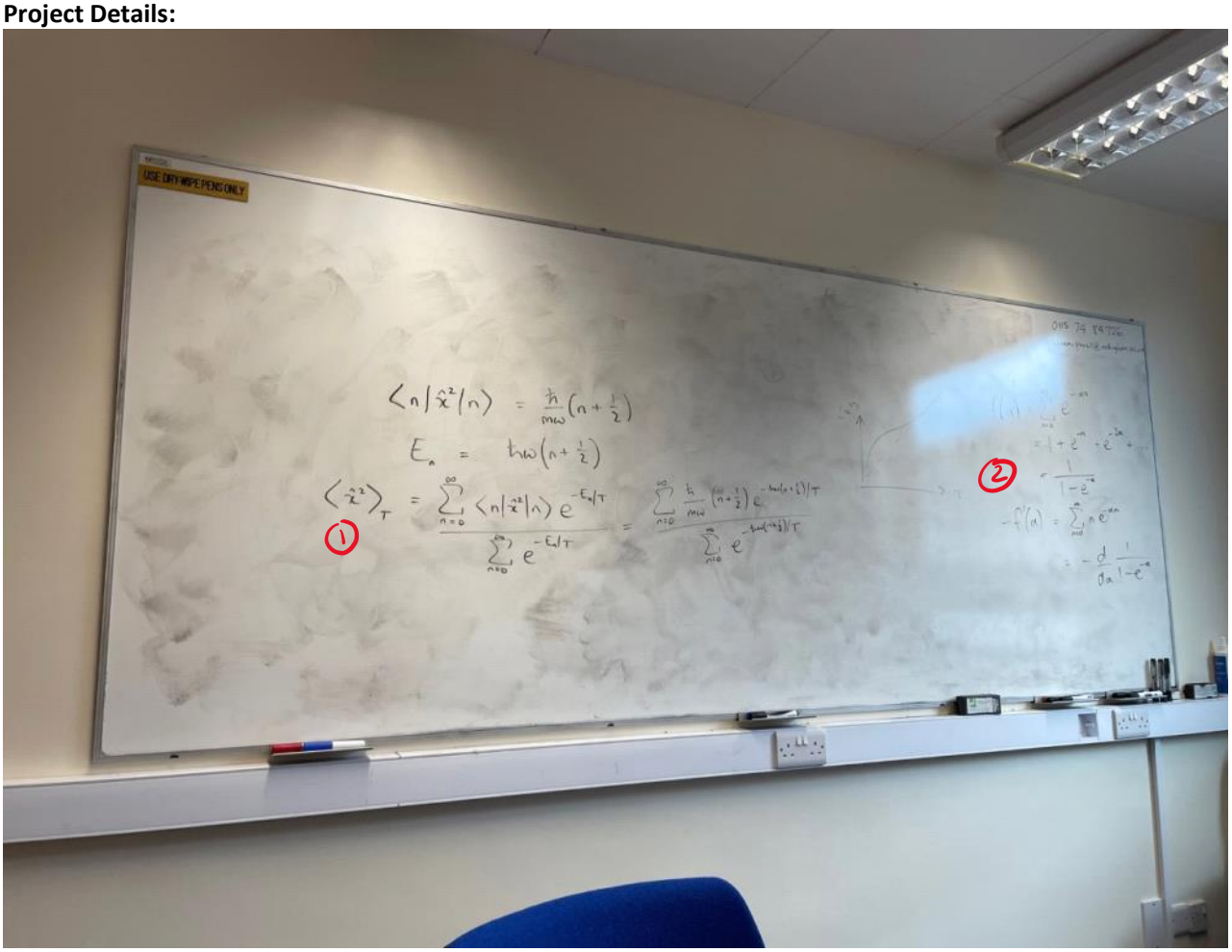
Tasks:

- Calculate the thermal expectation value and plot it for N<50 - see if it agrees with our simulation.

Distribution of Tasks:

- Both would produce graphs and data using our codes to confirm everything was working.
- We would then compare our results to see if any issues arose.

Project Details:



1) This is the calculation that is needed to find the thermal expectation value. It uses values from last week to find a weighted average of the square of the position of a particle by using a weighting from the energy of a given state.
2) Standard results to help us calculate the sums.

A few comments:

To get data close to the expected value we will have to run the simulation for many iterations. This will take time - so ensure everything is working before setting the number of iterations to be large.

The simulation will not agree with the expected value for very small T - due to approximations made in the path integral method. We will try and see this in our plots.

If the simulation follows the expected result, we will then move on to potentially looking at how to make our code more efficient and other potentials that do not have an exact analytical solution.

(TW) Statistical Results for Large N - Part 2 (09-16/11/2022)

09 November 2022

Statistical Results for Large N: (09/11/2022)

Following on from last week, we will set our cod to have larger values of N and plot that against the expected data. The expected will be for $\langle \hat{x}^2 \rangle_T$ that is the expectation vaalue of x^2 for different temperatures - not the simple case from last week.

This result should agree with our simulation for low T and large N. But will break for very small T depending on the value of N.

$\langle \hat{x}^2 \rangle_T$ **Calculation:**

$$\langle \hat{x}^2 \rangle_T = \frac{\sum_{n=0}^{\infty} \langle n|\hat{x}^2|n\rangle e^{-E_n/T}}{\sum_{n=0}^{\infty} e^{-E_n/T}}$$

$k_0 = 1$ with $\langle n|\hat{x}^2|n\rangle = \frac{\hbar}{m\omega}(n+\frac{1}{2})$

and $E_n = \hbar\omega(n+\frac{1}{2})$

$$\text{so } \langle \hat{x}^2 \rangle_T = \frac{\sum_{n=0}^{\infty} \frac{\hbar}{m\omega}(n+\frac{1}{2}) e^{-\hbar\omega(n+\frac{1}{2})/T}}{\sum_{n=0}^{\infty} e^{-\hbar\omega(n+\frac{1}{2})/T}}$$

So need to evaluate: $\frac{\hbar}{m\omega}\sum_{n=0}^{\infty} n e^{-\hbar\omega(n+\frac{1}{2})/T} = \frac{\hbar}{m\omega} e^{\frac{-\hbar\omega}{2T}} \sum_{n=0}^{\infty} n e^{-\frac{\hbar\omega n}{T}}$

$$\frac{\hbar}{2m\omega}\sum_{n=0}^{\infty} e^{-\hbar\omega(n+\frac{1}{2})/T} = \frac{\hbar}{2m\omega} e^{-\frac{\hbar\omega}{2T}} \sum_{n=0}^{\infty} e^{-\frac{\hbar\omega n}{T}}$$

and $e^{-\frac{\hbar\omega}{2T}} \sum_{n=0}^{\infty} e^{-\frac{\hbar\omega n}{T}}$

1) $f(\alpha) = \sum_{n=0}^{\infty} e^{-\alpha n} = \frac{1}{1-e^{-\alpha}}$

We have $\alpha = \frac{\hbar\omega}{T} \Rightarrow f(\frac{\hbar\omega}{T}) = \frac{1}{1-\exp(-\frac{\hbar\omega}{T})}$ //

2) Note $-f'(\alpha) = \sum_{n=0}^{\infty} n e^{-\alpha n} = -\frac{d}{d\alpha}\left((1-e^{-\alpha})^{-1}\right)$

$$= -\left((-1)\cdot(e^{-\alpha})(1-e^{-\alpha})^{-2}\right)$$

$$= \frac{e^{-\alpha}}{(1-e^{-\alpha})^2}$$

So above we have $\alpha = \frac{\hbar\omega}{T}$

$$\Rightarrow f(\tfrac{\hbar\omega}{T}) = \frac{\exp(-\frac{\hbar\omega}{T})}{(1-\exp(-\frac{\hbar\omega}{T}))^2}$$ //

$$\text{So } \langle \hat{x}^2 \rangle_T = \frac{\frac{\hbar}{m\omega} e^{-\frac{\hbar\omega}{2T}}\left(\frac{\exp(-\frac{\hbar\omega}{T})}{(1-\exp(-\frac{\hbar\omega}{T}))^2} + \frac{1/2}{1-\exp(-\frac{\hbar\omega}{T})}\right)}{e^{-\frac{\hbar\omega}{2T}}\left(\frac{1}{1-\exp(-\frac{\hbar\omega}{T})}\right)}$$

$$= \frac{\frac{\hbar}{m\omega}\left[\frac{\exp(\frac{\hbar\omega}{T})}{(\exp(\frac{\hbar\omega}{T})-1)^2} + \frac{\exp(\frac{\hbar\omega}{T})}{2(\exp(\frac{\hbar\omega}{T})-1)}\right]}{\frac{\exp(\frac{\hbar\omega}{T})}{(\exp(\frac{\hbar\omega}{T})-1)}}$$

$$= \frac{\hbar}{m\omega}\left[\frac{\exp(\frac{\hbar\omega}{T})}{\exp(\frac{\hbar\omega}{T})-1} + \frac{\exp(\frac{\hbar\omega}{T})}{2}\right]\cdot\frac{1}{\exp(\frac{\hbar\omega}{T})}$$

$$= \frac{\hbar}{m\omega}\left[\frac{1}{\exp(\frac{\hbar\omega}{T})-1} + \frac{1}{2}\right]$$ //

- So $\langle \hat{x}^2 \rangle_T = \frac{\hbar}{m\omega}\left(\frac{1}{\exp(\frac{\hbar\omega}{T})-1} + \frac{1}{2}\right)$

The dimensions of this result are the same as x^2

given $\hbar = m = \omega = 1$

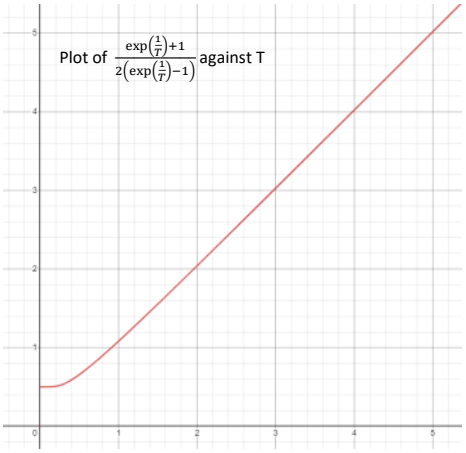
$$\Rightarrow \langle \hat{x}^2 \rangle_T = \frac{1}{\exp(\frac{1}{T})-1} + \frac{1}{2}$$

$$\Rightarrow \coth\left(\frac{1}{2T}\right)+1$$

$$\Rightarrow \langle \hat{x}^2 \rangle_T = \frac{1}{\exp(\frac{1}{T})-1} + \frac{1}{2}$$

$$= \frac{\exp(\frac{1}{T})+1}{2(\exp(\frac{1}{T})-1)}$$

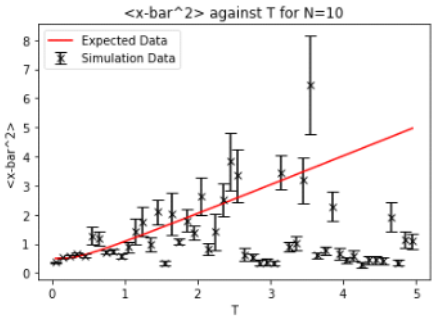
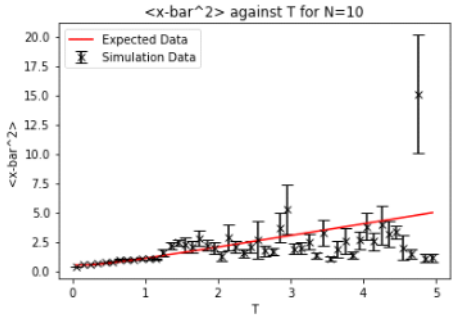
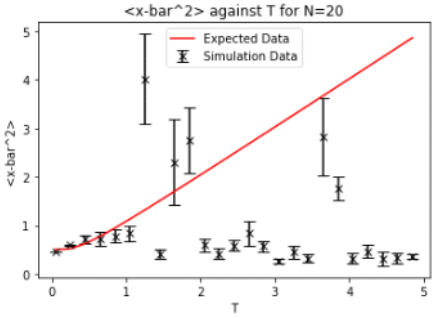
This plot tends to the line y=T for large T, so this is another good indication that it is the correct result.

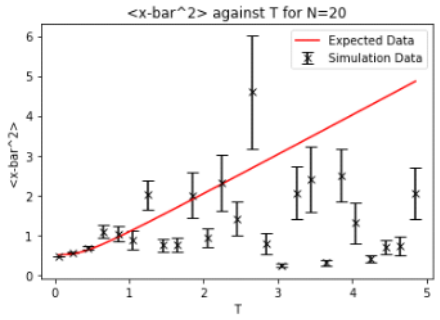
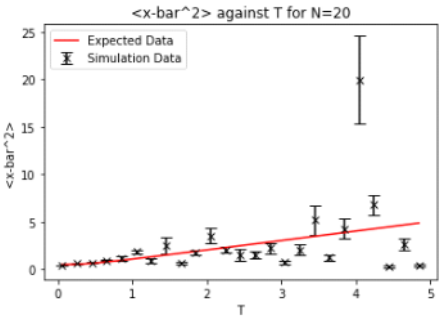
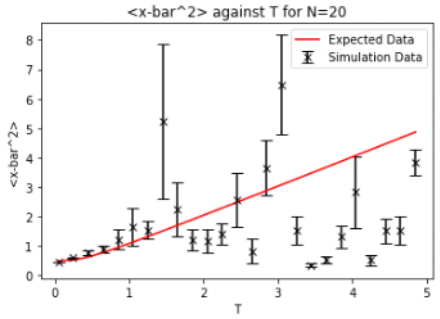
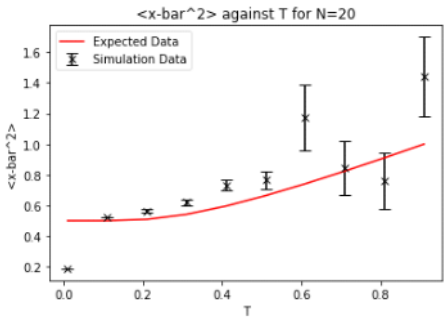
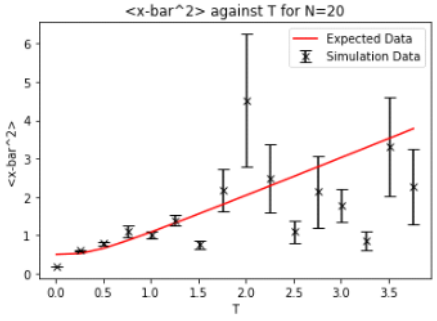


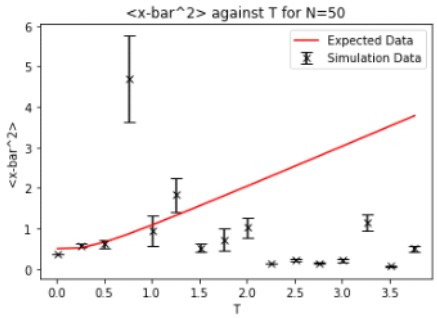
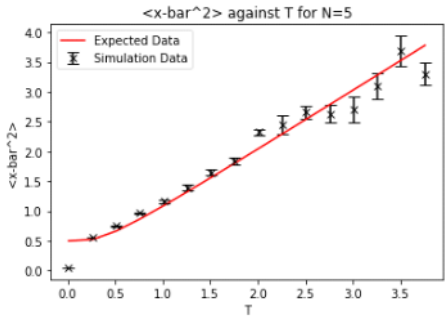
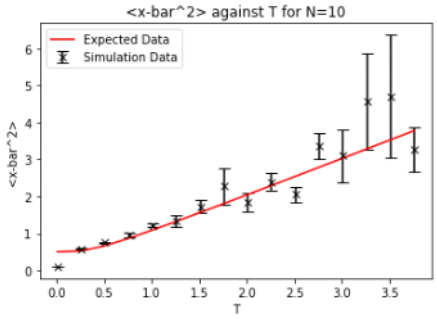
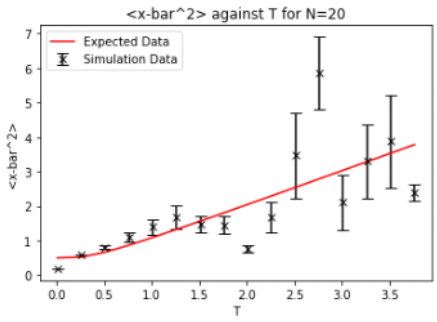
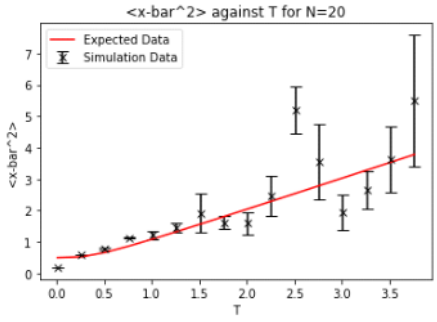
Simulation vs. Expected value for $\langle \hat{x}^2 \rangle_T$ for the QHO: (09-13/11/2022)

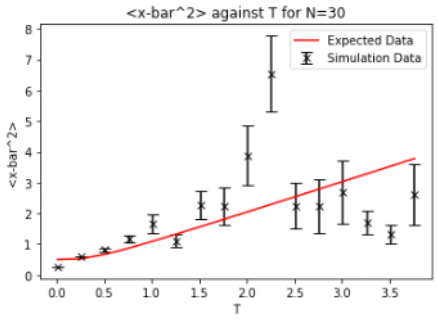
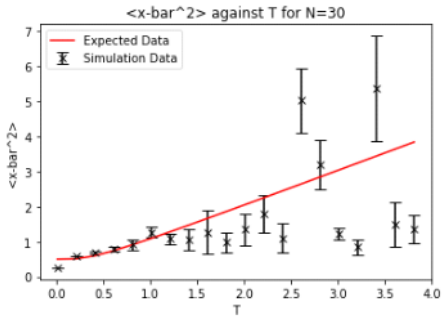
Now with this expectation value we could plot our simulation against the expected results over a large T range.
N is the number of points on the path and 'it' is the number of iterations for a single T value in the MMC method.

Results:

Run	Values	Plot	Comments
1	N=10, it = 50,000, T=[0.05,5] in steps of 0.1		This agreed with the expected data for small T, but did not at the larger end. I think this is due to the number of iterations in the MMC method, so the value found is not accurate enough. We need to do more iterations. **Note:** How for small T the value of $\langle \hat{x}^2 \rangle_T \approx \frac{1}{2}$ as shown last week.
2	N=10, it= 1,000,000 T=[0.05,5] in steps of 0.1		The vast majority of points follow the expected data now that the number of iterations is higher. The drawback is that this takes a long time to run. I will try this for a larger N, hoping to get a similar result. If this is the case (also for N=30,50) then I will be sure that my code is working correctly.
3	N=20, it=1,000,000 T=[0.05,5] in steps of 0.2		The majority of points (especially for larger T) do not follow the expected curve. I will try to up the number of iterations per path to see if this helps.

4	N=20, it=2,000,000 T=[0.05,5] in steps of 0.2		Same issue as above, though slightly better. Will up the number of iterations again.
5	N=20, it=2,500,000 T=[0.05,5] in steps of 0.2		Most points are around the expected curve with a few outliers. However, due to the extreme outlier at T=4, it is hard to see how close some of the points are to the expected curve. I do think this is an issue with the number of iterations, but might double check to see if there are any other issues. Will try with a larger number of iterations and see if that improves the plot first.
6	N=20, it=3,000,000 T=[0.05,5] in steps of 0.2		Similar results as above. The problem seems to occur for large values of T (small beta). I will test the code over small T to see what happens.
7	N=20, it=3,000,000 T=[0.01,1] in steps of 0.1		Similar issues to above, with the data not fitting the curve. I am going to try N=20 for a larger number of iterations hoping that it will get closer to the expected value. I think that the issue must be the number of iterations being too lower (and possibly N) rather than the code - as it worked very well for N=1,2,3. We know that the method approximates the system very well for large N and a large number of iterations per temperature (as the method requires consideration of all possible paths, that is the number of iterations being very large). I am going to try increasing these two variables to see what happens - this will take a long time to run, but there is no other way of testing to see if anything is incorrect as everything works for the simple cases of N=1,2,3 - which I would expect to be a good indication of the simulation working.
8	N=20, it=5,000,000 T=[0.01,4,0.25]		The plot is better than some of the other N=20 results but still doesn't follow the expected data fully. We see for low T, the results are close to the curve (except for T=0.01, which is expected as we know the method fails for very small T). 60% of points are within an error bar of the expected line. This took roughly 30 mins to compile. I am going to try a larger N to see if this will produce a plot closer to the expected value.

9	N=50, it= 6,000,000 T=[0.01,4,0.25]		Simulation does not follow the expected data very well. This could be due to the large increase in path length with a relatively small increase in number of iterations. I am going to try to look at a small N but with a large number of iterations to try to confirm the code agrees with N=5,10. There will be limitations due to the Monte Carlo method needing a large N to approximate integrals - but still the data should roughly follow what is expected.
10	N=5, it=5,000,000 T=[0.01,4,0.25]		The simulation follows the expected data quite closely. It doesn't fully match, but that is expected as we require N to be large for the simulation to tend to the expected data. This gives further evidence to the idea that the code is correct for larger N, but the number of iterations could be the issue. I am going to try for N=10, with the same number of iterations.
11	N=10, it=5,000,000 T=[0.01,4,0.25]		This plot is more promising with 86% of the simulation data being within an error bar of the expected data. This seems to suggest it is an iteration problem with the larger N. I am going to try N=20 and double the number of iterations - this, in theory, means each point along a path gets the same number of potential updates as for N=10. Notice here we have the same issue at very low T.
12	N=20, it=10,000,000 T=[0.01,4,0.25]		The data roughly follows the expected line, but still not as closely as I'd hope. This took quite a long time to run (~1 hr).
13	N=20, it=20,000,000 T=[0.01,4,0.25]		The data roughly follows the expected curve. 69% of points are within an error bar of the expected curve. This looks very similar to the graph above, so doubling the number of iterations did not change the overall outcome that much in this case. I am going to try N=30 with the same number of iterations to see if this week produce data closer to the expected curve. This took ~2hrs to run. We again have the value for very low T not near to the expected data. Most points are roughly near to the curve, and I believe this is due to a relatively low N value and the statistical nature of the method.

14	N=30, it=20,000,000 T=[0.01,4,0.25]		Some points follow the expected data, but not general pattern tending to what is expected. Will do this again just with more points to see if it will tend to what is expected.
15	N=30, it=20,000,000 T=[0.01,4,0.2]		We see that for low T the points follow the curve. It is for the higher temperatures that this isn't the case. I still believe this is due to the number of iterations. But increasing them anymore will take a long time, so I wonder if there is a more efficient way of calculating things to help speed this process up.

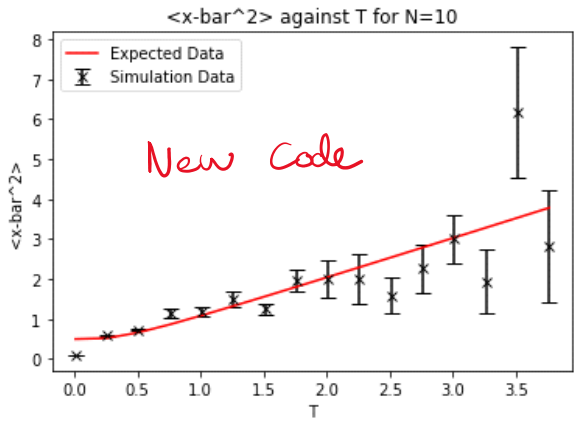
Conclusion: (15/11/2022)

For lower N, after a large number of iterations, the data follows the expected curve. This isn't happening for larger N - probably due to the number of iterations. Each run at a large N and large number of iterations takes a very long time to complete. I plan on discussing these results at our next meeting and seeing if my analys is make sense - especially by comparing with Toby's results. If he is able to produce more accurate results for larger N than I am, I shall look at my code again. But given that things work for a smaller N, I be lieve this is just due to the number of iterations - so making the code more efficient will be a good next step.

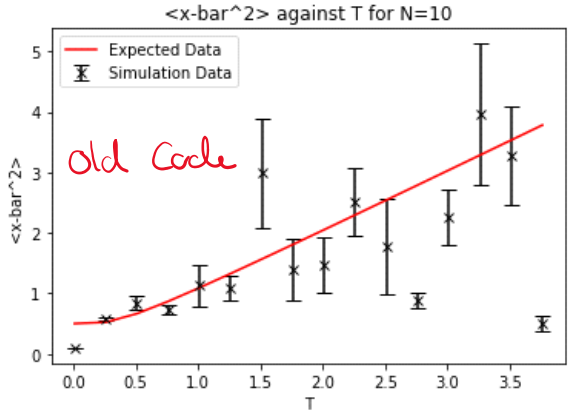
Note: (16/11/2022)

Having had a look through my code **I realised some of the terms of the DS calculation were missing - this meant that some points were not updating when they were supposed to, This means more iterations were being run than actually needed to be to produce more accurate results. I will produce some more graphs with this change to see if this solves the issue.** (Discovered on 16/11/2022)

I quickly ran the case for N=10 for it=500,000: producing this graph:



N=10 for it=500,000 in the older code:



This is a lot better than previous N=10 for a similar number of iterations. I will do some more graphs again in the next week 's work, but I believe this solves the issues. The number of iterations therefore doesn't need to be as high for more accurate data as more points are being updated. This is especially true for larger T, as is expec ted due to the epsilon terms which depend on T.

TK

Saturday, November 12, 2022 2:06 PM

Please expand tab for full weekly report

Theory

Saturday, November 12, 2022 2:07 PM

Outline:

- This week I will be calculating thermal expectation value with respect to temperature
- Use this as the new expected curve
- Run simulation for large N, checking to make sure that at least 68.2% fit within error bars

Background:

$$< \hat{x}^2 >_T = \frac{\left(\sum_{n=0}^{\infty}(<n|\hat{x}^2|n>e^{-\frac{E_n}{T}})\right)}{\sum_{n=0}^{\infty} e^{-\frac{E_n}{T}}} = \frac{\left(\sum_{n=0}^{\infty}\frac{\hbar}{m\omega}(n+\frac{1}{2})e^{-\frac{\hbar\omega(n+\frac{1}{2})}{T}}\right)}{\sum_{n=0}^{\infty} e^{-\frac{\hbar\omega(n+\frac{1}{2})}{T}}}$$

Plan:

- Calculate $< \hat{x}^2 >_T$
- Use this as expected curve
- Run for N=10, N=20 & N=50
- Ensure that at least 68.2% fit within error bars

Method:

$$g(\alpha) = \sum_{n=0}^{\infty} e^{-\alpha n}$$

$$= 1 + e^{-\alpha} + e^{-2\alpha} + \cdots$$

$$= \frac{1}{1-e^{-\alpha}}$$

$$-g'(\alpha) = \sum_{n=0}^{\infty} n e^{-\alpha n}$$

$$= -\frac{d}{d\alpha}\frac{1}{1-e^{-\alpha}}$$

$$< \hat{x}^2 >_T = \frac{\frac{\hbar}{m\omega}\left[\sum_{n=0}^{\infty} n e^{-\frac{\hbar\omega(n+\frac{1}{2})}{T}} + \frac{1}{2}\sum_{n=0}^{\infty} e^{-\frac{\hbar\omega(n+\frac{1}{2})}{T}}\right]}{\sum_{n=0}^{\infty} e^{-\frac{\hbar\omega(n+\frac{1}{2})}{T}}}$$

$$\sum_{n=0}^{\infty} e^{-\frac{\hbar\omega(n+\frac{1}{2})}{T}} = \sum_{n=0}^{\infty} e^{-\frac{\hbar\omega n}{T}} e^{-\frac{1}{2}\frac{\hbar\omega}{T}}$$

$$= e^{-\frac{\hbar\omega}{2T}} \sum_{n=0}^{\infty} e^{-\frac{\hbar\omega n}{T}}$$

$$= e^{-\frac{\hbar\omega}{2T}}\left(\frac{1}{1-e^{-\frac{\hbar\omega}{T}}}\right)$$

$$\& \sum_{n=0}^{\infty} n e^{-\frac{\hbar\omega(n+\frac{1}{2})}{T}} = e^{-\frac{\hbar\omega}{2T}} \sum_{n=0}^{\infty} n e^{-\frac{\hbar\omega n}{T}}$$

$$= e^{-\frac{\hbar\omega}{2T}}\left(-\frac{d}{d\alpha}\frac{1}{1-e^{-\alpha}}\right)$$

$$(1-e^{-\alpha})^{-1}$$

$$= -e^{-\frac{\hbar\omega}{2T}}(-1)(-e^{-\alpha})(1-e^{-\alpha})^{-2}$$

$$= \frac{-e^{-\frac{\hbar\omega}{2T}} e^{-\frac{\hbar\omega}{T}}}{(1-e^{-\frac{\hbar\omega}{T}})^2}$$

$$\therefore < \hat{x}^2 > = \frac{\frac{\hbar}{m\omega}\left[\frac{-e^{-\frac{\hbar\omega}{2T}} e^{-\frac{\hbar\omega}{T}}}{(1-e^{-\frac{\hbar\omega}{T}})^2} + \frac{1}{2} e^{-\frac{\hbar\omega}{2T}}\left(\frac{1}{1-e^{-\frac{\hbar\omega}{T}}}\right)\right]}{e^{-\frac{\hbar\omega}{2T}}\left(\frac{1}{1-e^{-\frac{\hbar\omega}{T}}}\right)}$$

$$= \frac{\hbar}{m\omega}\left[\frac{e^{-\frac{\hbar\omega}{T}}}{1-e^{-\frac{\hbar\omega}{T}}} + \frac{1}{2}\right]$$

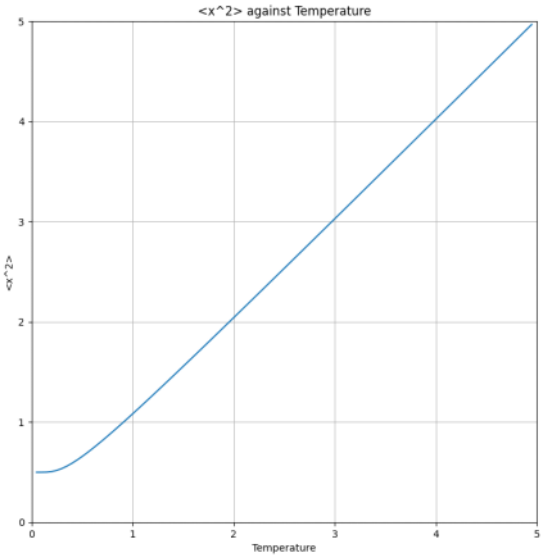
For our case: $\hbar = m = 1$

& $\omega = \sqrt{\frac{k}{m}} = 1$

$\therefore \langle x^2 \rangle = \frac{e^{-\frac{1}{T}}}{1 - e^{-\frac{1}{T}}} + \frac{1}{2}$

$= \frac{1}{e^{\frac{1}{T}} - 1} + \frac{1}{2}$

Sanity Check: $\left(\frac{1}{e^{\frac{1}{T}} - 1} + \frac{1}{2} \right)_{T \to 0} \to \frac{1}{2}$ ✓

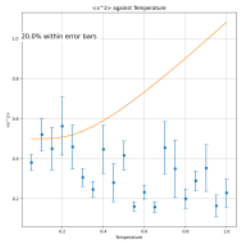
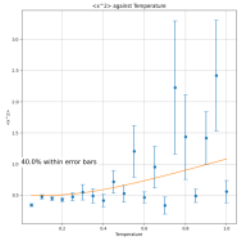
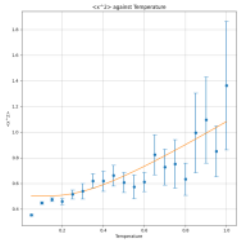
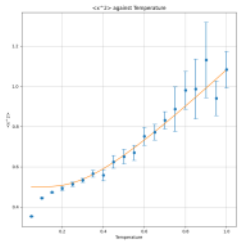


As expected.

Results

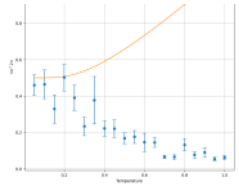
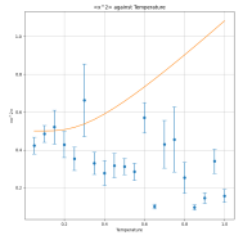
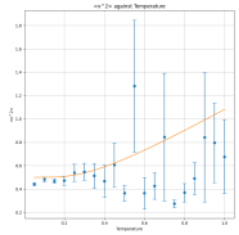
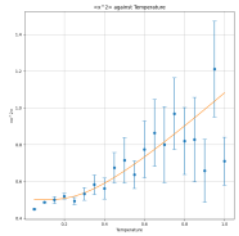
Sunday, November 13, 2022 11:33 AM

N=10:

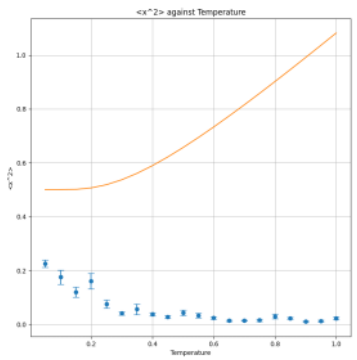
Details	Graph	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=10 Iterations =998		20.0% within error bars	Elapsed time = 1.2002291679382324	Larger temperature values are not deviating far from 0, because an update is less likely to be accepted, so more iterations are needed
N=10 Iterations =9998		40.0% within error bars	Elapsed time = 31.517925262451172	Now more deviation at higher temperatures, due to more iterations
N=10 Iterations =99998		60.0% within error bars	Elapsed time = 1774.0712595	Higher percentage within error bars, which is very promising. However, large errors for large temperature values
N=10 Iterations =999998		70.0% within error bars	Elapsed time = 169,653.631921	Within one standard deviation of expected.

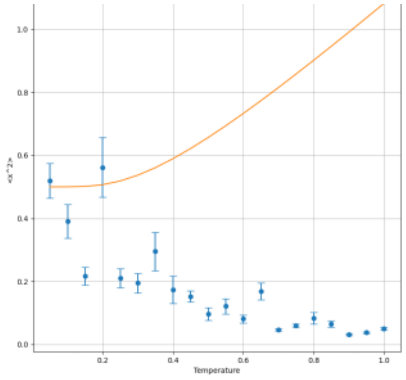
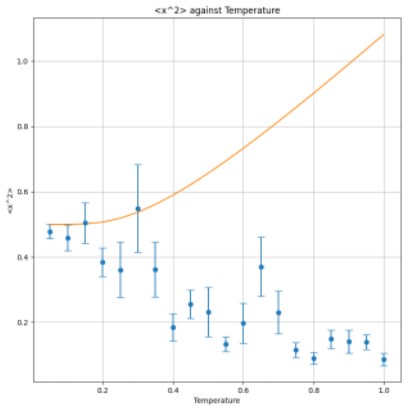
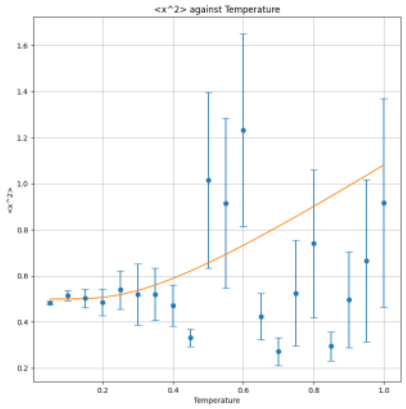
N=20:

Details	Graph	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=20 Iterations		15.0% within	Elapsed time = 2.5378797054	Even greater pronounced lack of deviation for higher temperatures

=998		error bars	29077	shown, implying that this becomes more of an issue the higher the value of N.
N=20 Iterations =9998		15.0% within error bars	Elapsed time = 46.167520999 90845	Less improvement for increasing iterations for larger N is shown, this means that larger N, need more iterations to produce reliable results.
N=20 Iterations =99998		50.0% within error bars		
N=20 Iterations =999998		75.0% within error bars		Within one standard deviation of expected.

N=50:

Details	Graph	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=50 Iterations=998		0.0% within error bars	Elapsed time = 5.5481483936309814	Much worse than previous attempts using smaller N. Shows the necessity for large iterations for larger N. This is due to the fact that as N increases, epsilon decreases, and hence the action increases, and hence probability of accepting an update decreases.
N=50 Iteration		10.0% within	Elapsed time =	Now more reliable for low temperature

s=9998		error bars	140.591374 15885925	values, but still need a lot more iterations
N=50 Iterations=99998		15.0% within error bars		
N=50 Iterations= 999998		50.0% within error bars		Time now taking far too long, need to improve efficiency of code next week to be able to run large N in reasonable time frames.

Code

Sunday, November 13, 2022 11:35 AM

```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #time how long code runs for

plt.close('all')

"""values"""
k = 1
beta = 1
m = 1
N = 50

"""functions"""
def action(x,beta,N,m,k):
    epsilon = beta/N
    S_sum = 0
    x = np.append(x,x[0])
    for j in range(0,N):
        S_sum += m/2*((x[j+1]-x[j])/epsilon)**2 + 0.5*k*x[j]**2
    S_N = epsilon * S_sum
    return S_N


def updateX(beta,N,m,k):
    """Pick arbitrary value for x"""
    x_values = np.zeros([N])
    x_values = np.append(x_values,x_values)
    x_values = np.reshape(x_values, (2,N))
    for i in range(9998):
        """duplicate array"""
        x_values = np.append(x_values,x_values[-1])
        x_values = np.reshape(x_values, (i+3,N))

        """Choose which x value to update"""
        point_selected = random.randint(0,N-1)
        x = x_values[-1, point_selected]

        """Update x and append"""
        delta = random.uniform(-1,1)
        x_updated = x+delta

        path = x_values[-1]
        path[point_selected] = x_updated
```

```
        """Test update"""
        if action(path,beta,N,m,k) <= action(x_values[-2],beta,N,m,k):
            pass
        else:
            prop = np.exp(action(x_values[-2],beta,N,m,k)-action(path,beta,N,m,k))
            if random.random() <= prop:
                pass
            else:
                x_values[-1, point_selected] = x
    #x_values = np.delete(x_values,0)
    return x_values

"""averages function"""
def meanError(x_values):
    x_split = np.split(x_values,10)
    x_split = np.delete(x_split,0,0)
    x_split = np.split(x_split,9)
    sum = 0
    x_split_mean = np.array([])
    for i in range(0,len(x_split)):
        sum += np.mean(x_split[i])
        x_split_mean = np.append(x_split_mean,np.mean(x_split[i]))
    mean = sum/len(x_split)
    std_err = np.std(x_split_mean)/(np.sqrt(len(x_split_mean))-1)
    return mean, std_err

"""plot figure"""
fig = plt.figure(figsize=(9,9))

"""plot <x^2> against Temperature"""
means = np.array([])
std_errs = np.array([])
temperature_values = np.arange(0.05,1.001,0.05)

start_time = time.time()
for i in range(0,len(temperature_values)):
    meanErrors = meanError(updateX(1/temperature_values[i],N,m,k)**2)
    means = np.append(means,meanErrors[0])
    std_errs = np.append(std_errs,meanErrors[1])
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))

ax5 = fig.add_subplot(111)
ax5.grid()
ax5.errorbar(temperature_values,means, yerr=[std_errs,std_errs], capsize=5, fmt="o")
ax5.set_xlabel('Temperature')
ax5.set_ylabel('<x^2>')
ax5.title.set_text('<x^2> against Temperature')

"""plot expected curve"""
expected = (np.exp(1/temperature_values)+1)/(2*(np.exp(1/temperature_values)-1))
ax5.plot(temperature_values,expected)

inErrorBars = 0
for i in range(len(means)):
    if means[i]-std_errs[i]<=expected[i] and means[i]+std_errs[i]>=expected[i]:
```

```
        inErrorBars += 1
inErrorsPercentage = inErrorBars/len(means)*100
#ax5.text(0,1,str(inErrorsPercentage)+'% within error bars',fontsize=15,backgroundcolor='w')
```

Week 9

Supervisor Meeting: 16/11/2022

Sunday, November 20, 2022 3:17 PM

Outline:

- Looked at optimising code, by decreasing memory usage:
  - Store means of paths instead of the full path
  - Update a sequential set of points
- Looked at zero-point energy/motion & what this means for its wavefunction at absolute zero

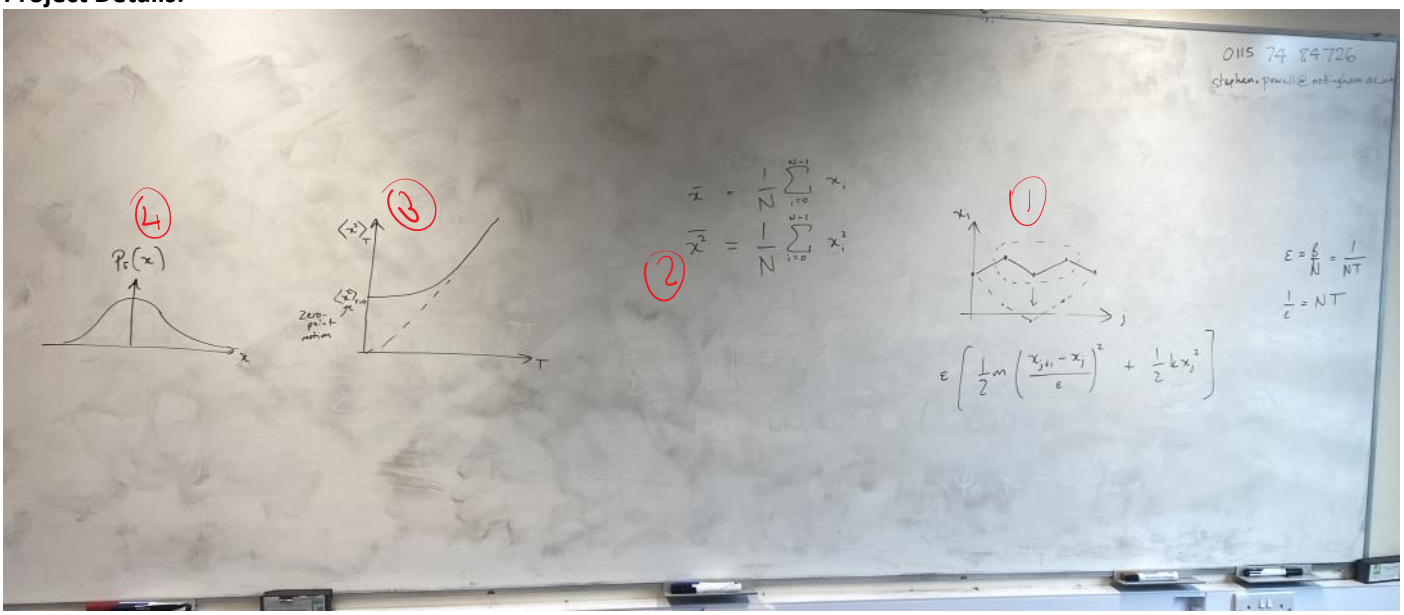
Tasks:

- Store means of paths instead of the full path at each iteration
- Potentially updating a sequential set of points
- Run for large N and compare zero-point motion value to expected

Distribution of Tasks:

- Both would produce graphs and data using our codes to confirm everything was working.
- We would then compare our results to see if any issues arose.

Project Details:



1) Here we looked at updating a sequential set of points, so that multiple points are being updated but there is less computation needed for the change in action, because the difference between the points in the set remains constant throughout the change, so this does not need to be computed again individually.
2) Store means of paths rather than the paths themselves at each iteration, this will lower the amount of data that needs to be stored, and should help optimise the program and decrease running time and memory usage.
3) The zero-point energy is the lowest possible energy of a quantum system. At this zero-point energy, the particle still has some uncertainty in position, as shown in this graph as T->0
4) Due to zero-point energy, at absolute zero temperature, the particles retain some vibrational motion, leading to a defined probability of finding the particle in a particular position, as given by the wavefunction.

Comments:

The following page is helpful for information regarding the zero-point energy: [Zero-point energy - Wikipedia]

Textbook regarding the zero-point energy:

1) F.Schwabl (1992). Quantum Mechanics, Springer-Verlag. Chapter 3.1

(TW) Statistical Results for Large N - Part 3 (16-20/11/2022)

16 November 2022

Statistical Results for Large N: (16/11/2022)

After discovering what was wrong with my code, the plan for this week is to produce plots for N=10,20,30,40,50 that agree with the expected curve - and shouldn't have too large of a run time (as the number of iterations will be lower). I may also change slightly the way the method run to produce plots to take up less memory- that is by storing the values of the mean squared value of x rather than the full paths.

Changes to code:

As mentioned as the end of the entry for week 8, I discovered a few terms were missing from my DS calculation. I forgot to include the old terms (those including the point that would be updated) in the calculation. DS is now:

Let x_i be the point which is being updated to x_i':

$$DS = \varepsilon\left[\frac{m}{2}\left(\frac{x_{i+1}-x_i'}{\varepsilon}\right)^2+\frac{m}{2}\left(\frac{x_i'-x_{i-1}}{\varepsilon}\right)+\frac{1}{2}kx_i'^2-\frac{m}{2}\left(\frac{x_{i+1}-x_i}{\varepsilon}\right)^2-\frac{m}{2}\left(\frac{x_i-x_{i-1}}{\varepsilon}\right)^2-\frac{1}{2}kx_i^2\right]$$

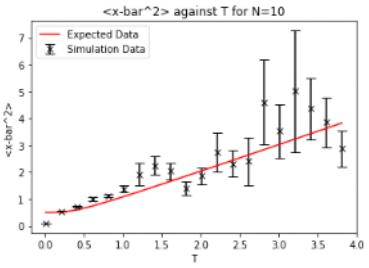
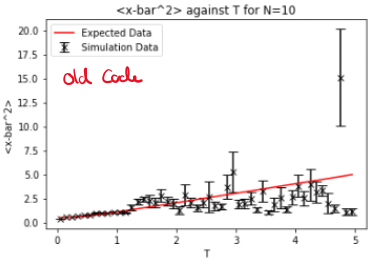
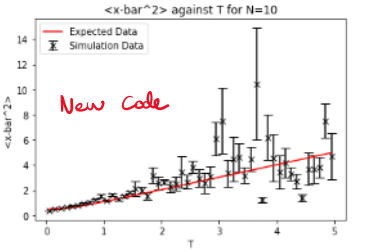
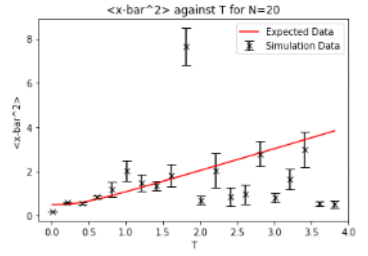
So $S_2 = S_1 + DS$

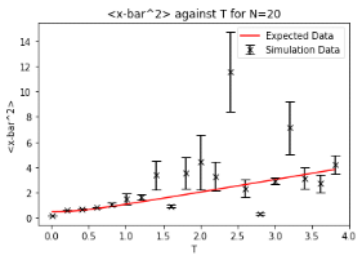
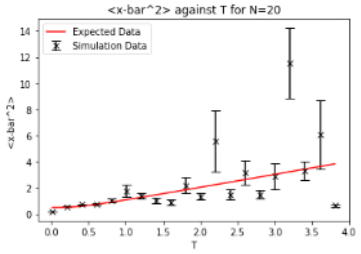
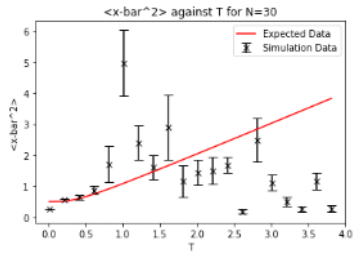
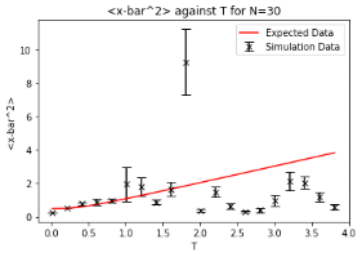
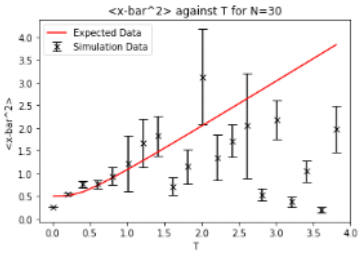
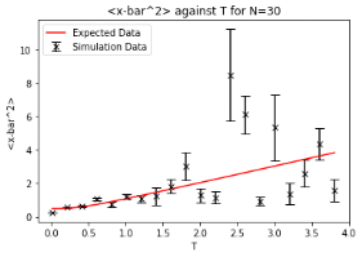
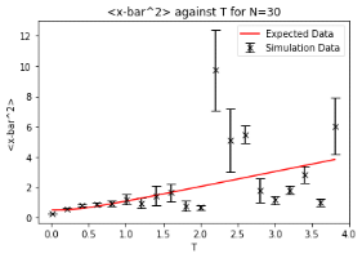
It was the three negative terms that I forgot to include originally. This meant that DS was larger than it was supposed to be- meaning updates were permitted less frequently than they ought to have been. This meant I needed to iterate over the MMC method more to produce more updates - causing the algorithm to run for a longer time.

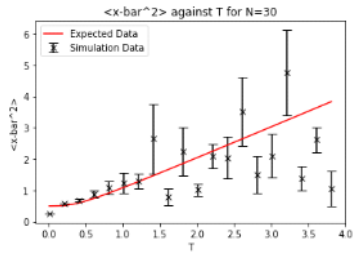
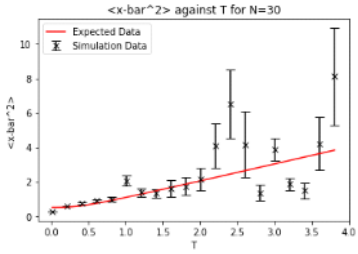
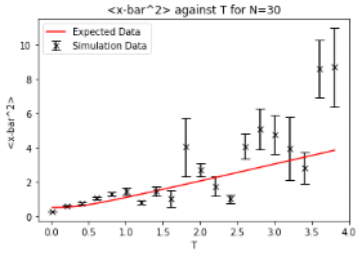
With this change DS can be negative (meaning the update is allowed) and $p = \exp(-\Delta S)$ is larger, meaning a higher probability of a point being updated as there is a larger range for the randm number to fall into.

Simulation vs. Expected value for $\langle \hat{x}^2 \rangle_T$ for the QHO: (16/11/2022)

With this change, I will produce plots for N=10,20,30,40,50. Similar to last week, but hopefully the plots will follow the expected data more closely. There could be larger error bars for large T - this was discussed in our supervisor meeting. If this is the case, a possible extension to the project could be done to reduce this error. This will be spoken about next week.

Run	Values	Plot	Comments
1	N=10; it = 500,000; T=[0.01,4] in steps of 0.2		This plot is a lot closer to the expected curve with these changes as compared to last week. As mentioned above we do have larger error bars for large T - will look into a way of sorting this next week, but will up the number of iterations first. A few points for low T are above the line - I expect this is due to N being too small for our simulation. We notice for very low T again, the simulation and expected data do not agree. Before changing N, I want to change the number of iterations to it=1,000,000 so I can directly compare to the code from last week. I will also change the temperature range to get a direct comparison. This plot took ~4 mins to produce.
2	N=10, it= 1,000,000 T=[0.05,5] in steps of 0.1	 	The new plot is much better than the older one, with many more points within an error bar of the expected curve. This took ~20mins to run - I know I do not need to do such a high number of iterations and points. Given N=10 works, I plan to do N=20 with the same number of points as above. I will produce a plot for it = 500,000, and see if more are needed.
3	N=20; it = 500,000; T=[0.01,4] in steps of 0.2		Moving onto N=20. This run took 4 mins, but clearly more iterations are needed to make it more accurate.

4	N=20; it = 1,000,000; T=[0.01,4] in steps of 0.2		This run took 8 mins. This is more accurate than above, though still not close enough to the expected line. Will up the number of iterations.
5	N=20; it = 1,500,000; T=[0.01,4] in steps of 0.2		This took 13 mins and produced a plot quite close to what is expected. Some inaccuracy will be due to N, and I'd expect the plots to get closer to the expected curve for a larger number of iterations. But I'll try a higher N, as this better approximates the quantum system. It is important to note, that in order to get a plot close to this in the old code, the number of iterations was 20,000,000 - so clearly this is a lot better. A larger number of iterations would push the points closer to the curve.
6	N=30; it = 1,500,000; T=[0.01,4] in steps of 0.2		This took 15 mins. More iterations are needed.
7	N=30; it = 2,000,000; T=[0.01,4] in steps of 0.2		This took 20 mins. Again, more iterations are needed.
8	N=30; it = 3,000,000; T=[0.01,4] in steps of 0.2		This took 30 mins. The points are getting closer to the expected curve - especially at higher T. Though still not accurate enough - so will increase the number of iterations.
9	N=30; it = 3,500,000; T=[0.01,4] in steps of 0.2		This took 37 mins. Points are closer to the curve, though still not as accurate as I would like.
10	N=30; it = 4,000,000; T=[0.01,4] in steps of 0.2		This took 40 mins. Closer to the curve than above, but want it to be more accurate.

11	N=30; it = 6,000,000; T=[0.01,4] in steps of 0.2		This took 65 mins. Most points are near to the curve, so I think the number of iterations is close to being the right number. It is worth noting a result like this took iterations=20,000,000 last time and roughly 180 mins to run - so clearly better. I am going to increase the number of iterations until more points at a higher T are closer to the curve.
12	N=30; it = 6,500,000; T=[0.01,4] in steps of 0.2		This took 64 mins. We are getting closer to the curve, though more iterations are needed.
13	N=30; it = 7,000,000; T=[0.01,4] in steps of 0.2		This took 73 mins. For lower T the points are very close to the expected curve, not as much for larger T. More iterations are still needed, and possibly more runs at the same number of iterations.

Conclusions:

The above graphs tend to be closer to the expected curve as compared to the old code. However, I still believe more iterations are needed. Before looking into this I plan on changing slightly how the method runs - I will calculate the average value of the path as soon as it is generated rather than after all paths have been generated for a given beta. This won't have a significant change on the run time of the simulation, but will mean less memory is needed as such large arrays are not needed to be stored.

General comments:

By thinking about the QHO it is clear why for large T the system tends to the classical result (N=1) - that is $\langle x^2 \rangle_T$ = T, as at high T the spacing between energy levels becomes very small, and the system can take on any value - the discrete values of energy tend towards a continuous system - the classical result.

The simulation breaks for small T, as the QMC method relies on $\varepsilon = \frac{\beta}{N} = \frac{1}{NT} < 1$. So for small T, $\varepsilon > 1$ and the simulation no longer approximates the QHO. Increasing N therefore allows us to look at a smaller T, as this reduces the size of ε.

Changing the method: (17/11/2022)

The overall change is quite simple, instead of running the 'pathavg' function after all the paths are created (for 10^6 iterations, this will be a very large set of arrays that will then be averaged over per paths), I shall return from the MMC method the average of a path as soon as it has been iterated over. To do this I changed the way the MMC method works.

First the 'MMC' method is run, with the same inputs as before. However, when an update is chosen another method runs the 'updateavg' method. This updates a point along a path, as occurred in the old MMC method, and returns the average of the updated path (if the update is allowed, otherwise returns the average of the old path), the full updated path (or old path if not updated) and the value of the action of this path. This data is used in a loop in the 'MMC' method. The 'MMC' method then produces an array of average values for each path, rather than the full array of paths - cutting down on the amount of memory used. These average values are then squared and put through the 'errordata' function as before.

This also means the 'pathavg' function is no longer needed - as is incorporated in the new 'MMC' method. The code is below:

```
def updateavg(xarr, S1, N, beta, m, k):
    #This fucntion updates a point along a given path. We first define some variables and calculate the change in action between the old and new paths.
    eps = beta/N
    pos = random.randint(0,N-1)
    delta = random.uniform(-1,1)
    xtemp = np.copy(xarr)
    xtemp[pos] = xarr[pos] + delta
    xtemp[N] = xtemp[0]
    DS = eps*((m/2)*((xtemp[pos+1]-xtemp[pos])/eps)**2 + (m/2)*((xtemp[pos]-xtemp[pos-1])/eps)**2 + 1/2*k*(xtemp[pos])**2
          - (m/2)*((xarr[pos+1]-xarr[pos])/eps)**2 - (m/2)*((xarr[pos]-xarr[pos-1])/eps)**2 - 1/2*k*(xarr[pos])**2)
    S2 = S1+DS
    #If this change in action is less than or equal to zero, the update is allowed. The path is updated, as is the action.
    if DS <= 0:
        xarr = np.copy(xtemp)
        S1 = S2
    else:
        #If DS is positive, we set a probability (p) to be the negative exponent of this DS and find a random number between (0,1).
        #If this random number is less than p, the update is allowed. If not, the old path is kept.
        p = np.exp(-DS)
        r = random.uniform(0,1)
        if r < p:
            #If the update is allowed, the new path and new action are stored.
            #If the update isn't, the old path and action are kept.
            xarr = np.copy(xtemp)
            S1 = S2
    #This method returns a path, the mean of the path, and the action of the path.
    return (xarr, np.mean(xarr), S1)

def MMC(N, beta, m, k, it):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    S1 = 0
    #Avgpath list stores the averages of all path generated, the first path has an average of 0.
    avgpath = [0]
    #We iterate over the number supplied-1 (as there are an iteration number of paths, the first being all zeros).
    #We use the updateavg function to update our paths, with the path and action of the path returned being stored for the next iteration.
    #The average value of the path is appended to the avgpath list.
    for i in range(it-1):
        update = updateavg(x, S1, N, beta, m, k)
        x = np.copy(update[0])
        avgpath.append(update[1])
        S1 = update[2]
    return np.asarray(avgpath)
```

```
def errordata(xarr):
    #Finds the mean and standard error for a given array of points
    sects = np.split(xarr, 10) #Splits array into 10 sections, on which we will calculate means and std deviations
    means = []
    for i in range(8):
        #for each section - except the first - we find the mean of each section
        section = sects[i+1]
        means.append(np.mean(section))
    #calculates the std error for the 9 sections we are interested in
    error = np.std(means)/np.sqrt(8)
    totalmean = np.mean(xarr)
    #returns a tuple of the mean and std error for that set of x values
    return (totalmean, error)

```

This code will produce results that is the same as the old version, just in a more optimised way in terms of storage (meaning for large number of iterations, the chance of memory overflow is mitigated). I quickly tested the case of N=1 for x and x^2 to see if the same results were produced as before. They were. (Note this change left the p input in the 'updateavg' method, this will be removed later, but it does not affect the results below, the p being present before may have also been causing some of the errors, but mainly due to when the average was calculated).

So from now I will use this updated method.
The graphs are produced using this code:

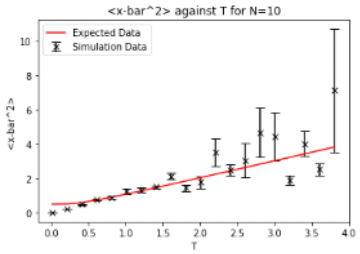
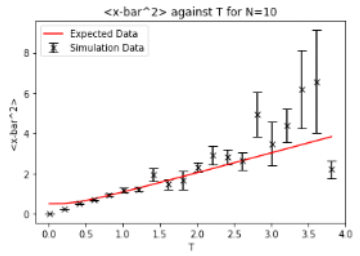
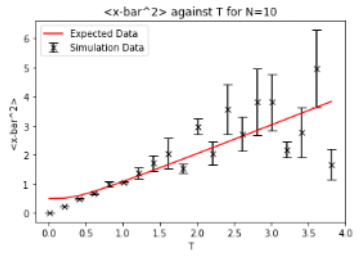
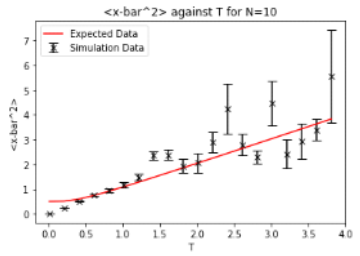
```
N = 10
m = 1
k = 1
it = int(100e4)
T = np.arange(0.01,4,0.2)
beta = 1/T
means = []
errors = []

for i in range(np.size(beta)):
    paths = (MMC(N, beta[i], m ,k ,it))**2
    data = errordata(paths)
    means.append(data[0])
    errors.append(data[1])

y = np.asarray(means)
f = (np.exp(1/T)+1)/(2*(np.exp(1/T)-1))
plt.errorbar(T, y, xerr = None, yerr = np.asarray(errors), marker='x', color='black', ecolor='black', linestyle = 'none', capsize = 5, label='Simulation Data' )
plt.ylabel('<x-bar^2>')
plt.xlabel('T')
plt.plot(T, f, color='red', label='Expected Data')
plt.legend()
plt.title('<x-bar^2> against T for N='+str(N))
```

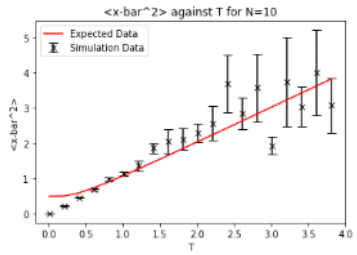
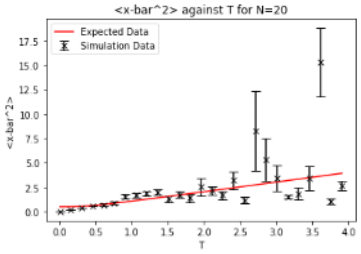
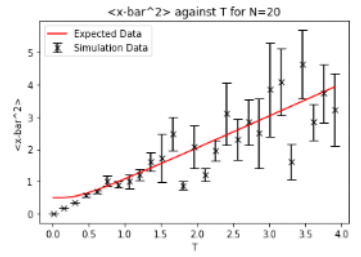
Notice the only difference to before is that the 'pathsavg' function is no longer needed.

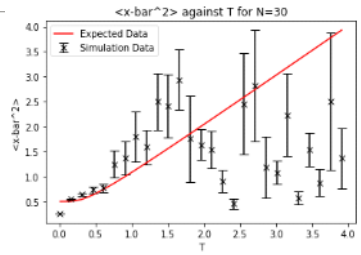
Simulation vs. Expected value for $\langle \hat{x}^2 \rangle_T$ for the QHO: (17/11/2022)

With this change, I plan on creating graphs for N=10,20,30. Only once I am happy with plots for these three will I consider looking at higher N - a higher N only makes the simulation more accurate closer to T=0. The problem at the moment is higher values of T.

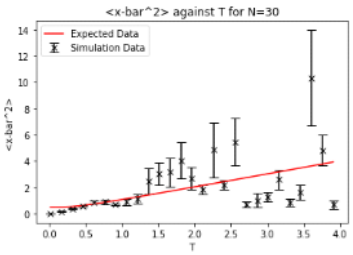
(19/11/2022) The plots below are also incorrect, as discussed below.

Run	Values	Plot	Comments
1	N=10; it = 1,000,000; T=[0.01,4] in steps of 0.2		This plot is similar to the above produced at it=1,000,000, except few points. This took 10 mins to produce - roughly the same as before. We see the same drop for low T as before. We expect for around T=0.1* the simulation will no longer agree with the expected data. I am quite happy with this result, but will do the same number of iterations again to see if we can get some more points closer to the curve. The larger error bars for larger T will be looked into next week. *For the method we have assumed $\varepsilon < 1 \implies \frac{\beta}{N} = \frac{1}{10T} < 1 \implies T > 0.1$ for our assumptions to work.
2	N=10; it = 1,000,000; T=[0.01,4] in steps of 0.2		9 mins
3	N=10; it = 1,000,000; T=[0.01,4] in steps of 0.2		9 mins
4	N=10; it = 1,500,000; T=[0.01,4] in steps of 0.2		13 mins

5	N=10; it = 1,500,000; T=[0.01,4] in steps of 0.2		This took 13 mins to run. With roughly 70% of points within an error bar of the expected data I am happy with the plot. We see the points do not follow the curve for low T - as discussed before. At higher T the error bars of larger - this will be discussed next week. I am happy to move onto N=20, starting with double the number of iterations.
6	N=20; it = 3,000,000; T=[0.01,4] in steps of 0.15		
7	N=20; it = 3,000,000; T=[0.01,4] in steps of 0.15		At this point I started to notice a difference between the older way of calculating the points (using the 'pathavg' function) and the new way. I needed to look into this before carrying on.



Old Way (Pathavg)



New Way

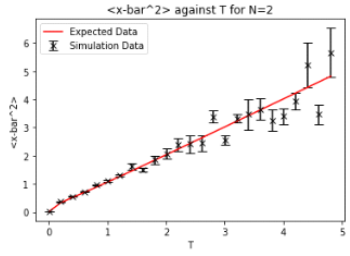
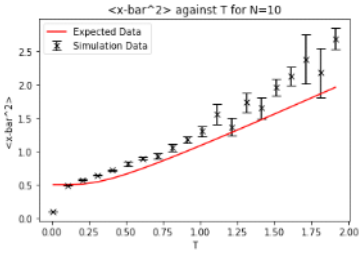
Finding the errors:(19/11/2022)
As you can see the there is a difference in error bar length, but also the behaviour near to zero- the older way behaves like we would expect, with a large drop at around 0.03, this doesn't happen in the new method. After looking at both ways in detail, calculating means by hand etc, I noticed in thenew method I was not calculating $\overline{x^2}$ but $\overline{x}^2$.

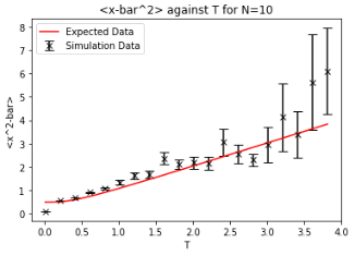
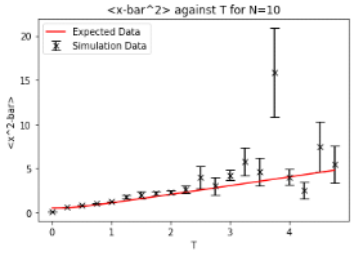
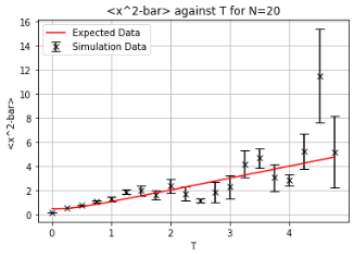
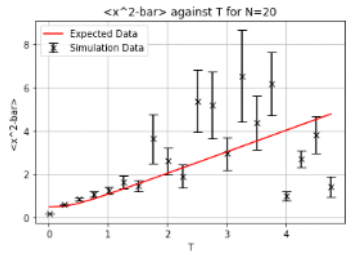
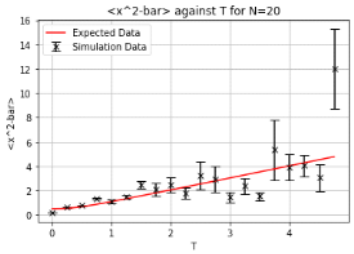
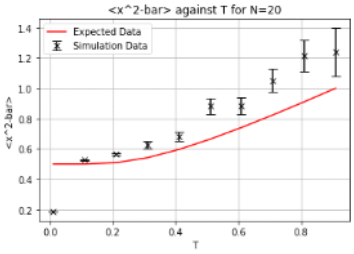
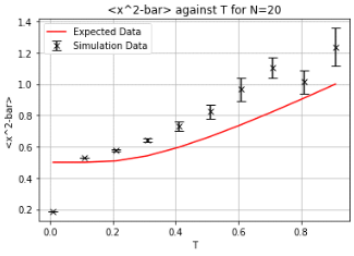
I decided to test, by changing when the power was taken, if I can produce the expected results with the newer code. Once thisissue is fixed, and tested with some of these results shown below, I will create a new page with the final results for different N with the newest version of the code. Theplots for smaller N (1,2,3) are correct, as they were done before the change in how DS calculated. **But the plots from N=10 onwards all contain some sort of error.**

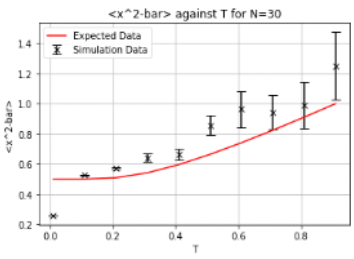
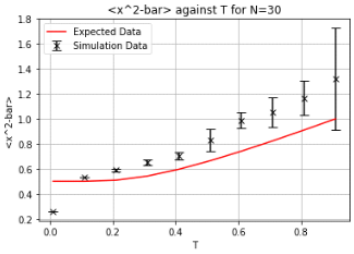
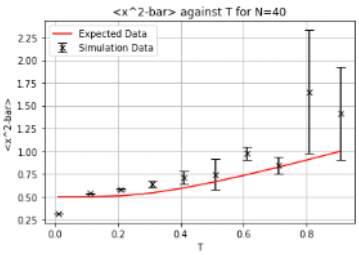
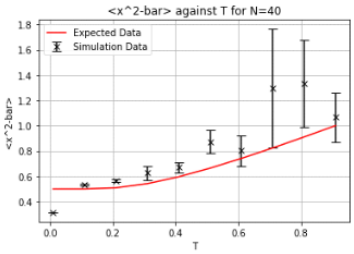
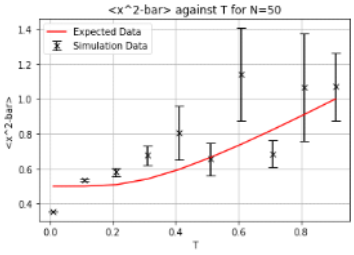
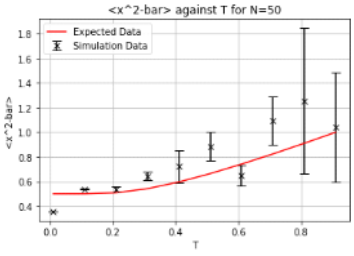
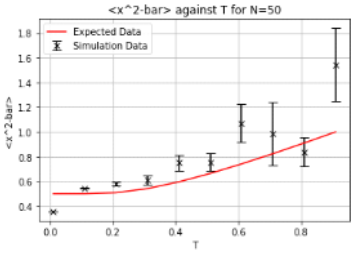
Reflection (23/11/2022):

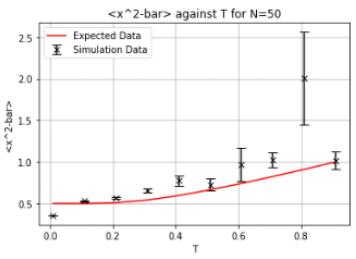
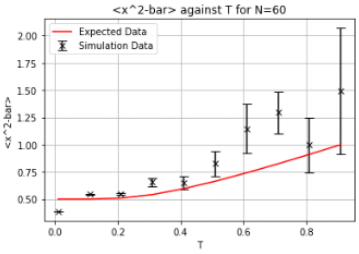
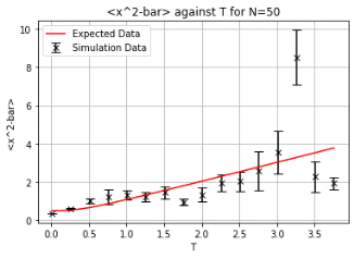
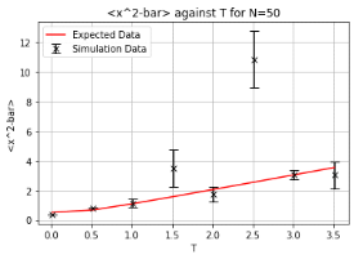
I believe the error with the new code was not when the square was taken, but using p as the variable for the power and the pobability. The newest update has solved this, and all results are correct. I just wanted to give the true reason for the erroneous results. This is commented on later on in this page as well.

Producing Results for larger N after error sorted:
The new MMC method now required a parameter that raises the values of the array to the value of this parameter before takingthe mean of the array - this will be set to 2 to get what we need.
After running the code to ensure it works, I first looked at N=2 and then N=10,20.

Run	Values	Plot	Comments
1	N=2; it = 50,000; T=[0.01,5] in steps of 0.2		I produced the plot for N=2 to get back the result from before. This confirms with the expected result for N=2. The simulation data agrees with the expected value for 76% of points. I am happy with the code agreeing with the expected result, so will go to N=10 to get accurate data. I will look at larger T and smaller T as well - to look at the drop off for around T=0.1 for N=10.
2	N=10; it = 500,000; T=[0.01,2] in steps of 0.1		The simulation doesn't align with the expected curve, but follows the same curve just shifted up. This probably means more iterations are needed, so will look at it=1,000,000. We do get back the expected drop off at around 0.1, not the gradual decrease to zero as was the issue above.

3	N=10; it = 1,000,000; T=[0.01,4] in steps of 0.2		The simulation is within an error bar of the expected curve ~75% of the time. This looks quite promising - we still have large error bars for the larger T (will look at next week). But the points follow the expected pattern. Will try again with a higher it=2,000,000
4	N=10; it = 2,000,000; T=[0.01,4] in steps of 0.2		75% of points are within an error bar of the expected curve. We have the usual drop off at low T, with the issue above fixed. Only one point is very far from the expected curve, all the other are within an error bar or just outside of one. I will up the number of iterations to try and close down the errors.
5	N=20; it = 4,000,000; T=[0.01,4] in steps of 0.2		70% of points are within an error bar - anything over ~65% is what we need. Will repeat this again to confirm the results, but am happy. Note: The title of the graph was changed to be more accurate, we take the average squared value, not the square average value.3
6	N=20; it = 4,000,000; T=[0.01,4] in steps of 0.2		Not as good as the above simulation - with no changes. This will happen with a statistical model. Although fewer points are within an error bar of the curve (55%), the range is less. Going to try more iterations.
7	N=20; it = 5,000,000; T=[0.01,4] in steps of 0.2		60% within an error bar, with only one point obviously not following the pattern. The last few graphs have seemed to confrim the simulation is working correctly now for a much smaller number of iterations than before the error in DS was found. I want to focus on the behaviour for low T, and then look at N=30 and possibly N=40,50. Once these are working I will produce plots at low T for multiple N values (hopefully showing that we get more accurate results for larger N) and a large range in T for a few values of N. I may also vary the number of iterations, showing how results become more accurate. **Overall I am finally happy that the code is working correctly** (within an allowed uncertainty due to the statistical nature of the model).
8	N=20; it = 4,000,000; T=[0.01,1] in steps of 0.1		We get a shifted pattern but not on the expected curve. I suspect this is due to N being relatively small - as we saw from the N=1,2,3 cases. I will try a larger number of iterations first then try N=30 at this small range.
9	N=20; it = 5,000,000; T=[0.01,1] in steps of 0.1		Similar to above, will try N=30 to see if we get more accurate results.

10	N=30; it = 6,000,000; T=[0.01,1] in steps of 0.1		Simulation data is closer to the line than for N=20. Will try more iterations.
11	N=30; it = 7,000,000; T=[0.01,1] in steps of 0.1		This has brought the simulated points to a shifted curve of the simulation data. I suspect by trying N=40 we will get closer to the curve.
12	N=40; it = 8,000,000; T=[0.01,1] in steps of 0.1		We have points close to the curve, or on it. Though with some larger errors at higher T. Will try a higher number of iterations.
13	N=40; it = 9,000,000; T=[0.01,1] in steps of 0.1		This has brought the errors down and the points closer to the expected curve. I will try N=50 to see if we can get even closer to the curve.
14	N=50; it = 9,000,000; T=[0.01,1] in steps of 0.1		Points are closer to the curve, though more iterations are needed as the expected pattern is not quite there.
15	N=50; it = 10,000,000; T=[0.01,1] in steps of 0.1		We are closer to the curve, with most points very close or within an error bar of it. We also see the expected pattern, just shifted. Will try a higher number of iterations.
16	N=50; it = 15,000,000; T=[0.01,1] in steps of 0.1		We are closer to the curve than above. Will try with a higher number of iterations to see if we can get ~70% of points in the curve. This also took ~45mins to produce - a lot quicker than before with the changes. Once a low T range follows the expected curve, I will look at larger T. I will produce a full table of 'nice' results once this is all complete.

17	N=50; it = 20,000,000; T=[0.01,1] in steps of 0.1		Points are closer to the curve. Two points were not - the one at T=0 (as expected) and T=0.8. I will try N=60 to see if we can get closer to the curve, but happy with the data.
18	N=60; it = 20,000,000; T=[0.01,1] in steps of 0.1		Results are close to the expected curve. I suspect more iterations would produce a curve closer to the expected line. But will look at a larger range of T for N=50.
19	N=50; it = 20,000,000; T=[0.01,4] in steps of 0.25		Results are close to the expected line with ~65% of points within an error bar of the expected result. I will look at this again, before looking at other results - namely the zero-point energy.
20	N=50; it = 30,000,000; T=[0.01,4] in steps of 0.5		75% of points are within the expected curve, with one obvious anomaly. This confirms to me that everything is working, as we have a good simulation.

Conclusion:
I am now happy with the code, and it is producing good results. I plan on collating some of the graphs together comparing different N values and iteration number values. I also want to look at the zero point energy distribution. This will be done in a different notebook.

Below is the full code:

```
import random
import numpy as np
import matplotlib.pyplot as plt
import time

def updateavg(xarr, S1, N, beta, m, k, p):
    #This fucntion updates a point along a given path. We first define some variables and calculate the change in action between the old and new paths.
    eps = beta/N
    pos = random.randint(0,N-1)
    delta = random.uniform(-1,1)
    xtemp = np.copy(xarr)
    xtemp[pos] = xarr[pos] + delta
    xtemp[N] = xtemp[0]
    DS = eps*((m/2)*((xtemp[pos+1]-xtemp[pos])/eps)**2 + (m/2)*((xtemp[pos]-xtemp[pos-1])/eps)**2 + 1/2*k*(xtemp[pos])**2
            - (m/2)*((xarr[pos+1]-xarr[pos])/eps)**2 - (m/2)*((xarr[pos]-xarr[pos-1])/eps)**2 - 1/2*k*(xarr[pos])**2)
    S2 = S1+DS
    #If this change in action is less than or equal to zero, the update is allowed. The path is updated, as is the action.
    if DS <= 0:
        xarr = np.copy(xtemp)
        S1 = S2
    else:
        #If DS is positive, we set a probability (p) to be the negative exponent of this DS and find a random number between (0,1).
        #If this random number is less than p, the update is allowed. If not, the old path is kept.
        p = np.exp(-DS)
        r = random.uniform(0,1)
        if r < p:
            #If the update is allowed, the new path and new action are stored.
            #If the update isn't, the old path and action are kept.
            xarr = np.copy(xtemp)
            S1 = S2
    #This method returns a path and the action of the path.
    return (xarr, S1)

def MMC(N, beta, m, k, it, p):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    S1 = 0
    #Avgpath list stores the averages of all path generated, the first path has an average of 0.
    avgpath = [0]
    #We iterate over the number supplied-1 (as there are an iteration number of paths, the first being all zeros).
    #We use the updateavg function to update our paths, with the path and action of the path returned being stored for the next iteration.
    #The average value of the path is appended to the avgpath list.
    for i in range(it-1):
        update = updateavg(x, S1, N, beta, m, k, p)
        x = np.copy(update[0])
        S1 = update[1]
        xpow = x**p
        avgpath.append(np.mean(xpow))
    return np.asarray(avgpath)
```

I did notice a slight issue with using p as a variable in the 'updateavg' function. This didn't cause an issue with the method, just some confusion. I removed the parameter from the function as it was not needed. (This was a legacy from an older version) (21/11/2022)

See comment in **reflection** for the correct assessment. (23/11/2022)

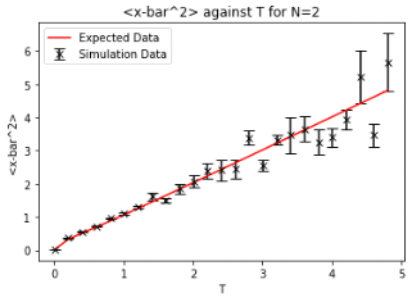
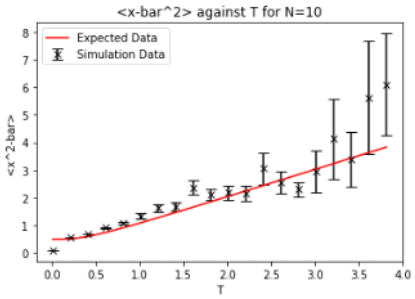
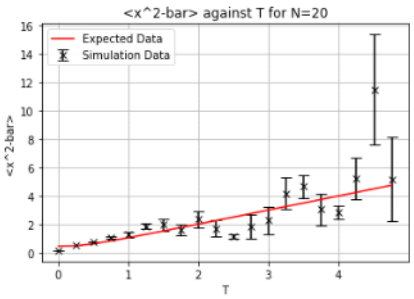
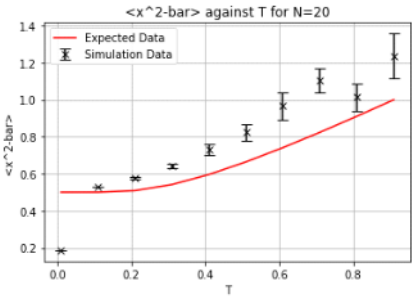
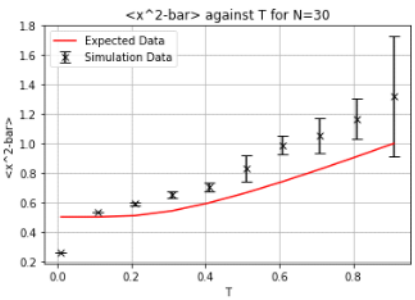
```
def errordata(xarr):
    #Finds the mean and standard error for a given array of points
    sects = np.split(xarr, 10) #Splits array into 10 sections, on which we will calculate means and std deviations
    means = []
    for i in range(8):
        #for each section - except the first - we find the mean of each section
        section = sects[i+1]
        means.append(np.mean(section))
    #calculates the std error for the 9 sections we are interested in
    error = np.std(means)/np.sqrt(8)
    totalmean = np.mean(xarr)
    #returns a tuple of the mean and std error for that set of x values
    return (totalmean, error)


starttime = time.time()
N = 50
m = 1
k = 1
it = int(30e6)
T = np.arange(0.01,4,0.5)
beta = 1/T
means = []
errors = []

for i in range(np.size(beta)):
    paths = (MMC(N, beta[i], m ,k ,it, 2))
    data = errordata(paths)
    means.append(data[0])
    errors.append(data[1])

y = np.asarray(means)
f = (np.exp(1/T)+1)/(2*(np.exp(1/T)-1))
plt.errorbar(T, y, xerr = None, yerr = np.asarray(errors), marker='x', color='black', ecolor='black', linestyle = 'none', capsize = 5, label='Simulation Data' )
plt.ylabel('<x^2-bar>')
plt.xlabel('T')
plt.plot(T, f, color='red', label='Expected Data')
plt.legend()
plt.title('<x^2-bar> against T for N='+str(N))
plt.grid()

print('Execution time: '+str((time.time()-starttime)/60) +' mins')
```

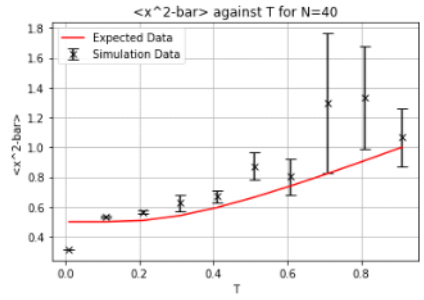
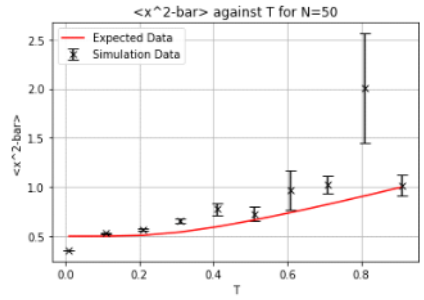
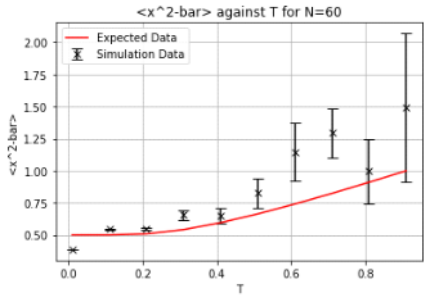
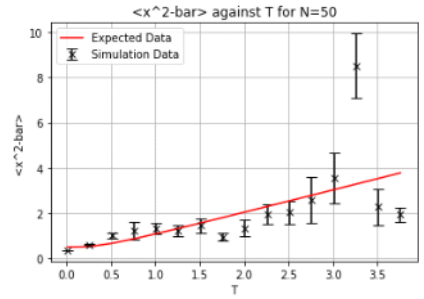
(TW) Final Results for Large N (21/11/2022)

21 November 2022

Final Results for Large N:

As mentioned in the previous page, I wanted to put together some of the more interesting results from my many different tries for large N. I will then look at the zero-point energy position probabilities. This is all for the final code.

Values	Plot	Comments
N=2; it = 50,000; T=[0.01,5] in steps of 0.2		This is not for a large N, but for N=2. This plot shows that the code works for a small N, and gives the classical result as T gets large. Here the expected data has been worked out explicitly (see Week 6/7 to see derivation of the curve), so will tend to zero as T tends to zero (this doesn't happen in the quantum system).
N=10; it = 1,000,000; T=[0.01,4] in steps of 0.2		With roughly ~75% of points within an error bar of the expected curve, this graph shows the method works for larger N. We not the drop off for low T, with the explanation of this found in the other page in week 9.
N=20; it = 4,000,000; T=[0.01,5] in steps of 0.2		Roughly 70% of points are within an error bar of the expected curve. The main issue is with large T results having large error bars. This will be discussed next week - there is a modification to the MMC method that means the expectation value can be found more accurately for larger T but moving a selection of sequential points at once.
N=20; it = 5,000,000; T=[0.01,1] in steps of 0.1		Since the method appeared to be working for large N at large T, I wanted to look at small T and the drop off of the method. We see here that although the N=20 case follows the pattern (but with an offset), something is clearly wrong. This is due to the relatively small value of N - we know the method only approximates the QHO for N tending to infinity.
N=30; it = 7,000,000; T=[0.01,1] in steps of 0.1		Similar to above, but the points are getting shifted close to our expected curve.

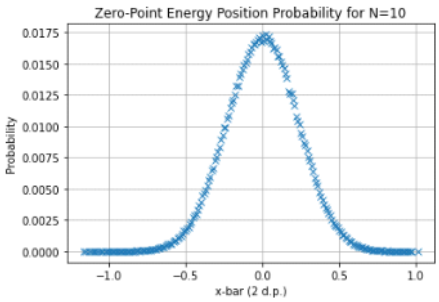
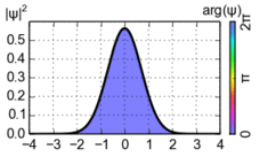
N=40; it = 9,000,000; T=[0.01,1] in steps of 0.1		As above, but getting even closer to the curve. It is also important to note the number of iterations needs to increase - this is so that each point on the path gets a similar number of chances to be chosen to be updated.
N=50; it = 20,000,000; T=[0.01,1] in steps of 0.1		This results is quite good at getting the points close to our expected curve. Clearly by increasing N (and the number of iterations) we get closer to the quantum result - this is as expected.
N=60; it = 20,000,000; T=[0.01,1] in steps of 0.1		Results are close to the curve, but this highlights why we also need to increase the number of iterations. Increasing the number of iterations would produce a mor accurate result - but this takes quite a lot of time as the algorithm is roughly of order n.
N=50; it = 20,000,000; T=[0.01,4] in steps of 0.25		Results are close to the expected line with ~65% of points within an error bar of the expected result. Increasing the number of iterations would help, but this will take time. For smaller N we have showed the system approximates the large T very well, so it doesn't seem to be too important at this stage to worry about that. When we look at changing the efficiency of the method (especially for large T) next week, I plan to come back to this and produce some comparisons.

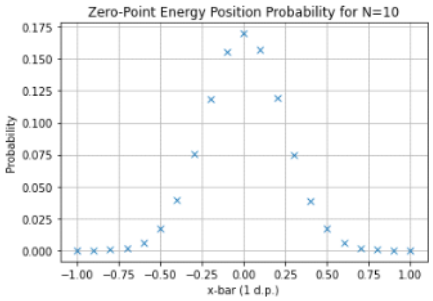
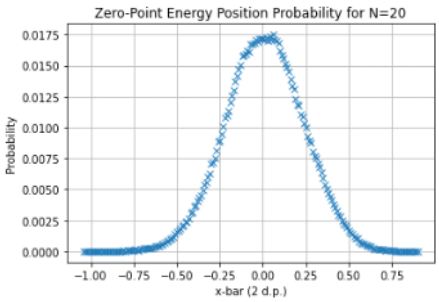
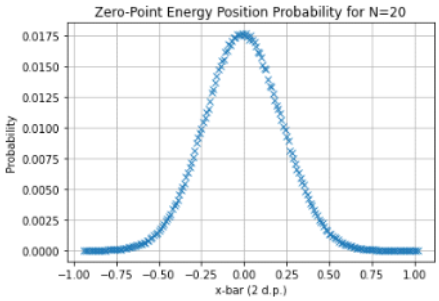
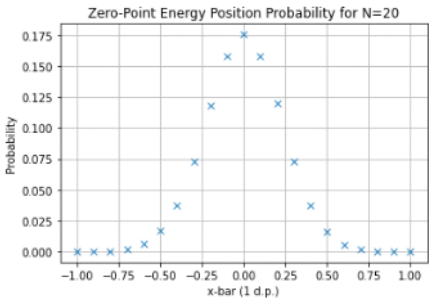
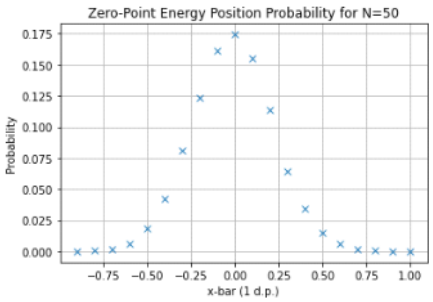
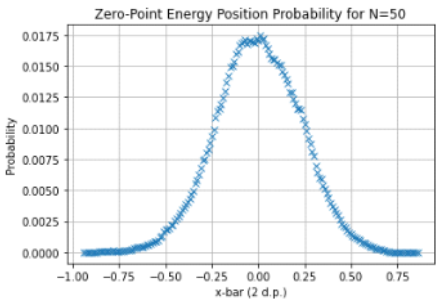
Zero-point energy Position:

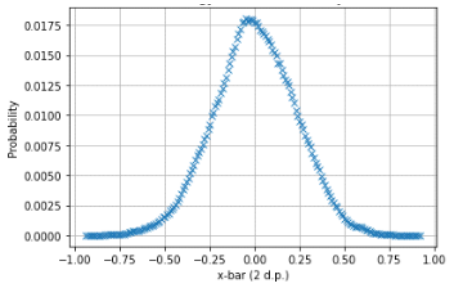
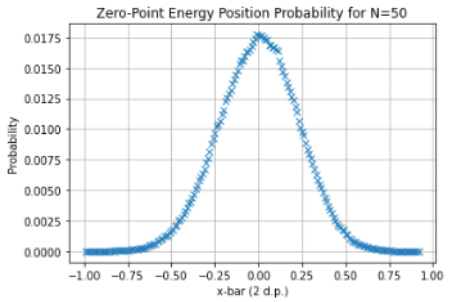
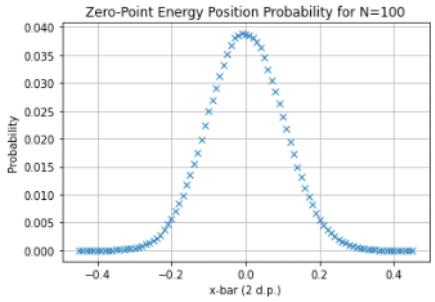
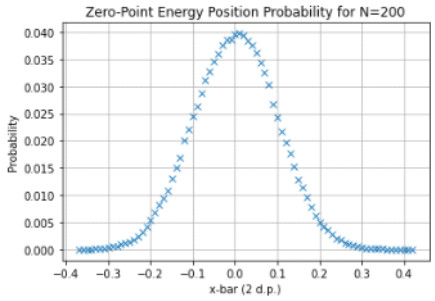
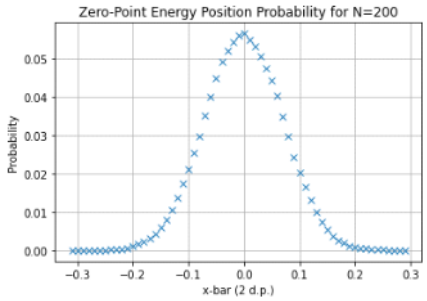
As mentioned in our supervisor meeting, the quantum system does not behave like a classical one at absolute zero. Instead of always finding our particle at x=0, we may also find it outside of this value, though it is most probable to find it here. The result of the probability of finding a particle at x against x should be a Gaussian.

To look into this I used the new MMC (with the power set to 1) method to plot the probability of finding the average position of a path against the average position. Given that the method produces long floats, I rounded the average path position value. I then c ounted the number of times each unique value of x arose, plotting each unique x against the probability (number of times a position aros e / number of iterations). The temperature could not be set to zero - due to division by zero, but also the method breaking down for T<1/N. I chose T=0.05 and look at the distribution for certain N, it and rounding values. The probabilities always sum to 1 - these are proportional to the absolute value squared of the wavefunction. Below are the results:

(m=k=1, T=0.05 with N and iteration number changing, unless otherwise stated.)

Run	Values	Plot	Comments
1	N=10; it = 1,000,000; Rounding: 2DP		This produces a Gaussian as expected. If you compare this result to the graph in the Wikipedia article (https://upload.wikimedia.org/wikipedia/commons/thumb/2/2f/QHO-groundstate-animation-color.gif/220px-QHO-groundstate-animation-color.gif) you do not get the same values. This will be down to the discrete nature of my simulation, as possibly due to proportionality constants. Also of course due to N being relatively small, and limitations on T due to the method. I am not too worried about the values of the probabilities, just the shape of the graph. 

			x $[(\hbar/(m\omega))^{1/2}]$
2	N=10; it = 1,000,000; Rounding: 1DP		Produces a similar result to above, just with fewer plot points due to a more stringent rounding parameter.
3	N=20; it = 1,000,000; Rounding: 2DP		Gaussian as expected with more points, as would be expected. I will look at changing the number of iterations, and the rounding parameter.
4	N=20; it = 2,000,000; Rounding: 2DP		More defined peak as opposed to the plot above. Increasing the number of iterations will produce a more accurate result - as expected.
5	N=20; it = 2,000,000; Rounding: 1DP		Very similar to above, just with a drastic decrease in the number of points plotted - as expected for changing the rounding parameter. I will now look at a large N, starting with N=50 to see if there are any changes. It looks like 2DP is probably the best rounding option.
6	N=50; it = 2,000,000; Rounding: 1DP		Produces a very similar result to above, so I'll stay at N=50 for now. Will change the rounding to 2DP first (I forgot to change this). Then will look at the number of iterations. Will possibly look at very large N (say N=100, so we can look at T=0.01)
7	N=50; it = 2,000,000; Rounding: 2DP		Less well defined peak as compared to N=20. More iterations are needed. I will keep to around 1,000,000 iterations per N.
8	N=50; it = 5,000,000;	Zero-Point Energy Position Probability for N=50	

	Rounding: 2DP		We still get a Gaussian, although slightly skewed. This is probably down to the number of iterations, or just chance. I will look at doing this again.
9	N=50; it = 5,000,000; Rounding: 2DP		Peak now at x=0, so the above skew was just due to the random nature of the MMC method. I am happy that the code produces a good Gaussian at different N, and have investigated the changes - this can be done in the report as well. I will lastly look at N=100.
10	N=100; it = 10,000,000; Rounding: 2DP **T = 0.01**		This produced a really good Gaussian. Also notice how the probability of finding x at zero increased - I suspect a larger N and lower T will produce plots similar to the Wikipedia Article. Since this took only 4 mins to complete, I am going to try N=200 and T=0.01. If the probabilities increase, then this will help confirm my suspicion that the reason for the smaller probabilities is due to N, amongst other things. Though the range of x has decrease - this could be due to T being on the limit of $\varepsilon < 1$. I'm not putting T=0.005 just yet, though will look at that also.
11	N=200; it = 10,000,000; Rounding: 2DP **T = 0.01**		Interestingly, the probabilities did not increase here so they are probably temperature dependent. I am going to try T=0.05 for the last result, as we are still getting Gaussians. Increasing the number of iterations wasn't needed. I will discuss the size of the probabilities at our next meeting - see if we can derive an expected curve for our setup.
12	N=200; it = 10,000,000; Rounding: 2DP **T = 0.005**		We still get a nice Gaussian, with a higher probability of x=0. This look like its due to changing T. There will be some issues with the method breaking down for small T, and other changes that will be causing this I suspect. I will ask about it at the next meeting.

Conclusions:

Overall I am happy with my code for the expectation value of x at large N. There are some slight issues at larger T, but this will be discussed next time, looking at how the method can be changed to make it more efficient for larger T.

The zero-point energy graphs produce Gaussian distributions. I suspect due to model assumptions, constants and N values are the reason they do not match the Wikipedia article - I will bring this up at the next meeting, to see if there is an expected curve to plot against. But the Gaussian is reproduced, so I am happy.

The next steps will be to discuss the zero point energy, but mainly to look at changing the algorithm so that is updates a co ntiguous selection of points on a path - we can investigate what this does to our graphs when compared to the old method. Once this is done, I suspect we may look at other systems outside of the QHO - **but we are very much ahead of schedule, having produced everything we set out to do in the project plan!**

Code to produce the Gaussian Plots:

```
    starttime = time.time()
    N = 200
    m = 1
    k = 1
    it = int(10e6)
    T = 0.005
    beta = 1/T
    DP = 2

    paths = (MMC(N, beta, m ,k ,it, 1))
    paths = np.round(paths,DP)
    x, y = np.unique(paths, return_counts=True)
    y = y/it
    plt.plot(x, y, marker='x', linestyle='none')
    plt.ylabel('Probability')
    plt.xlabel('x-bar ('+str(DP)+' d.p.)')
    plt.title('Zero-Point Energy Position Probability for N='+str(N))
    plt.grid()

    print('Execution time: '+str((time.time()-starttime)/60) +' mins')
```

Post Supervisor Meeting Comments:

Briefly spoke about the zero-point energy graphs. They gave the right shape, so happy with them.

13/12/2022:

These graphs will be recreated later on as not entirely correct. It is better to plot the probability density (ie diving by t he precision of rounding) and as a histogram bar plot. This will be look at in week 12, so see there for the correct plots.

TK

Sunday, November 20, 2022 6:20 PM

Please expand tab for full weekly report

Theory

Sunday, November 20, 2022 6:21 PM

Outline:

- This week we have seen how to improve the efficiency of our simulation:
  - Storing the means of the paths for each iteration, rather than the paths themselves
  - Changing multiple points at each trial, instead of one
    - This should improve the simulation for high T
- We also looked at plotting the zero-point motion and the expected curve
  - Gaussian

Background:

Zero-point motion:

- Zero-point energy
  - Lowest possible energy of a quantum system
  - Consequence of the uncertainty principle of quantum mechanics
- Zero-point motion
  - Consequence of the uncertainty principle of quantum mechanics
    - Can never come completely to rest, because precise values of position and velocity cannot be measured simultaneously

Update multiple points:

- Improves simulation for large T
  - For large T, epsilon becomes small, $\varepsilon = \frac{1}{NT}$
  - Lower action
  - So less likely to accept change
  - However, when updating multiple points, increases probability that an update will be accepted, and therefore less iterations needed

Plan:

- Create code to plot probability of finding particle at point x for zero-point energy
  - This will use the minimum value of temperature for when the simulation holds
    - Calculated in a previous week (Week 7)
- Create code with improved methods of action and means calculation
  - Check code returns valid results

Method: when x_2 is updated, becoming x_2', & x_1 is the point to its left & x_3 to its right

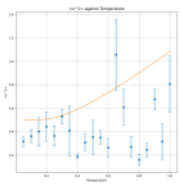
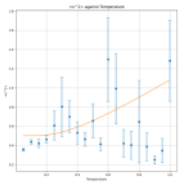
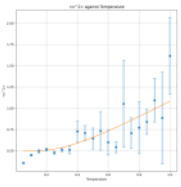
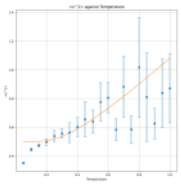
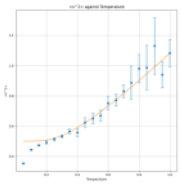
$$\frac{1}{\varepsilon}\Delta S = \frac{m}{2}\left[\left(\frac{x_2'-x_1}{\varepsilon}\right)^2 + \left(\frac{x_3-x_2'}{\varepsilon}\right)^2\right] + \frac{1}{2}kx_2'^2 - \frac{m}{2}\left[\left(\frac{x_2-x_1}{\varepsilon}\right)^2 + \left(\frac{x_3-x_2}{\varepsilon}\right)^2\right] - \frac{1}{2}kx_2^2$$

$$\Rightarrow \Delta S = \frac{m}{2\varepsilon}\left[(x_2'-x_1)^2 + (x_3-x_2')^2 - (x_2-x_1)^2 - (x_3-x_2)^2\right] + \frac{1}{2}k\left(x_2'^2 - x_2^2\right)$$

Improved Means Calculation

Monday, November 21, 2022 11:36 AM

N=10:

Details	Graph	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Previous time taken (ex. plotting) (in week 8)	Comments
N=10 Iterations =1,000		25.0% within error bars	Elapsed time = 0.5324275493621826	Elapsed time = 1.2002291679382324	A lot quicker, over twice as quick; the results look promising also. Therefore, I am happy with these results. Will now try for higher iterations to check reliability of results.
N=10 Iterations =10,000		40.0% within error bars	Elapsed time = 5.214507341384888	Elapsed time = 31.517925262451172	Much faster results, roughly 6 times as fast, results looking better for higher iterations, as expected. High errors for large temperature values.
N=10 Iterations =100,000		60.0% within error bars	Elapsed time = 50.18563461303711		Odd that less points within error bars of expected values. Will need to run this again. However, visually it can clearly be seen
N=10 Iterations =100,000		70.0% within error bars	Elapsed time = 49.800501346588135		Within one standard deviation of expected results.
N=10 Iterations = 1,000,000		70.0% within error bars	Elapsed time = 1396.461566209793		Within one standard deviation of expected results.

Iterations=1,000,000:
N=10: Elapsed time = 1396.461566209793
N=20: Elapsed time = 628.7893409729004
N=50: Elapsed time = 475.8284294605255

Overall, I am incredibly pleased with these results, as they are shown to be as reliable as before, and much more time efficient.

Improved Means Calculation Code

Monday, November 21, 2022 11:37 AM

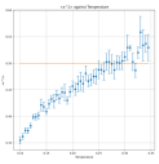
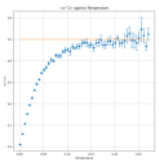
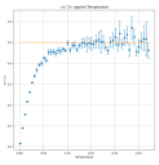
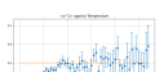
```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""

"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #check speed of program

plt.close('all')

"""values
k = 1
beta = 1
m = 1
N = 1
"""

"""functions"""
def updateX(beta,N,m,k):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([1000])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(1000):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        x_values[-1,point_selected] += random.uniform(-1,1)

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2+(x_values[-1,(point_selected+1)%N]-
x_values[-1,point_selected%N])**2-(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%
N])**2-(x_values[-2,(point_selected+1)%N]-x_values[-2,point_selected%N])**2) + 0.5
*k*epsilon*(x_values[-1,point_selected%N]**2-x_values[-2,point_selected%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
```

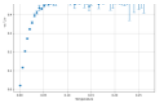
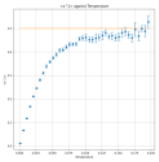
```
            x_values = np.delete(x_values,-2,0)
        else:
            x_values = np.delete(x_values,-1,0) # changes updated point back to original value
      means[i] += np.mean(x_values**2)
   return means

"""averages function"""
def meanError(means):
   mean_split = np.split(means,10)
   mean_split = np.delete(mean_split,0)
   mean_split = np.split(mean_split,9)
   mean_split_means = np.zeros(9)
   for i in range(0,9):
      mean_split_means[i] += np.mean(mean_split[i])
   overall_mean = np.mean(mean_split_means)
   return overall_mean

start_time = time.time()
print(meanError(updateX(1/0.05,10,1,1)))
print(meanError(updateX(1/0.05,10,1,1)))
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))
```

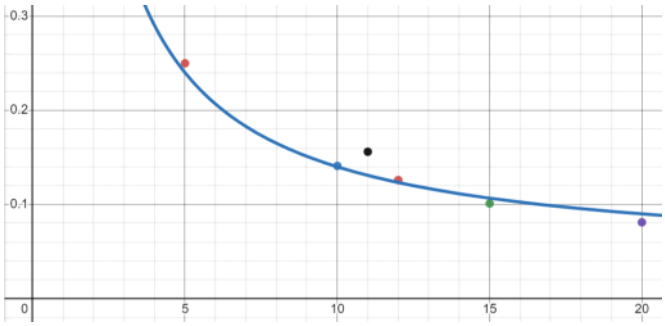
Lowest Temperature Results

Tuesday, November 22, 2022 3:30 PM

Here I am looking at the lowest temperature for which the simulation holds for different N

Details	Graph	Average T in error bars for 0.5	Lowest T in error bars for 0.5	Time Taken	Comments
N=5 It= 100,000 T=(0.1,0.35,0.005)		0.27850000000000001 [0.25 0.26 0.265 0.27 0.275 0.28 0.285 0.29 0.295 0.315]	0.25	Elapsed time = 131.0599505901336 7	Too great a spread of temperature values. Need to shorten this range for next run.
N=10 It= 100,000 T=(0.001,0.275,0.005)		0.2173888888888889 average temperature in error bars for 0.5 [0.141 0.166 0.171 0.186 0.191 0.201 0.206 0.211 0.221 0.226 0.231 0.236 0.241 0.246 0.251 0.256 0.261 0.271]	0.141	Elapsed time = 144.2065057754516 6	From this is can be seen that for sufficiently large N, the average is no longer as important, because a much larger spread of temperature values are within error bars of 0.5. Therefore, from now on will be focusing my attention on lowest temperature values.
N=15 It= 100,000 T=(0.001,0.275,0.005)		0.191 average temperature in error bars for 0.5 [0.101 0.116 0.131 0.136 0.141 0.146 0.151 0.156 0.161 0.166 0.171 0.176 0.191 0.196 0.201 0.206 0.211 0.216 0.221 0.226 0.241 0.251 0.256 0.261 0.266 0.271]	0.101	Elapsed time = 146.8179035186767 6	
N=20 It=		0.18013793103448278	0.081		

100,000 T=(0.001, 0.275,0.005)		average temperature in error bars for 0.5 [0.081 0.086 0.091 0.096 0.101 0.111 0.126 0.131 0.136 0.141 0.151 0.171 0.176 0.181 0.186 0.191 0.196 0.201 0.211 0.221 0.226 0.231 0.236 0.241 0.251 0.256 0.261 0.266 0.271]			
N=11 It= 100,000 T=(0.001, 0.2,0.005)			0.156	Elapsed time = 105.4592955112457 3	
N=12 It= 100,000 T=(0.001, 0.2,0.005)			0.126	Elapsed time = 99.68707251548767	



Follows roughly a 1/N + 0.04 shape

Lowest Temperature Code

Tuesday, November 22, 2022 3:31 PM

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 27 14:56:23 2022

@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #check speed of program

plt.close('all')

k = 1
beta = 1
m = 1
N = 12
epsilon = beta/N
iterations = 100000

"""functions"""
def updateX(beta,N,m,k):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([iterations])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(iterations):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        x_values[-1,point_selected] += random.uniform(-1,1)

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2+(x_values[-1,(point_selected+1)%N]-
x_values[-1,point_selected%N])**2-(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%
N])**2-(x_values[-2,(point_selected+1)%N]-x_values[-2,point_selected%N])**2) + 0.5
*k*epsilon*(x_values[-1,point_selected%N]**2-x_values[-2,point_selected%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
```

```
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
                x_values = np.delete(x_values,-2,0)
            else:
                x_values = np.delete(x_values,-1,0) # changes updated point back to original value
        means[i] += np.mean(x_values**2)
    return means

"""averages function"""
def meanError(means):
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0)
    mean_split = np.split(mean_split,9)
    mean_split_means = np.zeros(9)
    for i in range(0,9):
        mean_split_means[i] += np.mean(mean_split[i])
    std_err = np.std(mean_split_means)/(np.sqrt(len(mean_split_means))-1)
    overall_mean = np.mean(mean_split_means)
    return overall_mean, std_err

"""plot figure"""
fig = plt.figure(figsize=(9,9))

"""plot <x^2> against Temperature"""
means = np.array([])
std_errs = np.array([])
temperature_values = np.arange(0.001,0.2,0.005)
start_time = time.time()
for i in range(0,len(temperature_values)):
    meanErrors = meanError(updateX(1/temperature_values[i],N,m,k))
    means = np.append(means,meanErrors[0])
    std_errs = np.append(std_errs,meanErrors[1])
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))

ax1 = fig.add_subplot(111)
ax1.grid()
ax1.errorbar(temperature_values,means, yerr=[std_errs,std_errs], capsize=5, fmt="o")
ax1.set_xlabel('Temperature')
ax1.set_ylabel('<x^2>')
ax1.title.set_text('<x^2> against Temperature')

"""plot expected curve"""
expected = np.ones(len(temperature_values))/2
ax1.plot(temperature_values,expected)

inErrorBars = 0
valsInErrorBars = np.array([])
for i in range(len(means)):
    if means[i]-std_errs[i]<=expected[i] and means[i]+std_errs[i]>=expected[i]:
        inErrorBars += 1
        valsInErrorBars = np.append(valsInErrorBars,temperature_values[i])
inErrorsPercentage = inErrorBars/len(means)*100
#ax1.text(0,1,str(inErrorsPercentage)+'% within error bars',fontsize=15,backgroundcolor='w')

avInErrorBars = np.mean(valsInErrorBars)
```

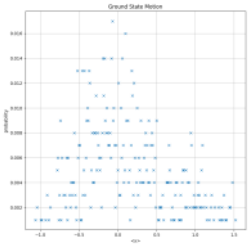
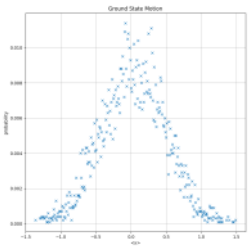
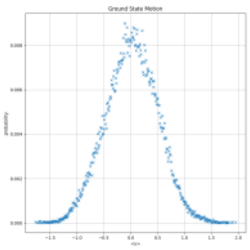
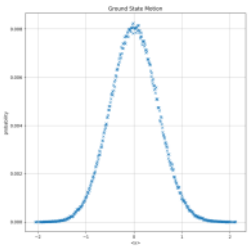
```
#ax5.text(0,1,str(avInErrorBars)+'average temp in error bars for 0.5',fontsize=
15,backgroundcolor='w')
print(str(avInErrorBars)+' average temperature in error bars for 0.5')
print(valsInErrorBars)
```

Zero-Point Probability Results

Tuesday, November 22, 2022 11:37 AM

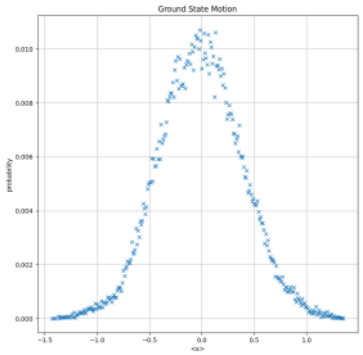
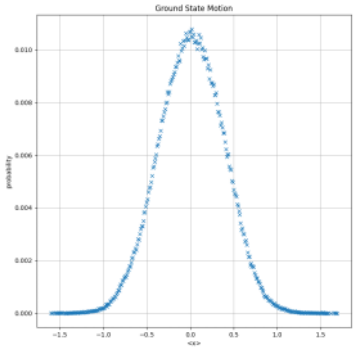
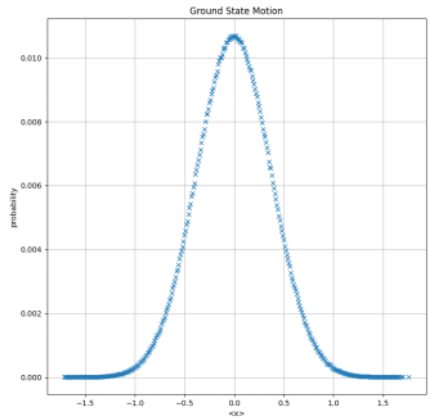
Here the temperature used = **1/N + 0.04**

N=5:

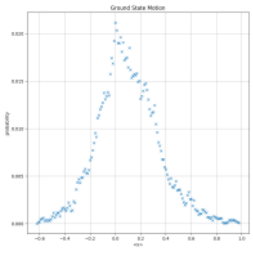
Details	Graph	Time Taken for Calculations (ex. Plotting)	Comments
N=5 Iterations= 1,000		Elapsed time = 0.03112101554870 6055	Much higher iterations needed, but there is a peak seen at around 0, which is promising. Also, do not need to run such low iterations for higher N.
N=5 Iterations= 10,000		Elapsed time = 0.25200295448303 22	Starting to look more like a Gaussian.
N=5 Iterations= 100,000		Elapsed time = 2.29649853706359 86	Much more promising.
N=5 Iterations= 1,000,000		Elapsed time = 25.1831758022308 35	

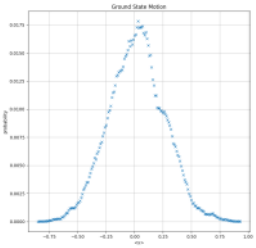
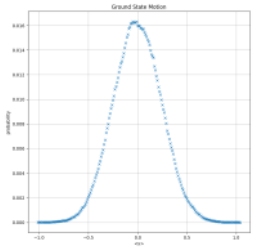
N=10:

Details	Graph	Time Taken for Calculations (ex. Plotting)	Comments

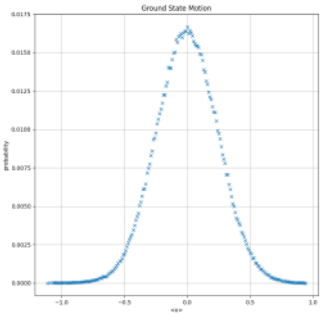
N=10 Iterations =100,000		Elapsed time = 6.386128187179565	
N=10 Iterations = 1,000,000		Elapsed time = 24.504896640777588	Nice Gaussian plot produced. Higher probability for <x> for N= 10, rather than N=5. Implying, that the higher the value of N, the higher the probability that <x>=0.
N=10 Iterations = 10,000,000		Elapsed time = 659.6262724399567	

N=50:

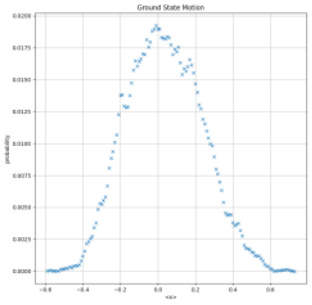
Details	Graph	Time Taken for Calculations (ex. Plotting)	Comments
N=50 Iterations= 100,000		Elapsed time = 6.431970834732056	Does not fit Gaussian very well at all. This shows how at higher N, a larger number of iterations is needed to achieve reliable results.

N=50 Iterations= 1,000,000		Elapsed time = 64.2285552024841 3	Still not a good Gaussian, need to run for a lot higher iterations. This is possibly due to N being so large.
N=50 Iterations= 10,000,000		Elapsed time = 666.951281070709 2	Much nicer Gaussian plot. Under further investigation as detailed below, the probability shape is temperature dependent, from the fact that the below plot uses N=10 with the same temperature. However, increasing N makes the values more reliable.

N=10

Details	Graph	Time Taken for Calculations (ex. Plotting)	Comments
N=10 Iterations= 1,000,000 Temperature =0.06		Elapsed time = 64.54838442802429	Same rough shape as for N=50, proving temperature dependence.

N=200

Details	Graph	Time Taken for Calculations (ex. Plotting)	Comments
N=200 Iterations= 10,000,000		Elapsed time = 666.2829310894012	Need to use a higher number of iterations, in order to get a plot that better fits a Gaussian

Therefore, using my code the higher the N, the lower the temperature used, and therefore the closer the final graph is to the true zero-point motion.

Zero-Point Probability Code

Tuesday, November 22, 2022 11:37 AM

```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time

plt.close('all')

"""values"""
k = 1
m = 1
N = 200

lowest_temp = 1/N + 0.04

iterations = 10000000

"""functions"""
def updateX(beta,N,m,k):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([iterations])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(iterations):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        x_values[-1,point_selected] += random.uniform(-1,1)

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2+(x_values[-1,(point_selected+1)%N]-
x_values[-1,point_selected%N])**2-(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%
N])**2-(x_values[-2,(point_selected+1)%N]-x_values[-2,point_selected%N])**2) + 0.5
*k*epsilon*(x_values[-1,point_selected%N]**2-x_values[-2,point_selected%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
```

```
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
                x_values = np.delete(x_values,-2,0)
            else:
                x_values = np.delete(x_values,-1,0) # changes updated point back to original value
        means[i] += np.mean(x_values)
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0)
    return means

"""averages function"""
def meanError(means):
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0)
    mean_split = np.split(mean_split,9)
    mean_split_means = np.zeros(9)
    for i in range(0,9):
        mean_split_means[i] += np.mean(mean_split[i])
    overall_mean = np.mean(mean_split_means)
    return overall_mean

start_time = time.time()
positions = np.round(updateX(1/lowest_temp,N,m,k),2)
position, count = np.unique(positions, return_counts=True)
probability = count/iterations
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))


"""plot figure"""
fig = plt.figure(figsize=(9,9))

ax1 = fig.add_subplot(111)
ax1.set_xlabel('<x>')
ax1.set_ylabel('probability')
ax1.set_title('Ground State Motion')
ax1.grid()
ax1.plot(position,probability,marker='x',linestyle='none')
```

Week 10

Supervisor Meeting: 23/11/2022

23 November 2022

Outline:

- Briefly spoke about our results from last week, and that we have finished the original goal of the project.
- Spoke about the next steps: we will look at changing the Monte Carlo method to update a contiguous section of points along a path, which should reduce errors for larger T. Will look at N=20.
- Spoke about the Zero-point energy.

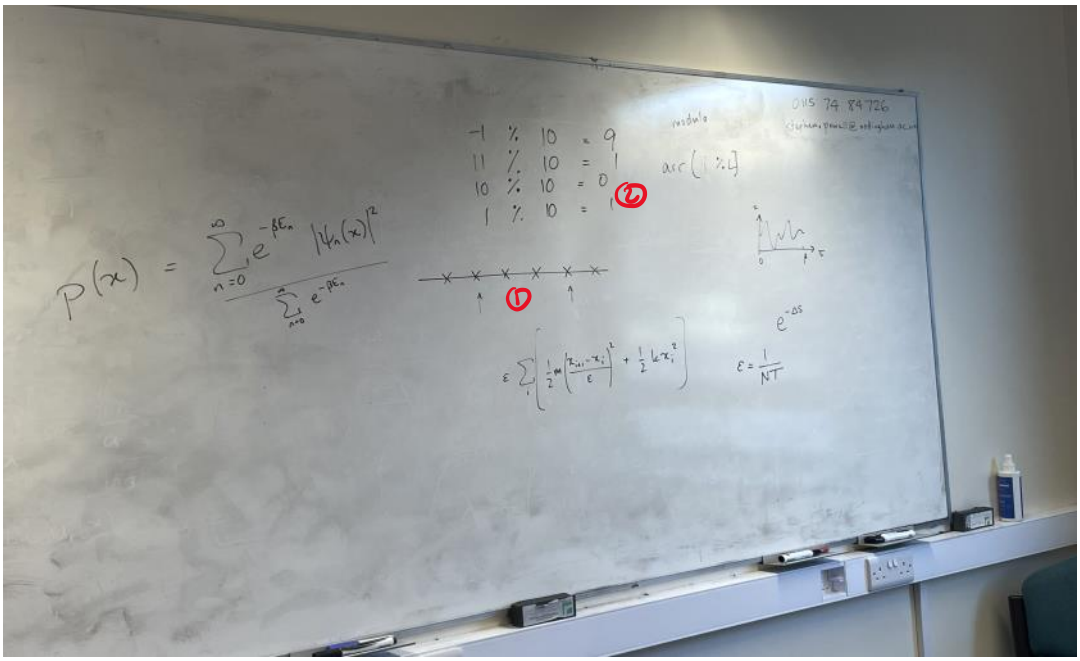
Tasks:

- Create a way of showing the variation of points along a path for low T and high T - we expect higher T paths to be flatter.
- Create the new method for calcuating the change in action by selecting multiple points, compare this against the original method.

Distribution of Tasks:

- Both would produce graphs and data using our codes to confirm everything was working.
- We would then compare our results to see if any issues arose.

Project Details:



1) We spoke about the new Monte Carlo method, a point will be chosen at random before, with a uniformly generated random number between [1,5] - this will be the number of points that will be updated, all be consecutive. The update will not change the kinetic energy term linking the points inside the chain that is moved, only the potential terms of the updated points and the end kinetic terms. This should mean more updates occur for higher T. Having a longer chain though will increase the likelihood that a point is rejected - so a balance will need to be found. It may not be beneficial at lower T to move a collection of points - this will have to be investigated, and possibly mitigated against.
2) We spoke about if the point chosen at random is towards the end of the chain, and thus has consecutive points outside of the path. We use the fact that the path loops back on itself; eg if point 19 is chosen then points 20,21,22 are equivalent to points 0,1,2. Modular arithmetic will help here - distinguished in python by the '%' operator.

A few comments:

The method should work at higher T, due to having a smaller epsilon meaning a larger kinetic energy term reducing the likelihood of an update being allowed. So by reducing the impact of the kinetic energy term, we should reduce errors at higher T. This should mean that a lower number of iterations are needed to get more accurate results. To confirm this, we first want to produce a way of looking at how the paths look at high and low T. We suspect that the paths for high T

are flatter, while at lower T they are more oscillatory.

Once this is done, then we can look at the new algorithm and compare its results to the old one. The algorithm will work as follows:

1. Choose a random point along the path as before.
2. Choose a random number between [1,5] (although this may need to be changed if too many points are rejected to a smaller range). This will be the number of points in the 'chain' that will be updated.
3. Move each point in this 'chain' by the delta value [-1,1] and calculate the change in action between this new path and the old one. The change in action will only consider the kinetic terms of the end points and the potential terms of each point along the chain. Remembering to account for the old kinetic terms and potentials!
4. This change in action is then treated as it was for the single case, with the update being accepted if DS<0 or the exponential of -DS is greater than a random number between (0,1).

(TW) Modifying the MMC method (23-26/11/2022)

23 November 2022

Modifying the MMC method: (16/11/2022)

As discussed in the entry from the supervisor meeting, I will be changing the MMC method to hopefully make it more efficient for larger T - this has been the main issue with the code. Although it is close to the expected result for large T, it takes a large number of iterations to get there. This change should mean fewer iterations are needed. But first I need to produce a way of looking at the shape of paths at high and low T.

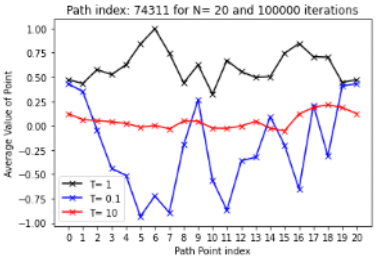
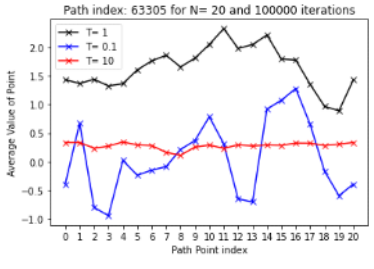
Shape of paths for High and Low T:
I plan on producing a plot for N=20, constant at their usual values for a number iterations at high and low T. I will choose a random point along the path for each temperature and plot each point along that path - the same random path number (path index) is used for all temperatures. This will allow me to compare how the paths look at different temperatures.
This will mean I will have to slightly modify the MMC method to return the full paths rather than the averages. This will be a temporary change.

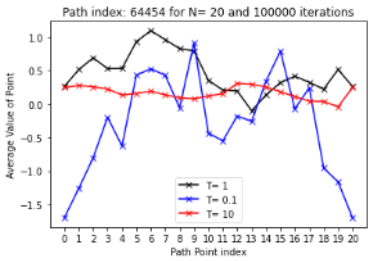
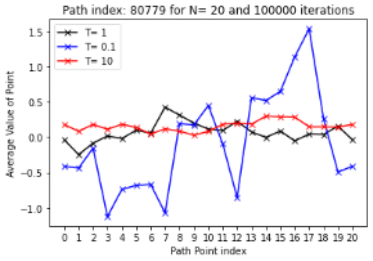
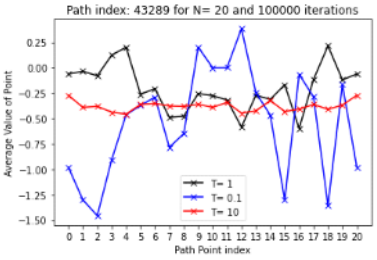
To do this I renamed the 'updateavg' method to **'updatex'**, the 'MMC' method to **'avgMMC'** (This returns the average values of the path raised to a given power), and created a new method **'pathsMMC'** which is the same as the 'avgMMC' method but returns an array of the full path:

```
def updatex(xarr, S1, N, beta, m, k):
    #This fucntion updates a point along a given path. We first define some variables and calculate the change in action between the old and new paths.
    eps = beta/N
    pos = random.randint(0,N-1)
    delta = random.uniform(-1,1)
    xtemp = np.copy(xarr)
    xtemp[pos] = xarr[pos] + delta
    xtemp[N] = xtemp[0]
    DS = eps*((m/2)*((xtemp[pos+1]-xtemp[pos])/eps)**2 + (m/2)*((xtemp[pos]-xtemp[pos-1])/eps)**2 + 1/2*k*(xtemp[pos])**2
              - (m/2)*((xarr[pos+1]-xarr[pos])/eps)**2 - (m/2)*((xarr[pos]-xarr[pos-1])/eps)**2 - 1/2*k*(xarr[pos])**2)
    S2 = S1+DS
    #If this change in action is less than or equal to zero, the update is allowed. The path is updated, as is the action.
    if DS <= 0:
        xarr = np.copy(xtemp)
        S1 = S2
    else:
        #If DS is positive, we set a probability (p) to be the negative exponent of this DS and find a random number between (0,1).
        #If this random number is less than p, the update is allowed. If not, the old path is kept.
        p = np.exp(-DS)
        r = random.uniform(0,1)
        if r < p:
            #If the update is allowed, the new path and new action are stored.
            #If the update isn't, the old path and action are kept.
            xarr = np.copy(xtemp)
            S1 = S2
    #This method returns a path and the action of the path.
    return (xarr, S1)

def avgMMC(N, beta, m, k, it, p):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    S1 = 0
    #Avgpath list stores the averages of all path generated, the first path has an average of 0.
    avgpath = [0]
    #We iterate over the number supplied-1 (as there are an iteration number of paths, the first being all zeros).
    #We use the updateavg function to update our paths, with the path and action of the path returned being stored for the next iteration.
    #The average value of the path is appended to the avgpath list.
    for i in range(it-1):
        update = updatex(x, S1, N, beta, m, k)
        x = np.copy(update[0])
        S1 = update[1]
        xpow = x**p
        avgpath.append(np.mean(xpow))
    return np.asarray(avgpath)

def pathsMMC(N, beta, m, k, it):
    #This is the same as the avgMMC method but returns the full paths back
    x = np.zeros(N+1)
    S1 = 0
    paths = [x]
    for i in range(it-1):
        update = updatex(x, S1, N, beta, m, k)
        x = np.copy(update[0])
        S1 = update[1]
        paths.append(x)
    return np.asarray(paths)

```

Results:
I produced three different curves at temperatures, T, = 0.1 (valid as for N=20, lowest 0.05), 1, 10. I chose a random number to find the paths I was going to compare - I did not allow it to choose from the first 10% of paths as to allow for a large number of updates to occur, to allow for most points to be updated at least once. The 'path index' is the random path chosen from the set of all paths for a given T.

I did this a few times, with the results as follows:

Run	Plot
1	
2	

3	
4	
5	

Conclusion:
The plots showed the expected result, that a higher T value means a path is flatter. This means that moving a 'chain' of points in one go when updating a path will benefit high T, producing more accurate results as the kinetic terms do not have such a large impact on the change in action. This can be seen by looking at the ε term in the action and noticing for a large T, $\varepsilon = \frac{1}{NT}$ is smaller for a larger T, meaning the kinetic term is larger for a higher T as $\sim\varepsilon^{-1}$. So by focusing more on the potential terms (by moving a large chain) the update should be accepted more frequently than just a single point being changed. This ofcouse will need to be investigated. This may also mean that the single point update is more benifical for lower T - this will have to be investigated also, and somehow mitigated against if it causes an issue.

Changing the Monte Carlo Method (23-25/11/2022)**:**

Since the above results seems to indicate that this method will help for larger T, I will now create a new updated MMC method, and updated 'updatex' method, to change multiple paths at once. I will create a new function to do to, as to allow for comparison to the old method. The algorithm is outlined in the 'Supervisor Meeting' notes from this week.

The new functions are modified versions of the old 'avgMMC' and 'updatex' functions.

The code to do this is as follows:

```
def modupdatex(xarr, S1, N, beta, m, k):
    #This fucntion updates a selection point along a given path.
    #We first define some variables and calculate the change in action between the old and new paths.
    #Similar to the regular updatex
    eps = beta/N
    pos = random.randint(0,N-1)
    #l+1 is the number of points being updated, l=0 corresponds to the original point, so l+1=0 means 1 point being updated.
    l = random.randint(0,4)
    delta = random.uniform(-1,1)
    xtemp = np.copy(xarr)
    #We first add delta to each point in the selection.
    #The modulo function (%) is used incase the point chosen is towards the end of the path, so we loop back to the start.
    for i in range(l+1):
        xtemp[(pos+i)%N] = xarr[(pos+i)%N] + delta

    xtemp[N] = xtemp[0]
    #We calculate the change in action of the new path compared to the old one.
    DS = eps*((m/2)*((xtemp[pos]-xtemp[pos-1])/eps)**2 - (m/2)*((xarr[pos]-xarr[pos-1])/eps)**2 + (m/2)*((xtemp[(pos+1+l)%N]-xtemp[(pos+l)%N])/eps)**2
              - (m/2)*((xarr[(pos+1+l)%N]-xarr[(pos+l)%N])/eps)**2)
    for i in range(l+1):
        DS += eps*(1/2*k*(xtemp[(pos+i)%N])**2 - 1/2*k*(xarr[(pos+i)%N])**2)
    S2 = S1 + DS
    #If this change in action is less than or equal to zero, the update is allowed. The path is updated, as is the action.
    if DS <= 0:
        xarr = np.copy(xtemp)
        S1 = S2
    else:
        #If DS is positive, we set a probability (p) to be the negative exponent of this DS and find a random number between (0,1).
        #If this random number is less than p, the update is allowed. If not, the old path is kept.
        p = np.exp(-DS)
        r = random.uniform(0,1)
        if r < p:
            #If the update is allowed, the new path and new action are stored.
            #If the update isn't, the old path and action are kept.
            xarr = np.copy(xtemp)
            S1 = S2
    #This method returns a path and the action of the path.
    return (xarr, S1)

def modavgMMC(N, beta, m, k, it, power):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    S1 = 0
    #Avgpath list stores the averages of all path generated, the first path has an average of 0.
    avgpath = [0]
    #We iterate over the number supplied-1 (as there are an iteration number of paths, the first being all zeros).
    #We use the updateavg function to update our paths, with the path and action of the path returned being stored for the next iteration.
    #The average value of the path is appended to the avgpath list.
    for i in range(it-1):
        #This is the modified function, so uses the modified updatex method.
        update = modupdatex(x, S1, N, beta, m, k)
        x = np.copy(update[0])
        S1 = update[1]
        xpow = x**power
        avgpath.append(np.mean(xpow))
    return np.asarray(avgpath)
```

These new methods produced the following results:

I did this for N=20. The maximum number of points in an updated chain is l+1 - which is 5 unless stated otherwise.
m=k=1 as before.

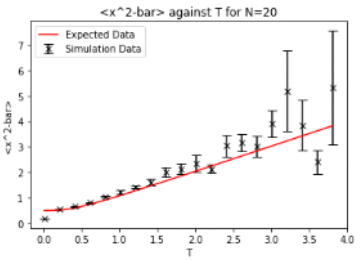
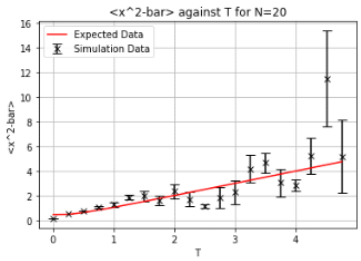
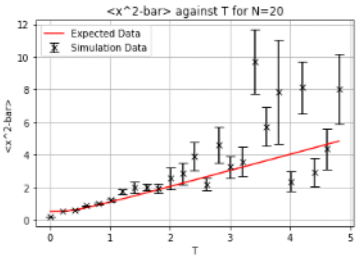
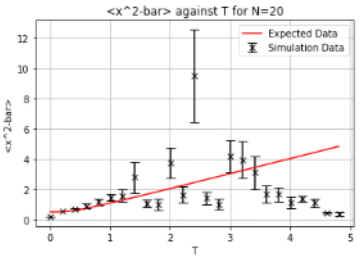
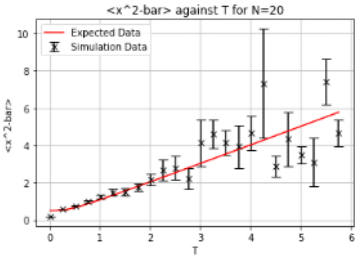
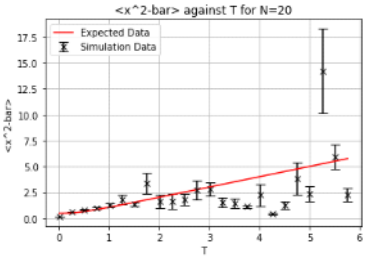
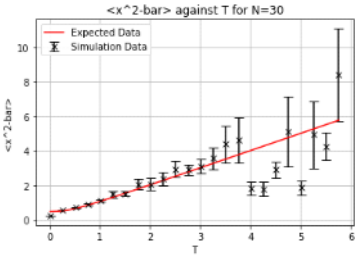
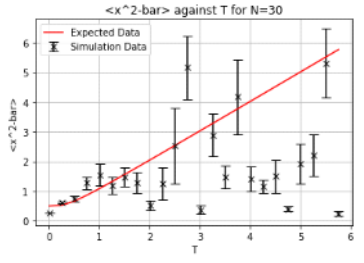
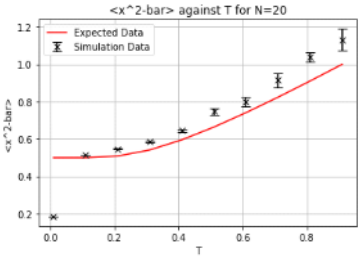
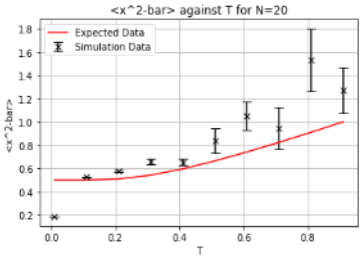
(*, see below): On this run I changed how l(max) was found. Over the tests, and the idea that we don't want l(max) to be too large, an l(max)+1 ~ 25% of the path length worked. I changed the code to this, meaning I didn't need to keep changing it manually for different N. **l(max) is the maximum number of additional points in a chain**. For some N, so we want l(max)+1 = N/4 i.e. l(max) = N/4-1. This may not be an integer, and we need to ensure $l_{max} \geq 0$. So I used the ceil function to round up l(max)-1 - this meant for N=4,3,2,1 l(max)=0, and for N>4, l(max)>0.

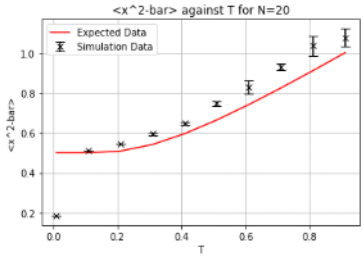
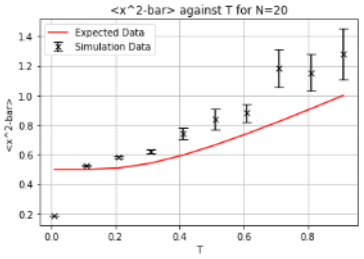
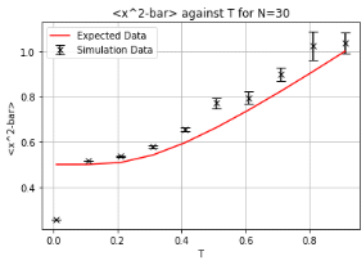
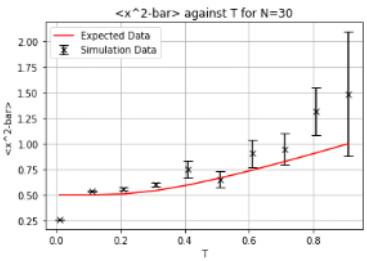
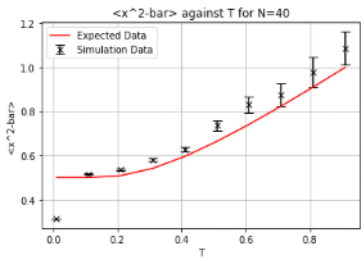
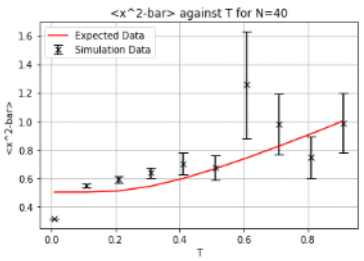
E.g.: N=1,2,3,4: l(max) = 0 i.e. no additional points changed.
N=5,6,7,8: l(max) = 1 i.e. one additional point possibly changed.
N=9,10,11,12: l(max) = 2 i.e. two additional points possibly changed.
etc.

Code:

```
92      #l+1 is the number of points being updated, l=0 corresponds to the original point, so l+1=0 means 1 point being updated.
93      q = np.ceil(N/4)- 1
94      l = random.randint(0,q)
```

q is thus the maximum number of additional points changed and q+1 is the maximum number of points changed and is roughly 25% of N, though could be slightly larger if N/4 is not an integer.

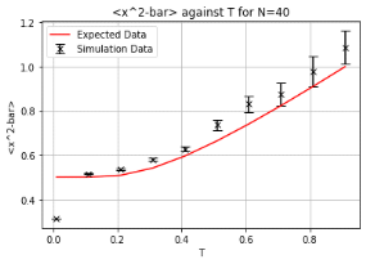
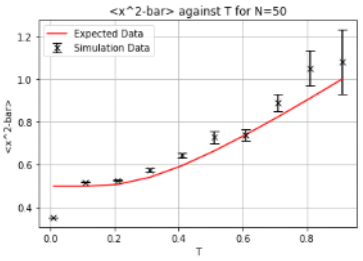
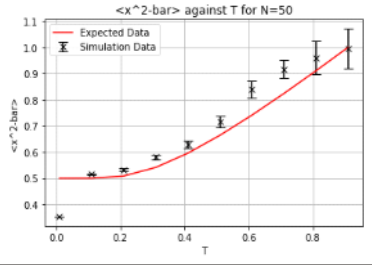
Run	Values	New Code Plot	Old Code Plot	Comments
1	N=20; it = 4,000,000; T=[0.01,5] in steps of 0.2 Max Number of points changed: l(max)+1=5			Both plots were produced with the same parameters (the old code plot went up to T=5 but otherwise the same). We see not only were more of the points closer to the expected curve (at all T) in the new code, but the error bars were smaller. In the old plot: ~70% of points were within an error bar of the expected curve. In the new plot: ~70% again, but those not on the curve were a lot closer. I want to see what happens if we reduce the number of iterations to on both plots - hopefully the new code is still ~65% or more within an error bar of the expected curve, meaning fewer iterations are needed to produce good results.
2	N=20; it = 1,000,000; T=[0.01,5] in steps of 0.2 Max Number of points changed: l(max)+1=5			A clear improvement using the new method. Old plot: 32% within an error bar of expected results. New plot: 56% within an error bar of expected results. New code still not in the 65% range, but much closer than the old version. Points as also closer to the expected results, even if not within an error bar of the expected line. We see more updates at higher T using the new method - this is what we wanted. The new method is more efficient than the old one. I am going to look 2,000,000 iterations for both, with a slightly larger range in T. I then might look at higher N - and will have to change l to match. The old code was slightly quicker: 4.2 mins (~27%) but produced much worse results. This will just be due to more calculations being doing.
3	N=20; it = 2,000,000; T=[0.01,6] in steps of 0.25 Max Number of points changed: l(max)+1=5	 28.8 mins	 23.3 mins (18% quicker)	Again the new code is much better than the old one. More points are on the expected line, and those that aren't are much closer. Old plot: 50% within an error bar of expected results. New plot: 67% within an error bar of expected results. (what we want) It is clear that the new code produces better results for the same parameters as the old - to get the same accuracy, the new code can be run for fewer iterations - so is more efficient. I will now look at N=30
4	N=30; it = 3,000,000; T=[0.01,6] in steps of 0.25 Max Number of points changed: l(max)+1=7	 48 mins	 34 mins (29% quicker)	Again we see the new code is much more accurate. Even those points not on the expected curve are much closer. Old plot: 33% within an error bar of expected results. New plot: 79% within an error bar of expected results. The new code produces more accurate results for the **same** number of iterations as compared to the old code. This means it can produce the same accuracy as the old one for much fewer iterations. In this case the old code produced a plot not at all near the ~65% of points within an error bar of the expected curve to be useful. While the new code produced one with 79% of points within an error bar of the expected data - meaning this plot is useful. For the old code a much higher number of iterations (probably at least double, meaning over an hour of run time) would be needed - taking more time overall. Although the old code is quicker for a like-for-like set-up, it is less accurate and will take **more** time to produce the same accuracy. I am now going to look at N=20 again but for a low T seeing if the new code affects the data's accuracy.
5 **(*)**	N=20; it = 2,000,000; T=[0.01,1] in steps of 0.1 With the code change: l(max)+1 = 5	 13 mins	 10 mins (30% quicker)	Both plots do not follow the expected curve, but are just shifted versions of this. We see that this shifted pattern is neater for the new code. I will try with more iterations. Distance of points to expected curve is smaller in the new code - I believe this offset is due to N being relatively small. Again the newer code is slightly slower, but much more accurate. We have smaller error bars and a smaller range of values.

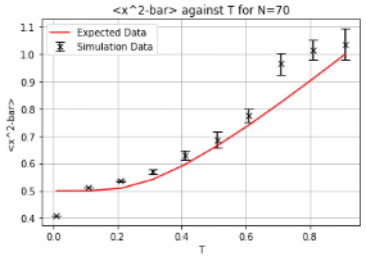
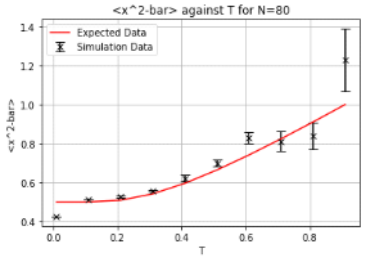
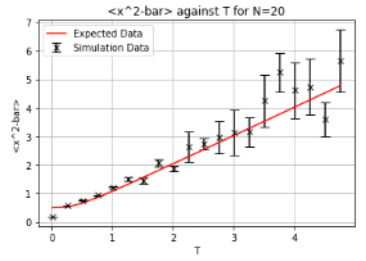
6	N=20; it = 3,000,000; T=[0.01,1] in steps of 0.1 With the code change: l(max)+1 = 5	 19 mins	 14 mins (26% quicker)	Similar to above. Will try with N=30. The new code seems to be better for low T as well - so any fears about it having a negative effect on low T seem to have been dispelled. I shall look at a fewer more low T runs before passing judgement though.
7	N=30; it = 3,000,000; T=[0.01,1] in steps of 0.1 With the code change: l(max)+1 = 8	 22 mins	 14 mins (36% quicker)	Similar to above, though points closer to the expected curve for both cases. The new code is closer still than the older one - though not too much difference. Will next look at N=40 for it=4,000,000
8	N=40; it = 4,000,000; T=[0.01,1] in steps of 0.1 With the code change: l(max)+1 = 11	 32 mins	 20 mins (38% quicker)	Similar comments to above. The new code is more accurate than the old code. The offset of the pattern is due to the N value - we can see this is getting closer to the expected value for an increase in N.

Conclusion: (25/11/2022)

The above results show that the modified method is more efficient than the old one - leading to more accurate results more for the 'like-for-like' conditions. Given this, I now want to produce some good results for low T (meaning a high N), and for a large range of T - trying to get the vast majority of points on the expected curve or very close to it. This of course will use the new **modified MMC model.**

QHO results using the modified MMC model: (25-26/11/2022)

Run	Values	New Code Plot	Comments
1	N=40; it = 4,000,000; T=[0.01,1] in steps of 0.1 Max Number of points changed: l(max)+1=11		Simulation data getting closer to the expected curve. Will try for a higher N.
2	N=50; it = 5,000,000; T=[0.01,1] in steps of 0.1 Max Number of points changed: l(max)+1=13		Better than above. Will try for more iterations.
3	N=50; it = 6,000,000; T=[0.01,1] in steps of 0.1 Max Number of points changed: l(max)+1=13		Slightly better than above - error bars reduced. But N is the more dominant factor - so will increase it again. Will keep to 1,000,000 iterations per point.

5	N=70; it = 7,000,000; T=[0.01,1] in steps of 0.1 Max Number of points changed: l(max)+1=18		Better than above, will try N=80.
6	N=80; it = 8,000,000; T=[0.01,1] in steps of 0.1 Max Number of points changed: l(max)+1=20		Similar comments to above. Could keep increasing N, but data is already quite good; 70% of points are on the expected curve (including the 1st point which we know will not follow curve due to limitations of the model). I am going to look at a large T range now.
7	N=20; it = 4,000,000; T=[0.01,5] in steps of 0.25 Max Number of points changed: l(max)+1=5		Very good results: 85% of points within an error bar of the expected curve, and those that aren't are still very close (ignoring the first point for known reasons).

Conclusion:

Above results are very good and show the model is working - with the new model working much better than the older version. Given this is the case, and that there is no issue at low T with the modified method, I am happy with the code. Will discuss at our next meeting the next steps - if we can look at different potentials, or if there are some other optimisation techniques to use.

TK

Saturday, November 26, 2022 6:42 PM

Please expand tab for full weekly report

Theory

Saturday, November 26, 2022 6:42 PM

Outline:

- Should use % modulo operation to loop around the path, instead of duplicating path
- Probability of finding particle in certain position
- Updating multiple points

Background:

$$p(x) = \frac{\sum_{n=0}^{\infty} e^{-\beta E_n} |\psi_n(x)|^2}{\sum_{n=0}^{\infty} e^{-\beta E_n}}$$

Plan:

- Use % operation in code to loop around path
- Update main code, so that it can update up to 5 points at a time
  - Check results are 68.2% within error bars
- Study probability outlined above

Method:

Select up to 5 points: random.randint (0,5)
Starting at previously randomly selected point

Select up to 5 points: random.randint(0,5) : n_update
Starting at previously randomly selected point, using % so that it can wrap around the array
Update all these points
Change in action calculated as:

For one point: $\Delta S = \frac{m}{2\epsilon}\left[(x_2' - x_1')^2 + (x_3 - x_2')^2 - (x_2 - x_1)^2 - (x_3 - x_2)^2\right] + \frac{1}{2}k\left(x_2'^2 - x_2^2\right)$

For multiple, where randomly selected point = x_R:

Potential term: $\Delta S_{pot} = \frac{1}{2}k\left[\left(x_R'^2 + x_{R+1}'^2 + \cdots x_{R+n\_update}'^2\right) - \left(x_R'^2 + x_{R+1}'^2 + \cdots x_{R+n\_update}'^2\right)\right]$

Kinetic term: $\Delta S_{kin} = \frac{m}{2\epsilon}\left[\left(x_R' - x_{R-1}\right)^2 + \left(x_{R+n\_update+1} - x_{R+n\_update}'\right)^2 - \left(x_R - x_{R-1}\right)^2 + \left(x_{R+n\_update+1} - x_{R+n\_update}\right)^2\right]$

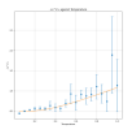
Total: $\Delta S_{tot} = \Delta S_{kin} + \Delta S_{pot} = \frac{m}{2\epsilon}\left[\left(x_R' - x_{R-1}\right)^2 + \left(x_{R+n\_update+1} - x_{R+n\_update}'\right)^2 - \left(x_R - x_{R-1}\right)^2 + \left(x_{R+n\_update+1} - x_{R+n\_update}\right)^2\right] + \frac{1}{2}k\left[\left(x_R'^2 + x_{R+1}'^2 + \cdots x_{R+n\_update}'^2\right) - \left(x_R'^2 + x_{R+1}'^2 + \cdots x_{R+n\_update}'^2\right)\right]$

Now need to update code using this structure

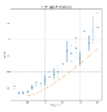
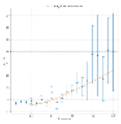
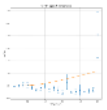
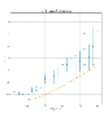
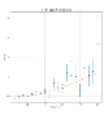
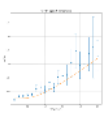
Results

Sunday, November 27, 2022 11:38 AM

Ensure code returns expected results:

Details	Maximum Number of Points Updated	Graph	Average Standard Error	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=5 It= 100,000 T=(0.05, 1,0.05)	1					
N=10 It= 100,000 T=(0.05, 1,0.05)	1			70.0% within error bars	Elapsed time = 187.5544142 7230835	Matches previously attained results using new code.

Now that I am certain the code is working correctly when only updating one point at a time I will try updating up to 5 points at a time, with N=20

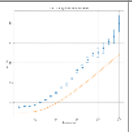
Details	Maximum Number of Points Updated	Graph	Average Standard Error	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=20 It= 100,000 T=(0.05, 1,0.05)	5				Elapsed time = 201.681916 47529602	Higher than expected potentially due to not enough iterations. I will test this now.
N=20 It= 100,000 T=(0.05, 1,0.05)	5		0.205031 66917775 023	70% within error bars	Elapsed time = 90.3822753 4294128	Interesting how for small temperatures, the values are not as reliable as they were in the previous code. However, it is a much more promising set of results than the one above.
N=30 It= 100,000 T=(0.05, 1,0.05)	5		0.138434 89213813 562	25% within error bars	Elapsed time = 219.529525 0415802	Appears as though for large N, a much larger number of iterations is needed to be made. Therefore, I will now test for lower N, and the same number of iterations.
N=10 It= 100,000 T=(0.05, 1,0.05)	5		0.116957 40941559 16	0% within error bars	Elapsed time = 211.363353 0139923	Again higher.
N=10 It= 100,000 T=(0.05, 1,0.05)	1		0.166960 52770542 497	55.0% within error bars	Elapsed time = 218.635961 77101135	Fits expected, same as previous code, which is promising. Strange how for increasing the number of points that are allowed to be updated, shifts the plot upwards. Will investigate this further.
N=10 It= 100,000 T=(0.05, 1,0.05)	2		0.119733 48963109 383	50.0% within error bars	Elapsed time = 212.043336 62986755	Shows how for increasing maximum number of points updated, the points start to get shifted upwards above the expected. Possibly an increase in number of iterations is needed to counteract this. Could also be to do with the calculation variables being used. I will check this before continuing.

From checking the code thoroughly the error appears to be made here:

```
62      def meanError(means):
63          mean_split = np.split(means,10)
64          mean_split = np.delete(mean_split,0)
65          mean_split = np.split(mean_split,9)
66          mean_split_means = np.zeros(9)
```

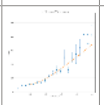
It is only deleting the first element of the first array, and not the first array itself.
Now use this code instead:

```
62      def meanError(means):
63          mean_split = np.split(means,10)
64          mean_split = np.delete(mean_split,0,axis=0)
65          mean_split_means = np.zeros(9)
```

Details	Maximum Number of Points Updated	Graph	Average Standard Error	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=10 It= 4,000,000 T=(0.05, 1,0.05)	3		0.022424219135690936	0.0% within error bars	3300.962286710739	This shows how the error in terms of the shift in points upwards is most likely due to the calculation of the action. This will need to be investigated next.

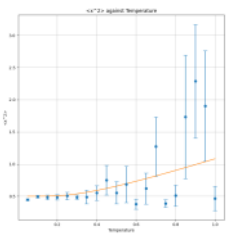
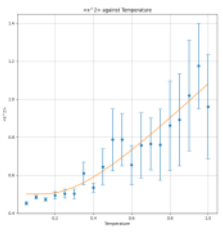
Found the action to be working correctly, therefore implying that the problem is more specific to the number of points being updated.
Potentially problem with action when looping over array boundaries.
That is the mistake!
Kinetic energy term works as expected; the potential term was incorrect need to loop in order to calculate potential term instead of using % operation.

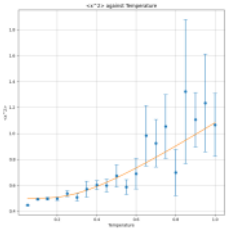
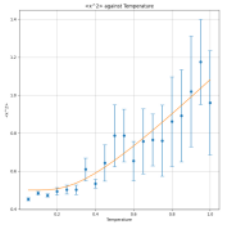
Final:

Details	Maximum Number of Points Updated	Graph	Average Standard Error	% in Error Bars	Time Taken for Calculations (ex. Plotting)	Comments
N=20 It= 100,000	5		0.12205368630485061	75.0% within error bars	Elapsed time = 58.3247323036193 85	Much better results, and within one standard deviation of expected. Will now run for higher N and number of iterations.
N=20 It= 100,000	5		0.059974201753 2382	55.0% within error bars	Elapsed time = 776.7997 54858017	Now that the standard errors are so small, due to the large number of iterations, the smaller temperature values are no longer within error bars of the expected, due to the fact that the assumption that beta is small breaks down. However, this is fine, due to the increased efficiency of the standard errors and this can be mitigated by excluding temperature values for when the simulation breaks down.

Old vs. New code:

Old code is the code used in week 9 for improved means calculation
New code is the code where multiple points can be updated with a maximum of 25% the number of N

Details	Old Graph	New Graph	Old Average Standard Error	New Average Standard Error	Old % in Error Bars	New % in Error Bars	Old Time Taken for Calculations (ex. Plotting)	New Time Taken for Calculations (ex. Plotting)	Comments
N=20 It= 100,000			0.25060888652585567	0.12205368630485061	60.0% within error bars	75.0% within error bars	Elapsed time = 44.306023597717285	Elapsed time = 58.324732303619385	Here it can be seen that the new code is a little slower but provides much better results. I will now check to see what happens for the same time.

N=20 It= 1,000,0 00 (Old) It= 100,00 0 (New)			0.13574119387852474	0.12205368630485061	75.0% within error bars	75.0% within error bars	Elapsed time = 598.12494134 90295	Elapsed time = 58.324 732303 619385	Due to the percentage within error bars being the same, and the average standard error still being higher in the old code, despite running for 10 times the amount of time as the new code. This goes to show how much more efficient the new code is in comparison to the previous code.

Code

Sunday, November 27, 2022 11:38 AM

```
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 27 14:56:23 2022

@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #check speed of program

plt.close('all')

k = 1
beta = 1
m = 1
N = 20
epsilon = beta/N
iterations = 100000
max_number_points_updated = 17

"""functions"""
def updateX(beta,N,m,k):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([iterations])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(iterations):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Consecutive points"""
        n_updated = random.randint(1,max_number_points_updated) #number of
consecutive points to update

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        updateVal = random.uniform(-1,1)
        #for i in range(n_updated):
       #     x_values
        x_values[-1,point_selected:point_selected+n_updated] += updateVal
        if point_selected+n_updated > N:
            x_values[-1,0:(point_selected+n_updated)%N] += updateVal

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2
+(x_values[-1,(point_selected+n_updated)%N]-
x_values[-1,(point_selected+n_updated-1)%N])**2-
```

```
(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%N])**2-
(x_values[-2,(point_selected+n_updated)%N]-
x_values[-2,(point_selected+n_updated-1)%N])**2)
        for j in range(n_updated):
            changeInAction += 0.5
*k*epsilon*(x_values[-1,(point_selected+j)%N]**2-
x_values[-2,(point_selected+j)%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
                x_values = np.delete(x_values,-2,0)
            else:
                x_values = np.delete(x_values,-1,0) # changes updated
point back to original value
        means[i] += np.mean(x_values**2)
    return means

"""averages function"""
def meanError(means):
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0,axis=0)
    mean_split_means = np.zeros(9)
    for i in range(0,9):
        mean_split_means[i] += np.mean(mean_split[i])
    std_err = np.std(mean_split_means)/(np.sqrt(len(mean_split_means))-1)
    overall_mean = np.mean(mean_split_means)
    return overall_mean, std_err

"""plot figure"""
fig = plt.figure(figsize=(9,9))

"""plot <x^2> against Temperature"""
means = np.array([])
std_errs = np.array([])
temperature_values = np.array([i for i in range(1,21)])/20
start_time = time.time()
for i in range(0,20):
    meanErrors = meanError(updateX(1/temperature_values[i],N,m,k))
    means = np.append(means,meanErrors[0])
    std_errs = np.append(std_errs,meanErrors[1])
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))

ax1 = fig.add_subplot(111)
ax1.grid()
ax1.errorbar(temperature_values,means, yerr=[std_errs,std_errs], capsize=
5, fmt="o")
ax1.set_xlabel('Temperature')
ax1.set_ylabel('<x^2>')
ax1.title.set_text('<x^2> against Temperature')

"""plot expected curve"""
expected = (np.exp(1/temperature_values)+1)/(2
```

```
*(np.exp(1/temperature_values)-1))
ax1.plot(temperature_values,expected)

inErrorBars = 0
for i in range(len(means)):
    if means[i]-std_errs[i]<=expected[i] and
means[i]+std_errs[i]>=expected[i]:
        inErrorBars += 1
inErrorsPercentage = inErrorBars/len(means)*100
#ax1.text(0,1,str(inErrorsPercentage)+'% within error bars',fontsize=
15,backgroundcolor='w')

avgError = np.mean(std_errs)
```

Week 11

Supervisor Meeting: 30/11/2022

30 November 2022

Outline:

- Discussed our results from last week, and how the modified method was much better than the original one.
- Spoke about what to do next: first to look at the number of points to move for an update - can we find an optimal fraction to move, is this temperature dependent?
- We could also look at other potentials.

Tasks:

- Create a way of showing the acceptance of a given update for a maximum number of points moved along a path - is this temperature dependent? If so, show this.
- Work out the optimal fraction of points to move.
- Look at a modification to the current potential - something like $V(x) = -\frac{1}{2}kx^2 + ux^4$ where u is a constant to be determined that gives interetsing results.
- Try to find some experimental or theoretical data to compare our results to for the above potential. This is only if something can be found readily available, due to the short amount of time we have left.

Distribution of Tasks:

- Both will do the above tasks.

Comments:

We all agreed that we were finished with the main goal of the project, and any new ideas would have to be relatively small due to the date - we don't want to make any large adjustments or look at different things that would take a lot of time.

The acceptance rate for an update should not be too low - this would indicate a change that is too large. It also shouldn't be too high - this would indicate a change that is too small. Both of these cases would be inefficient.

(TW) How many points to move in an update? (30/11-01/12/2022)

30 November 2022

How many points to move: (30/11/2022)

In the updated MMC method I chose that a maximum of roughly 25% of points should be moved in an update. This was an arbitrary choice. The goal of this week would be to compare different percentages of maximum points moved - to see which value gave the best acceptance rate. This may also be temperature dependent - if so could I find a way of incorporating T into the calculation.

I will produce graphs showing the acceptance rates of updates for a given fraction of points updated. All other parameters will be kept the same, and I will produce the usual position graph to see what the acceptance rate means in real terms - to try and see what a good percentage of updates accepted is.

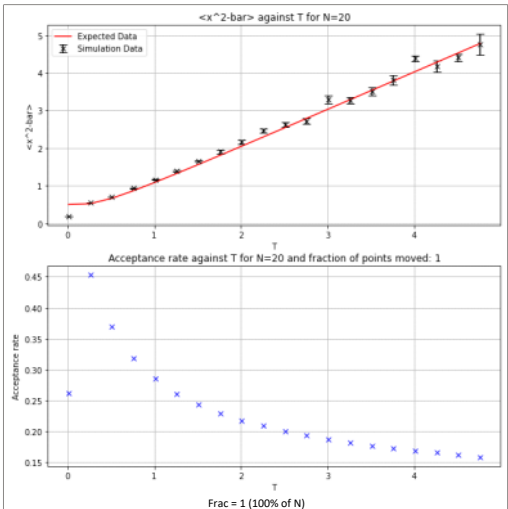
To do this, I altered the 'modupdatex' method to return '1' if an update was accepted and a '0' if it was rejected. The 'modavgMMC' method summed these values to find the total number of accepted updates for a given Temperature. This was done for a range of temperatures, and the fraction of updated accepted (accepted updates / number of iterations) was plotted against T. This was alongside the usual plot. The methods also took a 'frac' input, where this corresponded to the maximum number of points allowed to be updated for a given N.
E.g.: if frac=1/4 and N=20, then 1/4 of N - 5, was the maximum number of allowed points to be updated.

The code is as follows:

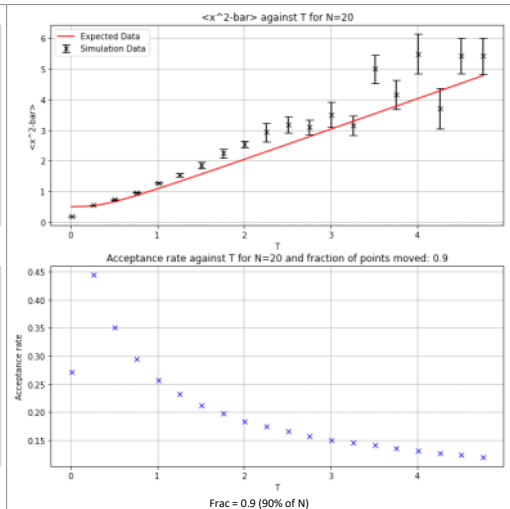
```
def modupdatex(xarr, S1, N, beta, m, k, frac):
    #This fucntion updates a selection point along a given path.
    #We first define some variables and calculate the change in action between the old and new paths.
    #Similar to the regular updatex
    accepted = 0
    eps = beta/N
    pos = random.randint(0,N-1)
    #l+1 is the number of points being updated, l=0 corresponds to the original point, so l+1=0 means 1 point being updated.
    q = np.ceil(N*frac)- 1
    l = random.randint(0,q)
    delta = random.uniform(-1,1)
    xtemp = np.copy(xarr)
    #We first add delta to each point in the selection.
    #The modulo function (%) is used incase the point chosen is towards the end of the path, so we loop back to the start.
    for i in range(l+1):
        xtemp[(pos+i)%N] = xarr[(pos+i)%N] + delta

    xtemp[N] = xtemp[0]
    #We calculate the change in action of the new path compared to the old one.
    DS = eps*((m/2)*((xtemp[pos]-xtemp[pos-1])/eps)**2 - (m/2)*((xarr[pos]-xarr[pos-1])/eps)**2 + (m/2)*((xtemp[(pos+l+1)%N]-xtemp[(pos+l)%N])/eps)**2
          - (m/2)*((xarr[(pos+l+1)%N]-xarr[(pos+l)%N])/eps)**2)
    for i in range(l+1):
        DS += eps*(1/2*k*(xtemp[(pos+i)%N])**2 - 1/2*k*(xarr[(pos+i)%N])**2)
    S2 = S1 + DS
    #If this change in action is less than or equal to zero, the update is allowed. The path is updated, as is the action.
    if DS <= 0:
        xarr = np.copy(xtemp)
        S1 = S2
        accepted = 1
    else:
        #If DS is positive, we set a probability (p) to be the negative exponent of this DS and find a random number between (0,1).
        #If this random number is less than p, the update is allowed. If not, the old path is kept.
        p = np.exp(-DS)
        r = random.uniform(0,1)
        if r < p:
            #If the update is allowed, the new path and new action are stored.
            #If the update isn't, the old path and action are kept.
            xarr = np.copy(xtemp)
            S1 = S2
            accepted = 1
    #This method returns a path and the action of the path.
    return (xarr, S1, accepted)
def modavgMMC(N, beta, m, k, it, power, frac):
    #N being the number of steps in imaginary time; beta, m, and k being system varibales
    #creates an empty array consisting of all zeros, of size N+1, that being x_0 to x_N
    #0 is the arbitary value chosen for the points
    x = np.zeros(N+1)
    S1 = 0
    accepted = 0
    #Avgpath list stores the averages of all path generated, the first path has an average of 0.
    avgpath = [0]
    #We iterate over the number supplied-1 (as there are an iteration number of paths, the first being all zeros).
    #We use the updateavg function to update our paths, with the path and action of the path returned being stored for the next iteration.
    #The average value of the path is appended to the avgpath list.
    for i in range(it-1):
        #This is the modified function, so uses the modified updatex method.
        update = modupdatex(x, S1, N, beta, m, k, frac)
        x = np.copy(update[0])
        S1 = update[1]
        xpow = x**power
        avgpath.append(np.mean(xpow))
        accepted += update[2]
    return (np.asarray(avgpath), accepted)
```

Results:
The usual parameters of: m=k=1, N=20, T=[0.01,5] in steps of 0.25 and It=1,000,000 were used. Frac gives the fraction of N that is the maximum number of points that maybe updated.

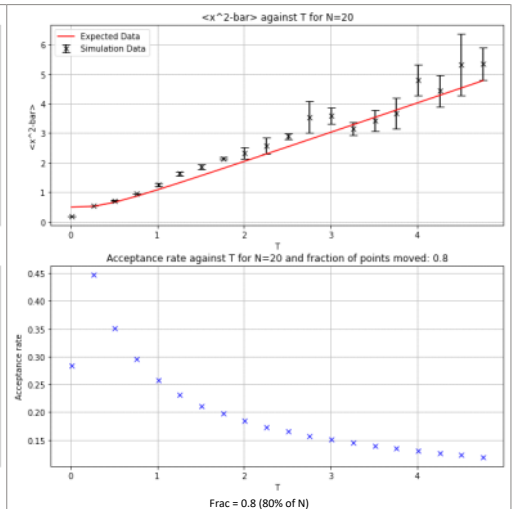
The acceptance rate shouldn't be too high or too low. I will look at the data to see a good range. This will probably be temperature dependent.
I'll look at frac=1,09,0.8,0.7 etc to see what happens.



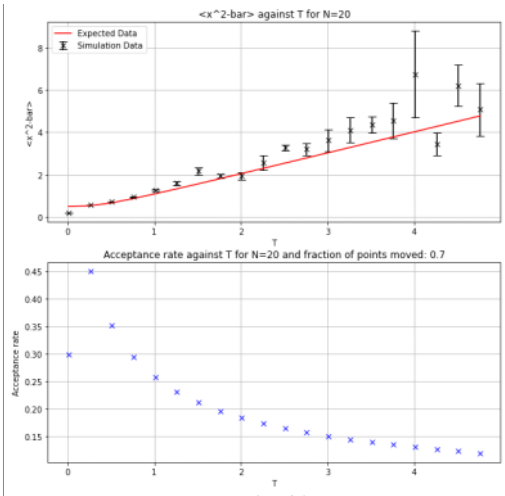
Frac = 1 (100% of N)
Max Number of points changed: 20



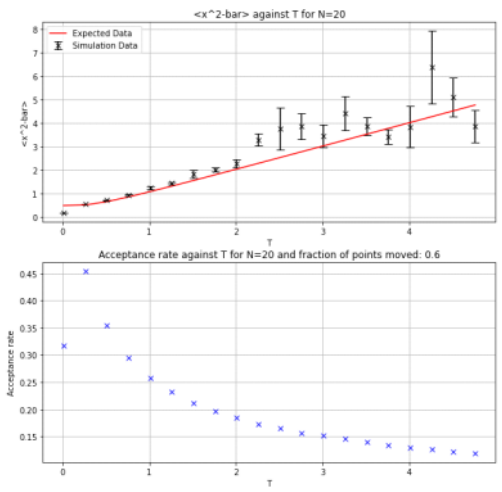
Frac = 0.9 (90% of N)
Max Number of points changed: 18



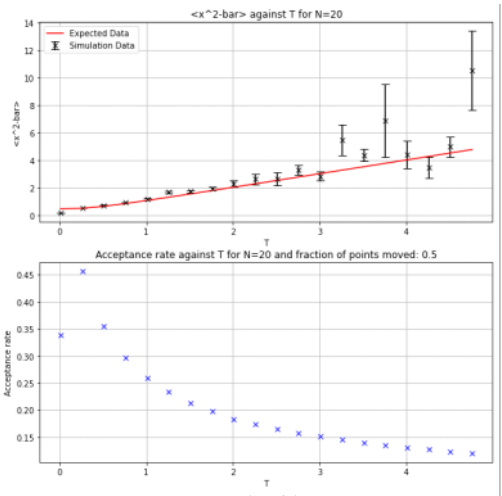
Frac = 0.8 (80% of N)
Max Number of points changed: 16



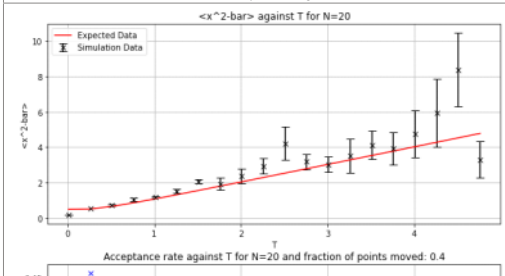
Frac = 0.7 (70% of N)
Max Number of points changed: 14

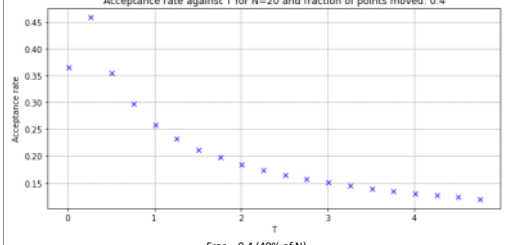


Frac = 0.6 (60% of N)
Max Number of points changed: 12

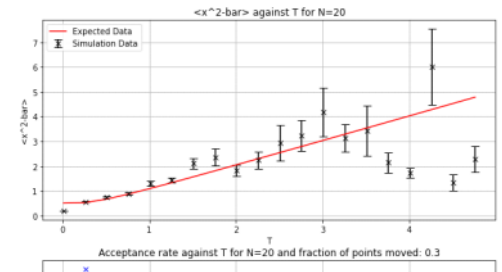


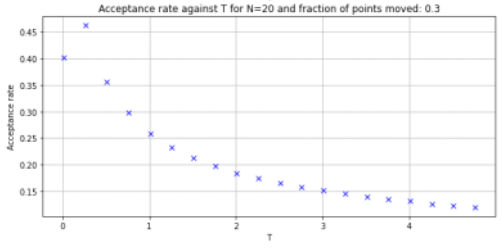
Frac = 0.5 (50% of N)
Max Number of points changed: 10



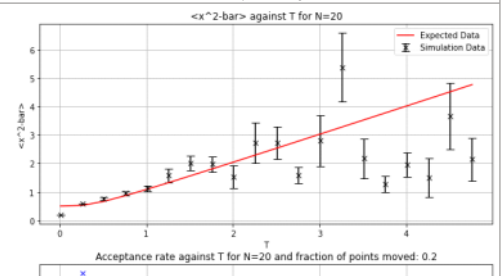


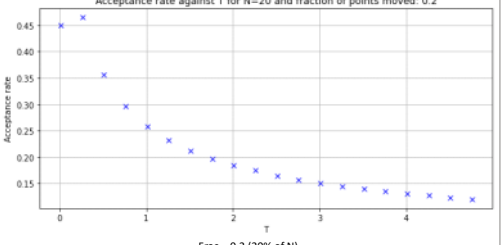
Frac = 0.4 (40% of N)
Max Number of points changed: 8





Frac = 0.3 (30% of N)
Max Number of points changed: 6

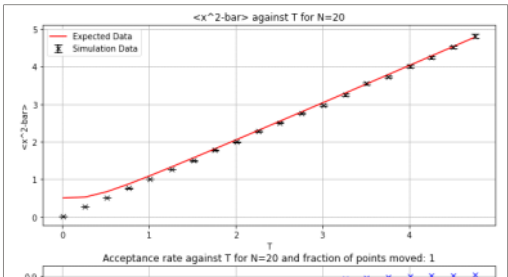


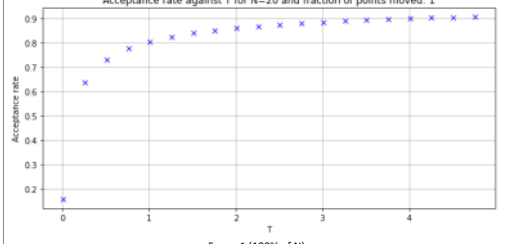


Frac = 0.2 (20% of N)
Max Number of points changed: 4

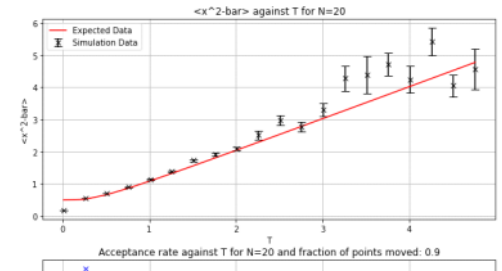
Comments:
The above show that the acceptance rate is a function of temperature - it looks roughly like an exponential decay (ignoring the first point). We also see very little difference with changing this fraction value - this is probably due to the random nature of the number of points moved. As it stands it makes sense to leave the fraction as 1 - this also produced the best data for all T values.

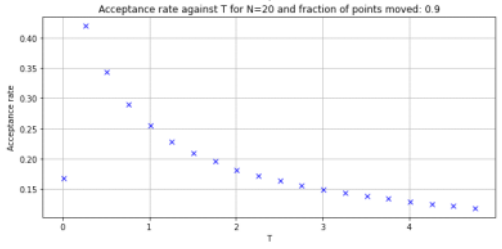
I now want to look at **removing the random choice of the number of points moved: the number moved will always be the designated fraction of N.** I suspect this will give worse results than above. Especially for a large fraction.



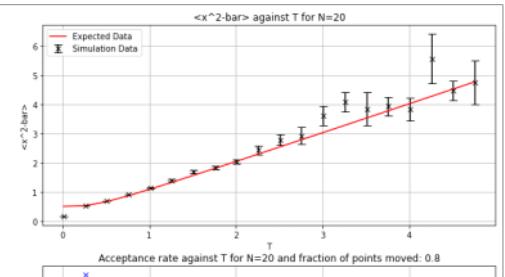


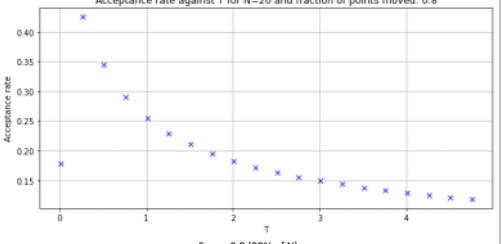
Frac = 1 (100% of N)
Number of points changed: 20
Note: This is essentially the N=1 case, so the classical result (as shown above) is expected.



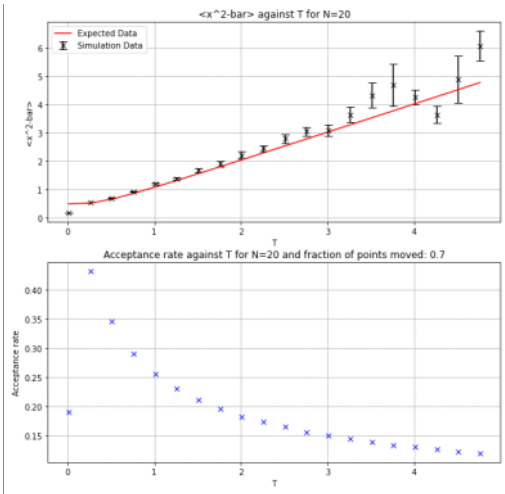


Frac = 0.9 (90% of N)
Number of points changed: 18

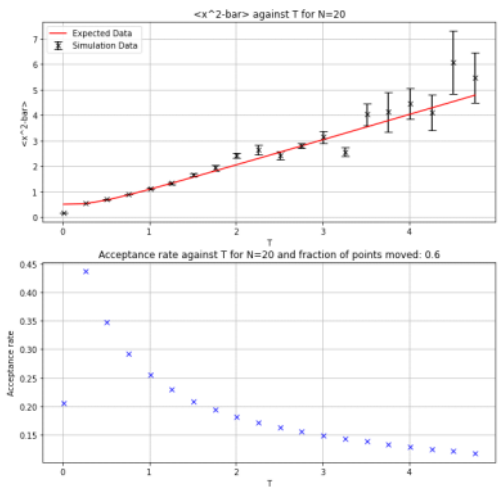




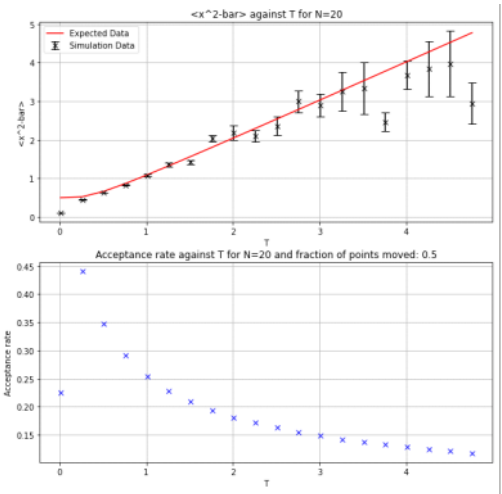
Frac = 0.8 (80% of N)
Number of points changed: 16



Frac = 0.7 (70% of N)
Number of points changed: 14



Frac = 0.6 (60% of N)
Number of points changed: 12



Frac = 0.5 (50% of N)
Number of points changed: 10

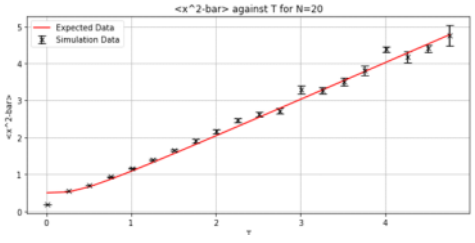
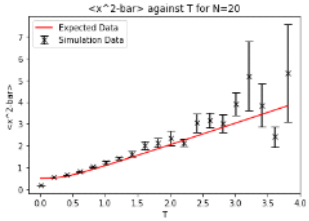
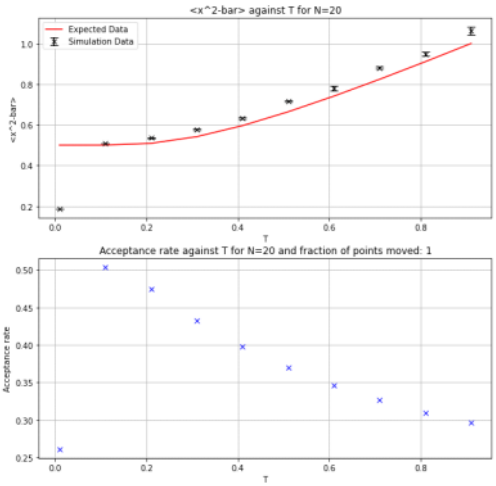
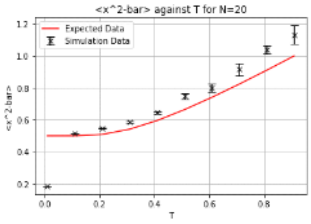
Comments:
We get a similar pattern to above, expect for the case of frac=1. The acceptance rate is always quite low for a large T - this is expected due to the comments from last week. The best results are for a random number of points being update, with the max number being equal to N.

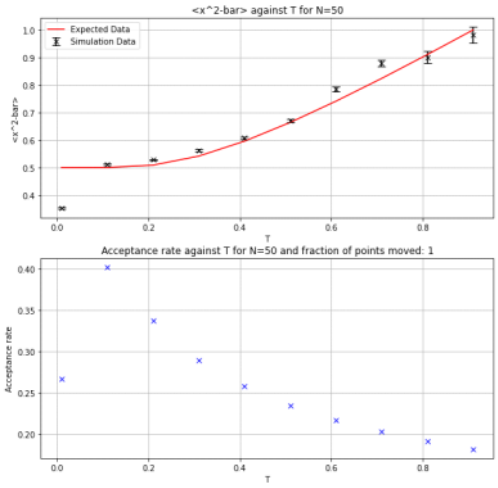
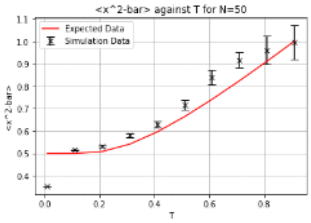
For a random number between 1 point and N points moved, you expect to see an even distribution of points chosen to be moved. As seen the by the acceptance rates in the first set of graphs, they are roughly the same at each given T, independent of the number chosen - this is probably due to the random nature of choosing the number of points to move. Given that the plot of N=20 at it=1,000,000 for a maximum number of points moved = 20 is much accurate than for a similar case for the max number of points moved being 20% or 25% of N for the same number of iterations - it makes sense to allow the range to be from 1 point to N points - randomly chosen. This makes the algorithm more efficient, and means fewer iterations are needed to produce accurate data.
So I **will alter the code to update a random number of points between 1 and N.** I will create a few plots with this set-up to see how they compare to other versions of the code, and look at low temperatures for larger N. If these still produce accurate results then I shall keep this method, and look into a different potential. I'll also produce an acceptance rate plot for the old method - of simply updating one point at a time to see how this affects things. This may explain why older plots needed many more iterations as compared to now.

```
        i = random.randint(0,N-1)
        delta = random.uniform(-1,1)
        xtemp = np.copy(xarr)
        #We first add delta to each point in the selection.
        #The modulo function (%) is used incase the point chosen is towards the end of the path, so we loop back to the start.
        for i in range(1+1):
            xtemp[(pos+i)%N] = xarr[(pos+i)%N] + delta
```

The fraction of N has been removed - the additional number of points moved is now N-1. So the maximum number of points moved is N in total.

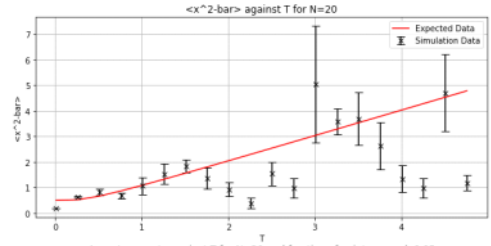
Results: (01/12/2022)
m=k=1, Max number of points changed=N (random number between 1 and N changed)

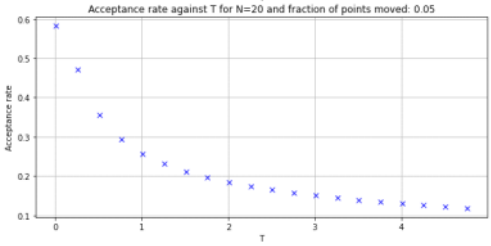
Run	Values	Code Plot	Comments
1	N=20; it = 1,000,000; T=[0.01,5] in steps of 0.25		80% of points are within an error bar of the expected line, with only the point at T=0.01 not close to the expected data. This took roughly 3,000,000 iterations for the case of 25% of points being moved - so this plot took roughly 1/3 of the time of the pervious plot. We also have much smaller error bars.  Old code plot, where max number of points moved was 25% of N. The interesting case would be for low T - to see if we can get the points closer to the expected curve.
2	N=20; it = 1,000,000; T=[0.01,1] in steps of 0.1		The plot produced here is very similar to the case below from last week, where a maximum number of points moved was 0.25 of N:  We have smaller error bars in the new plot (left) and points slightly closer to the curve. We have the usual point at very low T not close to the line - expected. I shall try a higher N for the same number of iterations to see how close we can get to the curve. The acceptance rate at this low T is much higher than for large T, but a largest of 50% is alright - the points are still following the expected pattern.

3	N=50; it = 1,000,000; T=[0.01,1] in steps of 0.1	 	The code below is from when the maximum number of points moved was 25% of N. This was done for it= 6,000,000:  We see more accurate results in the new code, with fewer iterations than for the older version with more iterations. This shows the new way - i.e. choosing an random number of points to update between 1 and N - is more efficient than the older way.

Conclusion:
The results show that having a random number of points updated between 1 and N is the most efficient modification to the method. This produces more accurate results in like-for-like cases, or the same accurately for fewer iterations - it is thus more efficient. I shall therefore keep with this new method.

Before looking at alternative potentials, I want to produce an acceptance rate graph for simply moving 1 point, for comparison. This was done for N=20 at it=1,000,000. The fraction was 0.05, which works out to be one point moved:





The acceptance rate here is similar to other fractions of N. it is slightly higher for large T, but for low T roughly the same. This makes sense when you look at the plot of position squared - the points at higher T do not follow the expected line. This is because we have roughly the same number of updates accepted - but an update only moves a single point, so will take longer to get the expected value.

Overall Conclusions:
The most efficient maximum number of points to move is N - with a random number of points moved between 1 and N. This produces more accurate results for the same number of iterations, meaning the code is more efficient as can be run for a lower number of iterations to get the same accuracy. The acceptance rate is similar for all upper bounds on the number of points to move, especially when a random number is chosen. This follows our understanding of this modified method and why it is useful to move a number of points in one go, as opposed to a single point.

Moving forwards, I will use this new updated version, with a maximum number of points moved equal to N. The next step will be to look at other potentials - either ones with known solutions, or approximate solutions to allow me to compare my results against. This is if I can find something readily available to use as a comparison.

TK

Saturday, December 3, 2022 6:53 PM

Please expand tab for full weekly report

Theory

Saturday, December 3, 2022 6:53 PM

Outline:

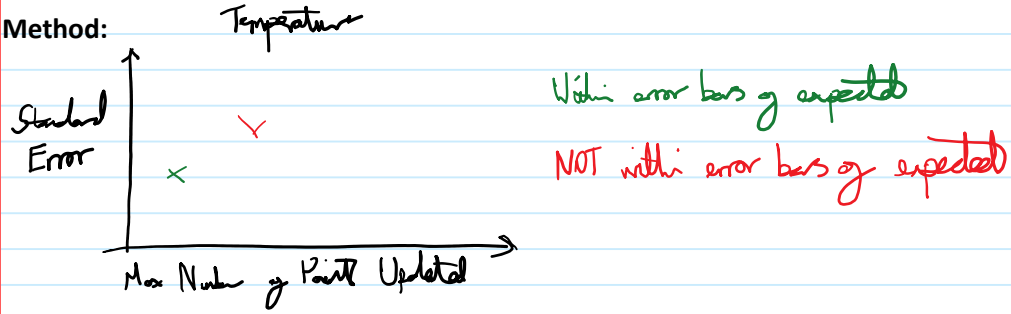
- Want to find the optimal number of points to update
  - How does this depend on temperature

Plan:

- Overall Logic structure
  - Pick temperature
  - Repeat for different number of points updated
  - Change temperature

- Temperature values
  - Minimum temperature value, as previously calculated for certain N, T=1/N+0.04
  - Maximum temperature value, chosen arbitrarily = 5
  - Step = 0.5
- Number of points updated
  - Minimum number = 1
  - Maximum number = N
  - Step = 1
- Analysis
  - Mean closest to expected value
  - Standard errors small

Method:

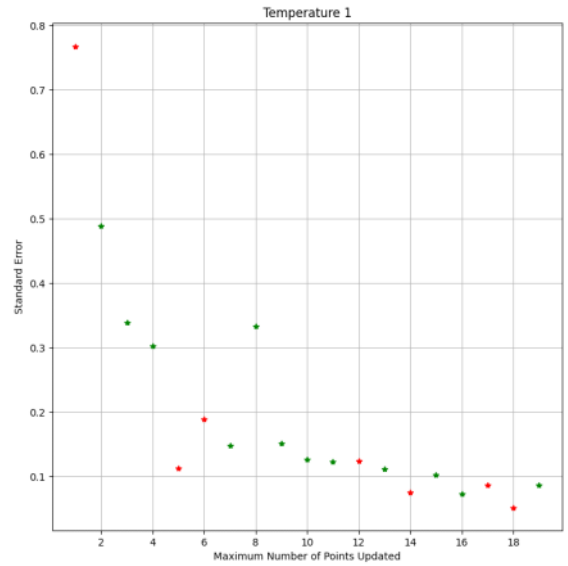
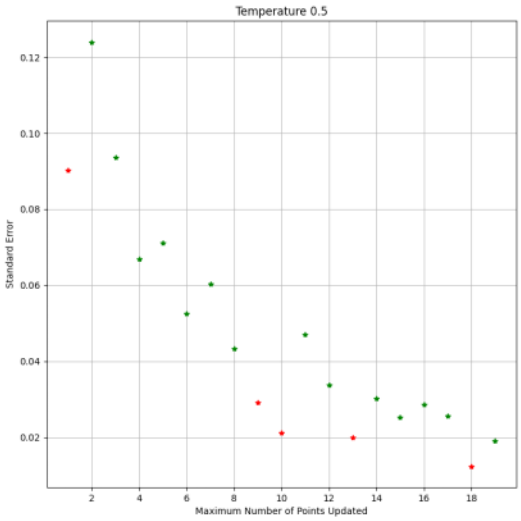


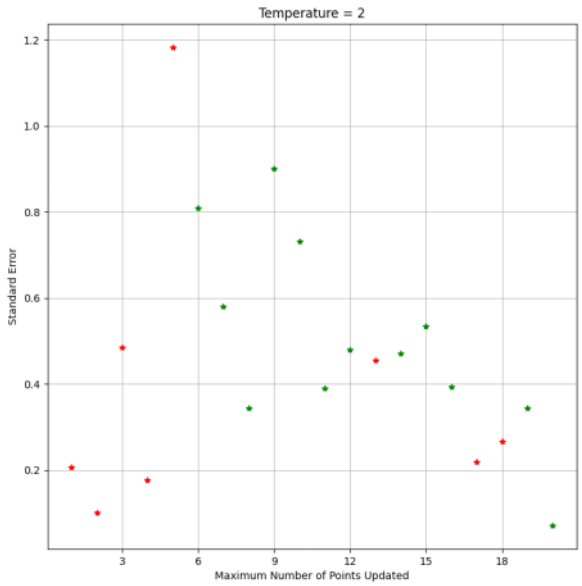
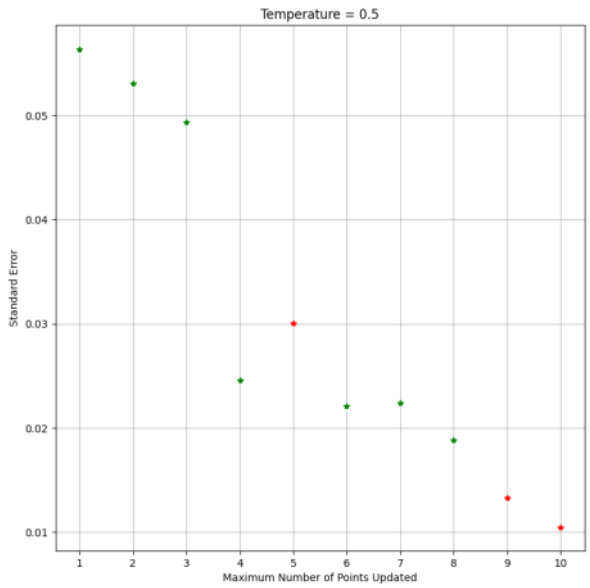
Take GREEN points with lowest standard error from each temperature

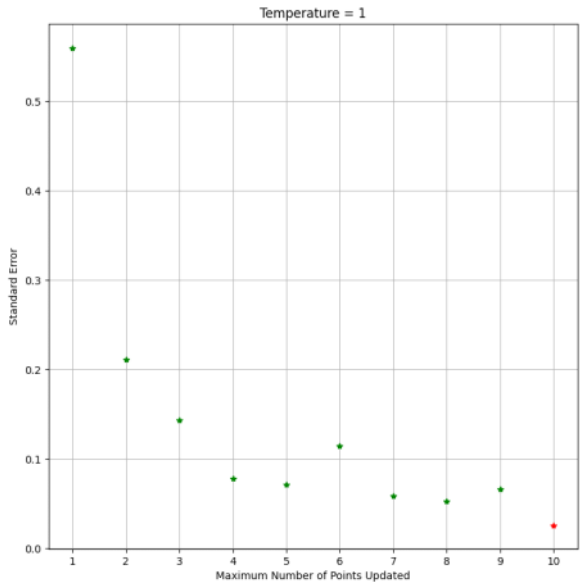
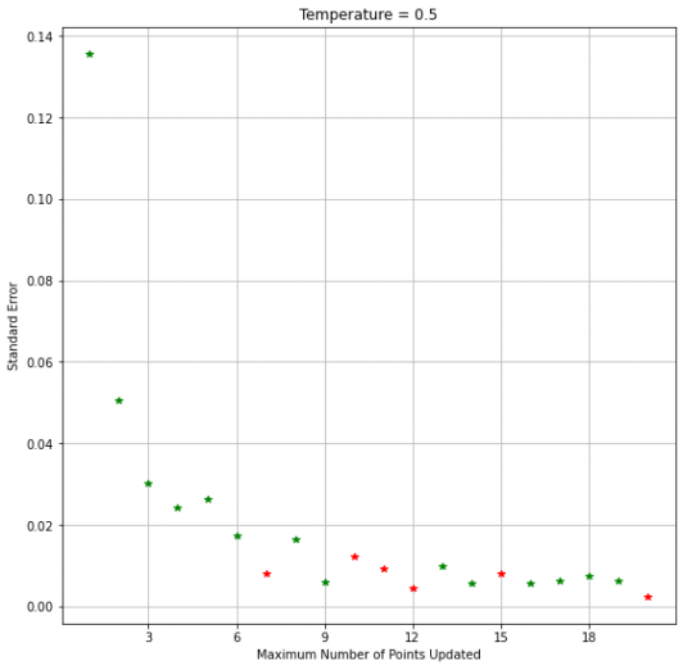
Plot on a graph, find rough curve that fits → new update number

Results

Sunday, December 4, 2022 11:38 AM

Details	Graph	Comments
N=20 It= 100,000 T=1		From this initial graph it can be seen that as the number of points that can be updated increases, the standard error decreases. Although there are points where not within error bars of expected, this is probably due to the low number of iteration. I will try with higher iterations to see if this is the case.
N=20 It= 100,000 T=0.5		Similar to above, need to try for more iterations, but does look very promising.

N=20 It= 100,000 T=2		This shows how for larger temperatures, a higher number of iterations is required, and hence it is a lot more scattered, but there does seem to be a similar trend to the one outlined above.
N=10 It= 100,000 T=0.5		Similar to the graphs above. However, the highest number of points updated were not within error bars of expected; need to try this out for a higher number of iterations and see if this is corrected.

N=10 It= 100,000 T=1		All but the highest number of iterations is within error bars of expected, but overall shape is consistent with above.
N=20 It= 1,000,000 T=0.5		This code shows again that for N points updated is not within error bars of expected, however this could just be due to the low temperature. Will investigate this next.

Conclusion:

Therefore, it can be seen that it makes the most sense, to set the maximum number of points that can be updated equal to N; this leads to the lowest standard error within error bars of the expected value for large number of iterations.
Further investigation would be needed into the distribution of selecting the number to be updated, and how this would affect the percentage of results within error bars of expected.

Code

Sunday, December 4, 2022 11:38 AM

```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #check speed of program
from matplotlib.ticker import MaxNLocator

plt.close('all')

k = 1
beta = 1
m = 1
N = 20
epsilon = beta/N
iterations = 1000000

temperatureVal = 0.5

"""functions"""
def updateX(beta,N,m,k,n_update):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([iterations])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(iterations):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        updateVal = random.uniform(-1,1)

        x_values[-1,point_selected:point_selected+n_update] += updateVal
        if point_selected+n_update > N:
            x_values[-1,0:(point_selected+n_update)%N] += updateVal

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2
+(x_values[-1,(point_selected+n_update)%N]-
x_values[-1,(point_selected+n_update-1)%N])**2-
(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%N])**2-
(x_values[-2,(point_selected+n_update)%N]-
x_values[-2,(point_selected+n_update-1)%N])**2)
        for j in range(n_update):
            changeInAction += 0.5
```

```
*k*epsilon*(x_values[-1,(point_selected+j)%N]**2-
x_values[-2,(point_selected+j)%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
                x_values = np.delete(x_values,-2,0)
            else:
                x_values = np.delete(x_values,-1,0) # changes updated
point back to original value
        means[i] += np.mean(x_values**2)
    return means

"""averages function"""
def meanError(means):
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0,axis=0)
    mean_split_means = np.zeros(9)
    for i in range(0,9):
        mean_split_means[i] += np.mean(mean_split[i])
    std_err = np.std(mean_split_means)/(np.sqrt(len(mean_split_means))-1)
    overall_mean = np.mean(mean_split_means)
    return overall_mean, std_err

"""plot figure"""
fig = plt.figure(figsize=(9,9))

"""plot <x^2> against Temperature"""
means = np.array([])
std_errs = np.array([])
n_updated = np.arange(1,N+1)
start_time = time.time()
for i in range(0,len(n_updated)):
    meanErrors = meanError(updateX(1/temperatureVal,N,m,k,n_updated[i]))
    means = np.append(means,meanErrors[0])
    std_errs = np.append(std_errs,meanErrors[1])
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))

ax1 = fig.add_subplot(111)
ax1.grid()
#x1.errorbar(temperatureVal,means, yerr=[std_errs,std_errs], capsize=5,
fmt="o")
ax1.set_xlabel('Maximum Number of Points Updated')
ax1.set_ylabel('Standard Error')
ax1.title.set_text('Temperature = '+str(temperatureVal))
ax1.xaxis.set_major_locator(MaxNLocator(integer=True))


"""plot expected curve"""
expected = (np.exp(1/temperatureVal)+1)/(2*(np.exp(1/temperatureVal)-1))
#ax1.plot(temperatureVal,expected)

inErrorBars = 0
```

```
for i in range(len(means)):
    if means[i]-std_errs[i]<=expected and means[i]+std_errs[i]>=expected:
        inErrorBars += 1
        ax1.plot(n_updated[i],std_errs[i],'g*')
    else:
        ax1.plot(n_updated[i],std_errs[i],'r*')
inErrorsPercentage = inErrorBars/len(means)*100
#ax1.text(0,1,str(inErrorsPercentage)+'% within error bars',fontsize=
15,backgroundcolor='w')

avgError = np.mean(std_errs)
```

Double-Well Potential

Thursday, December 8, 2022 9:58 AM

$$V(x) = 3x^4 - 8x^2.$$

Potential to test with code, if time permits.

Week 12

Supervisor Meeting: 8/12/2022

08 December 2022

Outline:

- Discussed our results from last week, and how the modified method was more efficient than the original one.
- Spoke about looking at a different potential - this would be difficult to do in the time left, given that we couldn't find anything readily available. We decided not to pursue this any further.
- Looked back at the Gaussian probability for finding x at a given location for low T - could look at doing this for a higher temperature, the width of the Gaussian should be exactly the expectation value of x^2 at this temperature.
- Spoke about writing the report.

Tasks:

- Look at the probability distribution for higher temperatures as another result to talk about.

Distribution of Tasks:

- Both will do the above tasks.

Comments:

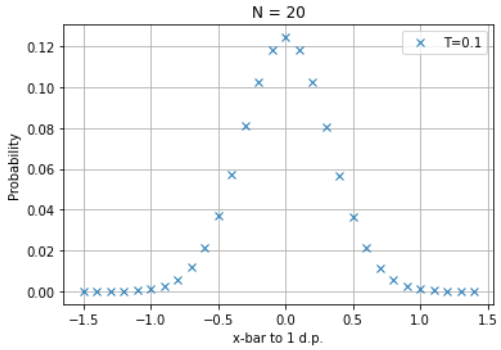
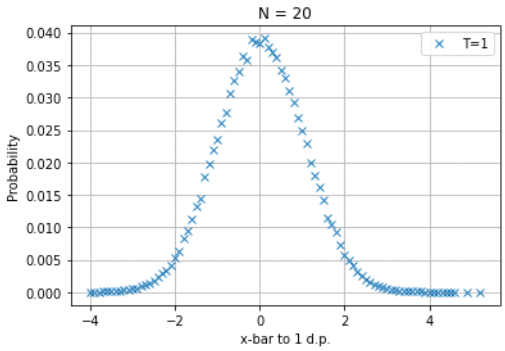
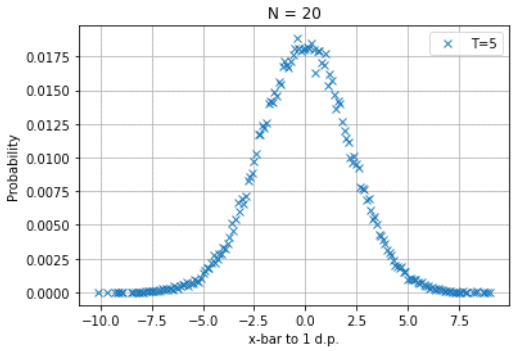
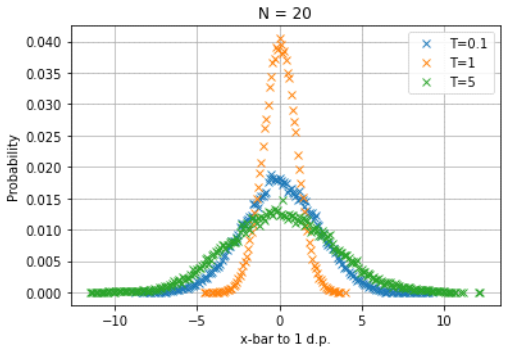
Apart from the Gaussian plots, there is no more results to look into. We decided to start to write our reports.

(TW) Average Position Plots: 13/12/2022

13 December 2022

How many points to move: (13/12/2022)

As mentioned in the meeting notes, we have covered the main idea of the project and looked at making it more efficient. Befor e I started to plan the report, I wanted to produce the position plots mentioned in the previous meeting for a range of temperatures to see if th ey did follow a Gaussian plot. The code used was the same as when I looked at the zero point position. Producing the plots wouldn't be diffic ult, but trying to normalise them such that a good Gaussian fit could be found would be.

Run	Parameters	Plot
1	N=20, it=1,000,000 T=0.1	
2	N=20, it=1,000,000 T=1	
3	N=20, it=1,000,000 T=5	
4	N=20, it=1,000,000 T=0.1, 1, 5	

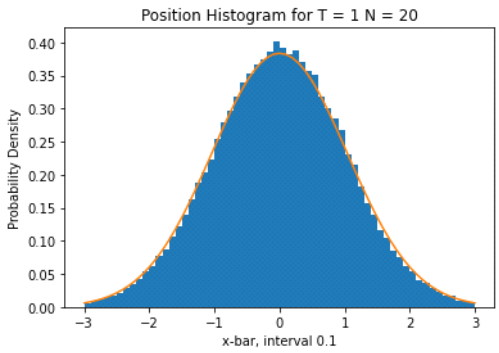
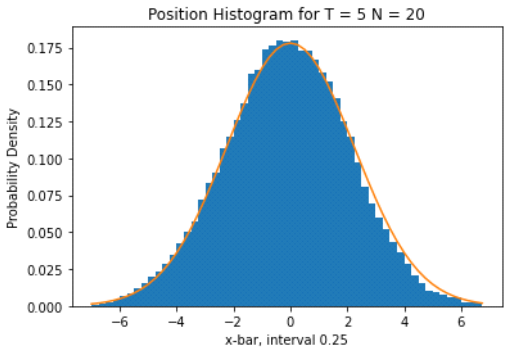
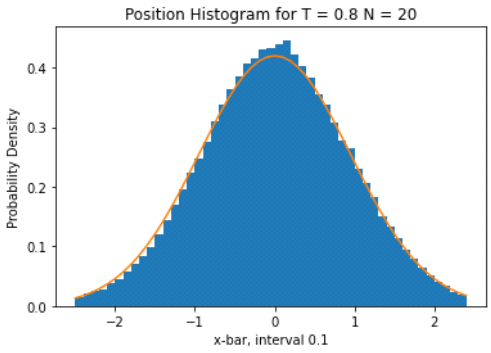
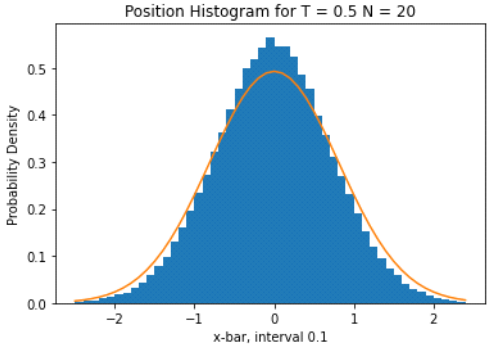
I managed to produce plots that look like Gaussians, that widen as the temperature increases. This is expected as the standar d deviation will simply be the square root of the expectation value of x^2, which for high T is equal to T. But I need to do some more work to find a curve that fits the data due to the ranges in pro babilities, as for these x-values a standard Gaussian is much higher - this could be due to constants etc. I think the Gaussian will be the same as normal, but will need to be scaled to fit the data.

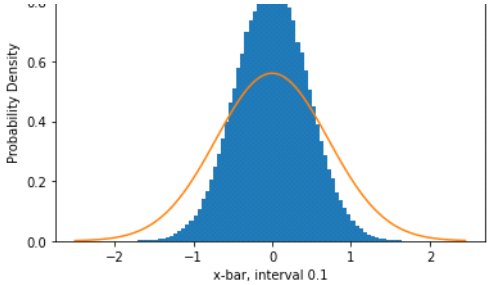
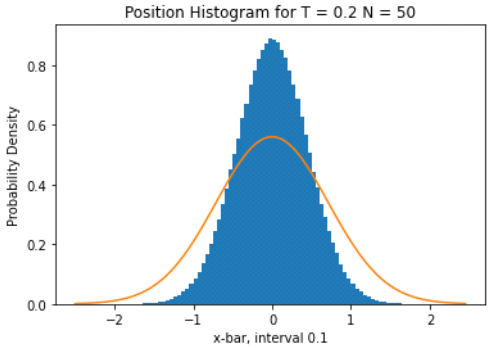
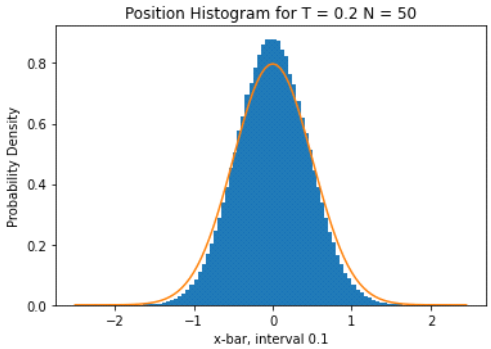
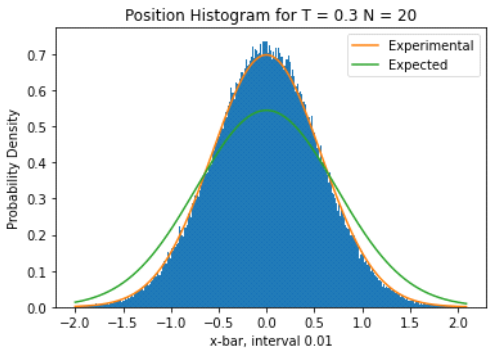
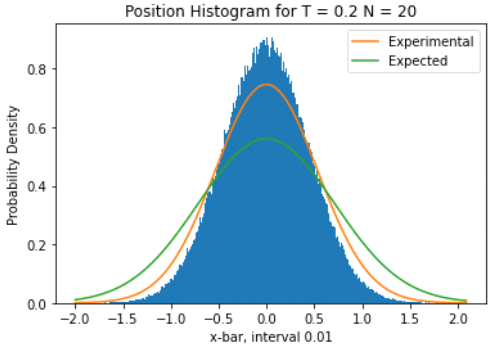
After speaking to my supervisor and discussing the problem we figured out what needed to be plotted. It would be better to pl ot a probability density over the raw probabilities as this would then match the gaussian distribution for the given data set. What was also being plotted above wa s essentially a histogram, with an interval of width 0.1, so it would be better to plot bars for the expected data using the histogram function.

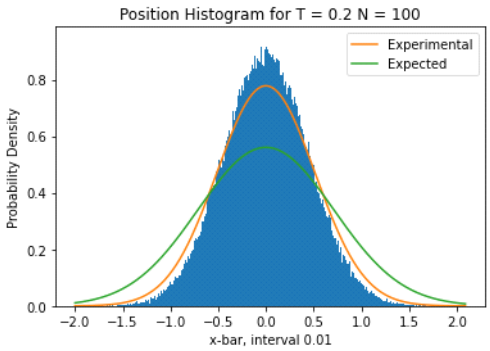
The new graphs are below, with the variance worked out by using the expectation value of x^2 as used in previous plots (this is because $\langle x \rangle = 0$) :

$$\langle x^2 \rangle = \frac{\exp\left(\frac{1}{T}\right)+1}{2(\exp\left(\frac{1}{T}\right)-1)} = \sigma^2$$

I used N=20 for the higher temperatures, as we saw in the previous few weeks that with the updated method this gave very accu rate data. For a low temperature - I may need to change N to be larger - such as N=50. For the same reasons as for last week - this produces data closer to the expected values.

Run	Parameters	Plot	Comments
1	N=20, it=1,000,000 T=1 $\sigma = 1.04$ (3sf) Interval / bin width = 0.1		Results look good and follow what is expected.
2	N=20, it=2,000,000 T=5 $\sigma = 2.24$ (3sf) Interval / bin width = 0.25		Similar to above. I had to make the bin width wider here due to the larger range of values. A few bars are outside of the curve, but only slightly and the data still follows what is expected.
3	N=20, it=1,000,000 T=0.8 $\sigma = 0.950$ (3sf) Interval / bin width = 0.1		Good results, N not an issue at this temperature.
4	N=20, it=1,000,000 T=0.5 $\sigma = 0.810$ (3sf) Interval / bin width = 0.1		Results are starting to deviate away from the expected curve. Possibly due to N, will look at a lower temperature then start to change N.
5	N=20, it=1,000,000 T=0.2 $\sigma = 0.712$ (3sf)		Method starts to break down for lower T. Will increase N and number of iterations.

	Interval / bin width = 0.1		
6	N=50, it=3,000,000 T=0.2 $\sigma = 0.712$ (3sf) Interval / bin width = 0.1		Again now following the expected curve - still a gaussian but with a smaller standard deviation. This may be due to how things work at lower T.
6.1			This is the same as the above plot but with $\sigma = 0.5$, which roughly is the previous value squared. **This is not what is expected!**
7	N=20, it=1,000,000 T=0.3 $\sigma = 0.733$ (3sf) Interval / bin width = 0.01		The orange line is a Gaussian using the $\langle x^2 \rangle$ produced by the model, while the green line is calculated from the expected result. This shows for lower T we start to vary from the expected result - this we also so in the graphs for lower T. I shall do the same for 0.2. So for very low temperatures we get a steeper gaussian than is expected.
8	N=20, it=1,000,000 T=0.2 $\sigma = 0.712$ (3sf) Interval / bin width = 0.01		Same as above, but now the plot is moving away from both distributions. Clearly the drop off temperature in the method is around 0.2. I have tried increasing N but this doesn't create much more accurate data. I suspect this is a limitation of the model, and with more time it could be investigated.

9	N=100, it=1,000,000 T=0.2 $\sigma = 0.712$ (3sf) Interval / bin width = 0.01		Plot gets closer to the experimental calculation of $\langle x^2 \rangle$. So the issue is probably due to the N being too small.

Conclusion:

We get good results for larger T, but around T=0.2 things start to deviate from what is expected. I believe this is due to ha ving N be too small, and such $\langle x^2 \rangle$ not exactly following what is expected. We also may have a non-zero ⟨x⟩ value which may cause errors.

However the results for T>0.2 are good and follow what is expected. Given more time I would have liked to have investigated t his discrepancy further, but given where we are in the term I do not believe I have the time. For lower T we still get gaussian distributions, showing the zero point energy, just ones that a re taller than expected.

TK

Thursday, December 8, 2022 11:40 AM

Please expand tab for full weekly report

Theory

Saturday, December 10, 2022 11:35 PM

Outline:
From the supervisor meeting, we discussed the fact that we should start writing up our reports, but we also talked about how the average position plot (like the one done for the zero-point energy, can also be plotted for different values of temperature.

Background:
Expected variance in Gaussian plots,

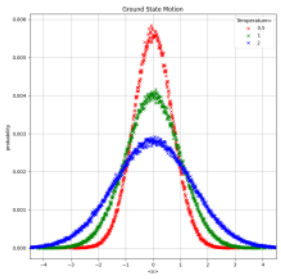
$$\sigma_x^2 = \langle x^2 \rangle - \langle x \rangle^2 = \langle x^2 \rangle = \frac{1}{e^{\frac{1}{T}} - 1} + \frac{1}{2}$$

Plan:
I will use the previous code used in week 9 for the zero-point motion probability graphs to produce plots for different values of temperature and compare these with the expected Gaussian plots.

- Run previous code at a given temperature above the minimum temperature as stated previously
- Plot expected Gaussian on the same graph, normalised
- Compare

Results

Sunday, December 11, 2022 12:04 AM

Details	Graph	Time Taken for Calculations (ex. Plotting)	Comments
N=20 It= 1,000,000 T=0.5,1,2		Elapsed time = 136.11920928955078	Agrees with what is expected analytically, that as temperature increases, as does the variance and standard deviation in the gaussian produced.

Code

Sunday, December 11, 2022 12:41 AM

```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time
import scipy #used for calculating expected Gaussian

plt.close('all')

"""values"""
k = 1
m = 1
N = 10

temperatureVal = 1/N + 0.04

iterations = 1000000

"""functions"""
def updateX(beta,N,m,k,n_update):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([iterations])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(iterations):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        updateVal = random.uniform(-1,1)

        x_values[-1,point_selected:point_selected+n_update] += updateVal
        if point_selected+n_update > N:
            x_values[-1,0:(point_selected+n_update)%N] += updateVal

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2
+(x_values[-1,(point_selected+n_update)%N]-
x_values[-1,(point_selected+n_update-1)%N])**2-
(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%N])**2-
(x_values[-2,(point_selected+n_update)%N]-
x_values[-2,(point_selected+n_update-1)%N])**2)
        for j in range(n_update):
            changeInAction += 0.5
```

```
*k*epsilon*(x_values[-1,(point_selected+j)%N]**2-
x_values[-2,(point_selected+j)%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
                x_values = np.delete(x_values,-2,0)
            else:
                x_values = np.delete(x_values,-1,0) # changes updated
point back to original value
        means[i] += np.mean(x_values)
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0)
    return means

"""averages function"""
def meanError(means):
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0)
    mean_split = np.split(mean_split,9)
    mean_split_means = np.zeros(9)
    for i in range(0,9):
        mean_split_means[i] += np.mean(mean_split[i])
    overall_mean = np.mean(mean_split_means)
    return overall_mean

start_time = time.time()

positions = np.round(updateX(1/0.5,N,m,k,N),2)
position, count = np.unique(positions, return_counts=True)
probability = count/iterations

positions2 = np.round(updateX(1/1,N,m,k,N),2)
position2, count2 = np.unique(positions2, return_counts=True)
probability2 = count2/iterations

positions3 = np.round(updateX(1/2,N,m,k,N),2)
position3, count3 = np.unique(positions3, return_counts=True)
probability3 = count3/iterations

end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))


"""plot figure"""
fig = plt.figure(figsize=(9,9))

ax1 = fig.add_subplot(111)
ax1.set_xlabel('<x>')
ax1.set_xlim(-1.5,1.5)
ax1.set_ylabel('probability')
ax1.set_title('Ground State Motion')
ax1.grid()
ax1.plot(position,probability,marker='x',linestyle='none',color='r',label=
```

```
'0.5')
ax1.plot(position2,probability2,marker='x',linestyle='none',color='g',labe
l='1')
ax1.plot(position3,probability3,marker='x',linestyle='none',color='b',labe
l='2')
ax1.legend(title='Temperature=',loc='upper right')

"""expected Gaussian
variance = (np.exp(1/temperatureVal)+1)/(2*(np.exp(1/temperatureVal)-1))
x = np.linspace(-10, 10, 1000)
y = scipy.stats.norm.pdf(x,0,np.sqrt(variance))
y = y/np.sum(y)
ax1.plot(x,y)
"""
```

Ground State Energy Code

Wednesday, January 11, 2023 1:00 PM

```
# -*- coding: utf-8 -*-
"""
@author: tobyk
"""


"""import modules"""
import random #random number generator
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time

plt.close('all')

"""values"""
k = 1
m = 1
N = 10

lowest_temp = 1/N + 0.04

iterations = 100000

"""functions"""
def updateX(beta,N,m,k,n_update):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([iterations])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(iterations):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        updateVal = random.uniform(-1,1)

        x_values[-1,point_selected:point_selected+n_update] += updateVal
        if point_selected+n_update > N:
            x_values[-1,0:(point_selected+n_update)%N] += updateVal

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2
+(x_values[-1,(point_selected+n_update)%N]-
x_values[-1,(point_selected+n_update-1)%N])**2-
(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%N])**2-
(x_values[-2,(point_selected+n_update)%N]-
x_values[-2,(point_selected+n_update-1)%N])**2)
        for j in range(n_update):
            changeInAction += 0.5
*k*epsilon*(x_values[-1,(point_selected+j)%N]**2-
```

```
x_values[-2,(point_selected+j)%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
                x_values = np.delete(x_values,-2,0)
            else:
                x_values = np.delete(x_values,-1,0) # changes updated
point back to original value
        means[i] += np.mean(x_values**2)
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0,axis=0)
    mean_split = np.hstack(mean_split)
    return mean_split

"""averages function"""
def meanError(means):
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0)
    mean_split = np.split(mean_split,9)
    mean_split_means = np.zeros(9)
    for i in range(0,9):
        mean_split_means[i] += np.mean(mean_split[i])
    overall_mean = np.mean(mean_split_means)
    return overall_mean

start_time = time.time()
positions = updateX(1/lowest_temp,N,m,k,N)
end_time = time.time()
print('Elapsed time = ', repr(end_time - start_time))


"""plot figure"""
fig = plt.figure(figsize=(9,9))

ax1 = fig.add_subplot(111)
ax1.set_xlabel('<x>')
ax1.set_ylabel('probability')
ax1.set_title('Ground State Motion')
ax1.grid()
weights = np.ones_like(positions) / len(positions)
plt.hist(positions, bins=100, weights=weights)

#%%
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 10 19:09:16 2023

@author: tobyk
"""



"""import modules"""
import random #random number generator
```

```
import numpy as np #exp, append, array
import matplotlib.pyplot as plt #scatter plot
import time #check speed of program

plt.close('all')

k = 1
m = 1
N = 50
#lowest_temp = 1/N + 0.04
lowest_temp=0.5
beta = 1/lowest_temp
epsilon = beta/N
iterations = 5000000
max_number_points_updated = N-1

"""functions"""
def updateX(beta,N,m,k):
    epsilon = beta/N
    action = 0
    """Means"""
    means = np.zeros([iterations])
    """Pick arbitrary value for x"""
    x_values = np.array([np.zeros([N])])
    for i in range(iterations):
        """duplicate array"""
        x_values = np.append(x_values,x_values,axis=0)

        """Consecutive points"""
        n_updated = random.randint(1,max_number_points_updated) #number of
consecutive points to update

        """Choose which x value to update & update"""
        point_selected = random.randint(0,N-1)
        updateVal = random.uniform(-1,1)
        #for i in range(n_updated):
       #     x_values
        x_values[-1,point_selected:point_selected+n_updated] += updateVal
        if point_selected+n_updated > N:
            x_values[-1,0:(point_selected+n_updated)%N] += updateVal

        """Change in action"""
        changeInAction = m/(2*epsilon)*((x_values[-1,point_selected%N]-
x_values[-1,(point_selected-1)%N])**2
+(x_values[-1,(point_selected+n_updated)%N]-
x_values[-1,(point_selected+n_updated-1)%N])**2-
(x_values[-2,point_selected%N]-x_values[-2,(point_selected-1)%N])**2-
(x_values[-2,(point_selected+n_updated)%N]-
x_values[-2,(point_selected+n_updated-1)%N])**2)
        for j in range(n_updated):
            changeInAction += 0.5
*k*epsilon*(x_values[-1,(point_selected+j)%N]**2-
x_values[-2,(point_selected+j)%N]**2)

        """Test update"""
        if changeInAction <= 0:
            action += changeInAction
            x_values = np.delete(x_values,-2,0)
        else:
```

```
            prop = np.exp(-changeInAction)
            if random.random() <= prop:
                action += changeInAction
                x_values = np.delete(x_values,-2,0)
            else:
                x_values = np.delete(x_values,-1,0) # changes updated
point back to original value
        means[i] += np.mean(x_values)
    mean_split = np.split(means,10)
    mean_split = np.delete(mean_split,0,axis=0)
    mean_split = np.hstack(mean_split)
    return mean_split


positions2 = updateX(beta, N, m, k)
plt.hist(positions2, bins=100, density=True)

#%%
from scipy.stats import norm

"""plot figure"""
fig = plt.figure(figsize=(13,9))
ax1 = fig.add_subplot(111)
ax1.grid()
ax1.set_xlabel('position x',fontsize =15)
ax1.set_ylabel('probability density',fontsize=15)
ax1.set_xlim(-3,3)
ax1.tick_params(axis='both', labelsize=15)

ax1.hist(positions2, bins=100, density=True)

x_axis = np.arange(-3,3,0.01)

def func(x):
    return (np.sqrt(1/np.sqrt(np.pi))*np.exp(-x**2/2))**2

ax1.plot(x_axis, func(x_axis))
```

Welcome to Class Notebook

Thursday, June 9, 2022 1:24 PM

Your OneNote Class Notebook is a digital notebook for the whole class to store text, images, handwritten notes, attachments, links, voice, video, and more.

Each notebook is organized into three parts:

1. **Student Notebooks** — A private space shared between the teacher and each individual student. Teachers can access every student notebook, while students can only see their own.
2. **Content Library** — A read-only space where teachers can share handouts with students.
3. **Collaboration Space** — A space where everyone in your class can share, organize, and collaborate.

How to make the most of Class Notebook in your Class Team:

Start adding materials or collaborating in your Class Notebook today. Use the menu to the left to open or add new pages.

Work in groups. If you've added channels to your class team, use the Notes tab in those channels to continue working together in real time. Each channel connects to its own section in the Collaboration Space.

Go full-screen. Launch Class Notebook in full-screen to get more done. Select the double arrow in the upper right of your Microsoft Teams app to expand the window.

Access more features. Select Open in OneNote to launch Class Notebook in your OneNote app and access more features.

Learn more. Check out the FAQ: Class Notebook in Microsoft Teams page to learn more

FAQ: Class Notebook in Microsoft Teams

Thursday, June 9, 2022 1:20 PM

Where can I get more Class Notebook questions answered?
OneNote Class Notebook help center

Questions? Need assistance?
File a support ticket at: https://aka.ms/EDUSupport

Where can I find OneNote and Class Notebook training resources?
A few short interactive courses on the Microsoft Education Center:

- OneNote Class Notebook: A teacher's all-in-one notebook for students
- Getting Started with OneNote - Microsoft in Education
- OneNote: your one-stop resource - Microsoft in Education

Experiencing permissions issues with your Class Notebook?
Troubleshoot notebook permissions

Where do I find the settings to manage my Class Notebook?
Go into your Class Notebook in Teams and click the Class Notebook toolbar, and then click Manage Notebook. Teachers can edit sections, copy a notebook link, lock the Collaboration Space, or create a Teacher-only section group here.

To manage Collaboration Space permissions or generate parent and guardian links, open Class Notebook in OneNote and then select Manage Notebooks.

Join the conversation on social media:
Twitter: @OneNoteEDU and @msonenote
Facebook: OneNote