Project A

Simulating a Physical System

Introduction

You have already seen a number of examples using randomization to solve problems and to generate experimental results. In the percolation project, grids were generated according to a specified probability distribution to experimentally determine the percolation probability. This approach is an example of a Monte Carlo method, an algorithm that relies on repeated random sampling to compute results. Monte Carlo methods are often used for simulating physical and mathematical systems when other methods are impossible or computationally infeasible.

Objective and Background

The objective of this project is to use the Monte Carlo "demon algorithm" to estimate parameters (for example, temperature) in a physical system, specifically an ideal gas with moving molecules. We could simulate such a system by computing the pairwise interactions among all molecules at every time step in the simulation, but that approach would be computationally intractable. Moreover, in these systems we're not typically interested in the specific behavior of each individual molecule, but rather in the overall system behavior. For example, we want to know the expected distribution of molecule velocities in a gas (e.g., to understand its temperature), not which molecules have collided with which other molecules.

The key concept behind the demon algorithm is that any simulation must both preserve the total energy of the system (energy is conserved) and insure that no particle winds up with negative kinetic energy. The algorithm proceeds by choosing a molecule at random and proposing to make a small random change to its energy (e.g., to simulate the effect of a collision). To compensate for the change in energy of the molecule / particle, the opposite change is made to a special "demon" molecule, thus conserving the total energy of the system. Note that the demon molecule is never chosen at random for this change; it only acts to store the difference between the system energy and the total energy.

Before describing the demon algorithm in detail, we give a brief description of the system to be simulated. In the **ideal gas**, we want to study the velocity of molecules in the gas. This average velocity gives a measure of the temperature of the gas. Consider the state of the gas in an insulated container at an instant of time. Each of its molecules has some velocity that remains constant until the next collision involving the molecules. Because the molecules move quickly, on average, many of them will have experienced collisions during a short time span. Consequently, most of the molecules will have substantially different velocities than they originally did. This process can, in principle, be simulated directly. In practice, doing so is computationally too expensive or is infeasible (e.g., one step of the situation could mean solving n differential equations, with each equation having n^2 terms).

Sketch of Demon Algorithm

The demon algorithm is a common technique for simulating selected parameters of a complex system by maintaining a given macroscopic (total) energy, while randomly changing the microstate of the system. If a trial change would result in a reduction of energy in the system, additional energy is given to a **demon** (fictitious). If the change would result in additional system energy, the difference is taken from the demon (assuming it has sufficient energy). In our simulation, making a trial change will refer to changing a particle's velocity.

Initialize the system state. The sum of the system energy and the demon energy always equals the specified total energy. The demon energy must always be non-negative.
Execute one simulation step. One simulation step consists of making a specified number of trials, with each trial doing the following: Make a change to the state of the system and record the change in system energy as deltaEnergy. If the change is legitimate, i.e., if the demon energy resulting from making the change is non-negative, the change is accepted and we set
demonEnergy = demonEnergy - deltaEnergy
systemEnergy = systemEnergy + deltaEnergy
Update any parameters to be maintained during the simulation.
Repeat making trials until one simulation step is complete.
After one simulation step, update any additional parameters. Execute as many simulation steps as specified.
It is not obvious that this process can effectively simulate a physical system. In fact, there is no rigorous proof that it does. In practice, however, this algorithm works very well. The first part of the project uses the demon algorithm in an ideal gas simulation.

Ideal Gas Simulation

An ideal gas is a simple, uniform system: Molecules with energy are bouncing around inside a closed container. The internal molecular dynamics is simple – free motion with occasional intermolecular collisions. When using the demon algorithm for a simulation, we view the result of a collision as a random change of the molecule's velocity. Each molecule has a particular momentum, and collisions exchange part of that momentum.

Consider the velocity of a single molecule over time. Each collision can be thought of as a random change in velocity. Since we are interested in the average velocity of all molecules, the exact number and time of the collisions is not important. Rather, what matters is the total change in velocity over the time period we are considering. The demon algorithm takes advantage of this idea. We choose a total energy for the system (view it as the temperature). One step of the simulation consists of a number of trials. One trial perturbs the velocity of a random molecule by a random amount. The change is accepted if system and demon energy are less than the total energy. The difference between the system energy and the total energy is held by the algorithm's namesake, the *demon*.

Your first task is to write an ideal gas simulator that uses the demon algorithm. Download the demon.py file to begin your work. The ideal gas simulator should be written in a function that is already defined for you as ideal_gas:

```
def ideal_gas(N, totalEnergy, steps, state = 1, visuals = True):

    ...
```

The arguments to the function are as follows:

N: The number of molecules in the system
totalEnergy: The total amount of energy. That is, the energy of the N molecules + the demon energy.
steps: The number of steps to execute. Each step consists of N demon trials, as described below.
state: The initial state of the molecules, which can be one or two.
visuals: Whether or not to draw visuals for the simulation, which are described in the next section.

The energy of the system state is fully described by the molecular velocities. Recall that the energy of a moving molecule is $\frac{1}{2}mv^2$, where m is the mass of the molecule and v is the velocity. For our simulation, assume unit mass (i.e., $m = 1$). Steps 1 and 2 of the demon algorithm now look as follows. Your function should do the following:

Initialize the system state according to the state argument passed to the function. Since the function only accepts a number of molecules to simulate, you must set initial velocities for each molecule. The state variable determines whether or not the demon begins with all or none of totalEnergy:
If state == 1, we will give each molecule the same velocity. Since the energies given the velocities must sum up to totalEnergy (our demon initially has no energy), this means that each molecule starts with a velocity of sqrt(2 * totalEnergy / N).
If state == 2, we simply set all molecule velocities to zero and give the demon all of the energy.
Create a loop that will execute step iterations. Each iteration executes N trials, so we describe one trial:
Choose a random molecule (remember, the demon cannot be chosen here)
Generate a random number between -Dv and Dv, where Dv is the velocity corresponding to 10% of the total system energy divided by N. Thus, Dv = sqrt(2 * totalEnergy / N / 10).
Calculate the change in energy. Since we use unit mass (m = 1), the change in energy is deltaEnergy = $vNew^2 / 2 - vOld^2 / 2$.
If the change in energy is valid, then accept the change in velocity for that molecule. Remember that the change in energy is valid as long as the resulting demon energy is non-negative. Because not every change is legal, every iteration may not change N molecules, even though it executes N trials.
The demon and system energies must be updated to use in the next trial. However, since we must generate visuals for the next part of the project, we need to record these values in a list to keep track of the value at each iteration (not trial).
Calculate the average system energy and return it.

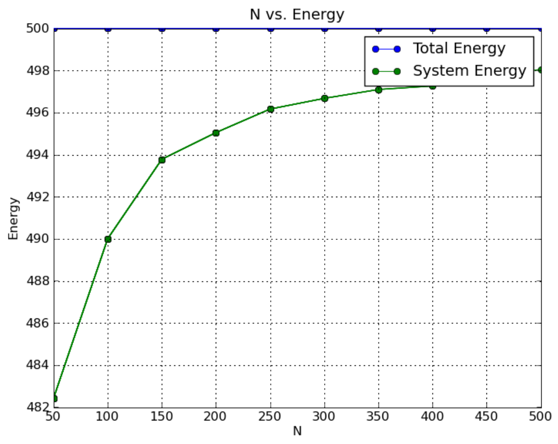
Experiments for the Ideal Gas Simulation

First, write and test the function ideal_gas. Note that in this project you will be handing in the function and the experimental work at the same time. Test the function ideal_gas with small values of N and small values of step. You won't know that you are computing the correct quantities until you generate the plots described next. Test the function with both initial system states.

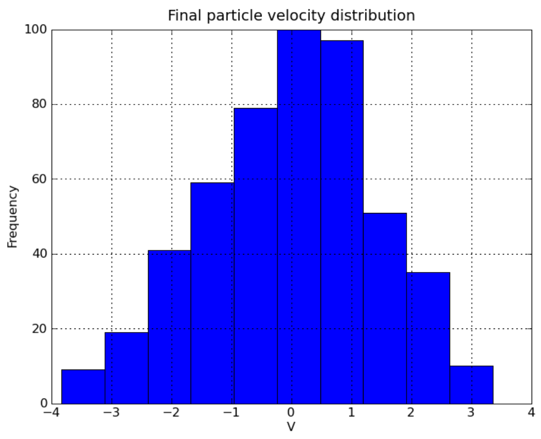
Your experimental work should consider systems with N= 50 to 500, with an increment of 50. For each N, run the function ideal_gas with steps = 3000 and totalEnergy = 500. Each experiment should be done for each of the two initial system states.

The simulation can use MatPlotLib or VPython. **Warning: If you use VPython, you should remove all traces of MatPlotLib from the skeleton code. The two have not been known to play well together.** A starting point for the varying values of N is provided in the main of the skeleton code. This part of the project requires you to write code to draw two graphs and two histograms.

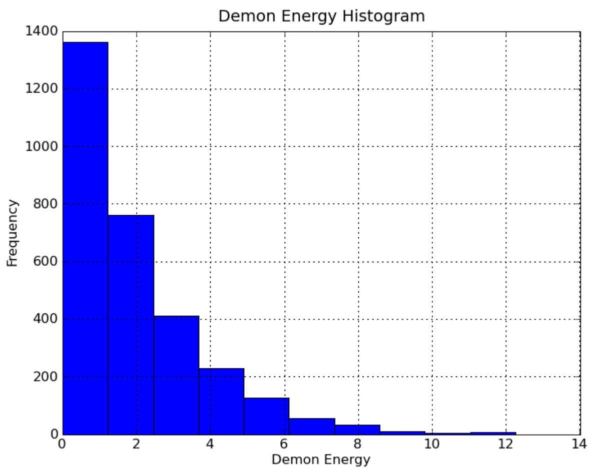
The main of the simulation uses the values returned by ideal_gas to generate the graph **N_versus_Energy.** The value returned by the function ideal_gas represents the average system energy of a simulation with particular value of N. You should graph the average system energy on the y-axis and N on the x-axis. In the same plot, show the total system energy in a different color (in the required experiment it is 500 for every value of N). You can generate the plot after the entire simulation or generate it incrementally after ideal_gas returns a value. Here is an example plot. nvsenergy1.png



The ideal_gas function should generate two histograms and one graph. Whether and when these plots are generated is determined by the fifth parameter (visuals) of ideal_gas. For example, ideal_gas(N, 500, 3000, 2, N==500) will generate plots only when N=500 and start in initial state 2. Note that omitting the 5th argument will plot every time ideal_gas is called.
Histogram **Final_Molecule_Velocity** uses the values of the N molecule velocities after *steps* iterations of ideal_gas (i.e., the values just before ideal_gas returns). Plot the velocities on x-axis and their frequencies on y-axis. You should use the signed velocities and 0 is thus in the center of the x-axis. You should experiment with the most appropriate number of bins. Here is an example plot. velocitydistr1.png



Histogram **Demon_Energy** generates a histogram of the observed demon energy. The plot is generated after the steps of ideal_gas are completed. The demon energy is recorded after each simulation step (using a list or array of size step). Show the demon energies on the x-axis and their frequencies on the y-axis. Again, select an appropriate number of bins. Here is an example plot. demonenergy1.png



Graph **Demon_Energy_Time** shows the steps demon energies over time. The graph has values from 1 to steps on the x-axis and the corresponding demon energies on the y-axis. Make sure to show these 3000 steps in an effective way.

If you are using VPython, you can show a histogram plot as soon you have data needed. If you are using Matplotlib, the interactive option allows showing results during the computation (See Interactive Matplotlib):

Interactive Matplotlib

Normally, matplotlib will not show plots until you call pylab.show(). However, if you call pylab.ion() before plotting, matplotlib will show the window immediately. For best results, decorate (set title, axis labels, etc) after calling pylab.plot().

Example code:

```
import pylab
```

```
xs = []
ys = []
pylab.ion()
for x in range(10):
        xs.append(x)
        ys.append(x ** 2)
        pylab.plot(xs, ys, 'b-o')
        pylab.pause(1)
pylab.title("curve of x squared")
pylab.xlabel("x")
pylab.ylabel("x ** 2")
pylab.show()
pylab.pause(10)
pylab.close()
```

Discussion

You need to hand in a discussion of your experimental results. The following questions should be addressed:

In graph N_versus_Energy, how close is the actual system energy to the total energy? How does it change as N changes? How do the plots differ for the two different initial system states?
Using histogram Final_Particle_Velocity, what kind of distribution do you judge the velocities to come from? How close is the molecule velocity to the initial velocity of sqrt(2*totalEnergy/N) used in initial state 1?
Using histogram Demon_Energy, what kind of distribution do you judge the demon energies to come from?
Comment of the differences and similarities of graph Demon_Energy_Time for different values of N and the two initial states.

Submission Instructions

Submit a program including the function ideal_gas, the main function driving the simulation in demon.py. Your program will contain the loop for the values of N considered. The rest of the code will differ significantly between students depending how the visualization is handled (such as building a user interface for displaying menus and the above graphs). You also need to include a PDF file (*.pdf) that includes a final report with the discussion of your experiment and answers the questions listed (The format of the final report is shown below). These files should all be placed into one folder in a compressed file (e.g. 5150309000_XXX(team_projA_v3).rar) and submitted to the course representative of your class. The course representatives compress all of them with a *.rar file and upload it in the subdirectory team_projA.

The deadline is due at **9pm** on **Jan. 8, 2016**.

References

Computational Physics. http://en.wikipedia.org/wiki/Computational_physics/

Fredrik Lundh. An Introduction to Tkinter. 1999.

Matplotlib. http://matplotlib.sourceforge.net/

matplotlib-1.2.0.win32-py3.2.exe. http://www.lfd.uci.edu/~gohlke/pythonlibs/

Molecular Dynamics. http://en.wikipedia.org/wiki/Molecular_dynamics/

Monte Carlo Method. http://en.wikipedia.org/wiki/Monte_Carlo_method/

Numpy: The Fundamental Package for Scientific Computing with Python. http://numpy.scipy.org/

numpy-MKL-1.6.2.win32-py3.2.exe. http://www.lfd.uci.edu/~gohlke/pythonlibs/

Tkinter. http://wiki.python.org/moin/TkInter/

Tkinter Documentation. 2010. http://wiki.python.org/moin/TkInter

VPython: 3D Programming for Ordinary Mortals. http://vpython.org/

VPython-5.74.win32-py3.2.exe. http://www.lfd.uci.edu/~gohlke/pythonlibs/

Appendix: Final Report Format

Team Name: Team Leader:

Date:

1 Prototype System Introduction

1.1 Functions

1.2 Running Environment

Windows 7

1.3 Developing Environment

PyScripter 2.5.3

…

2 Task Allocation

The tasks for each team member.

3 System Architecture

3.1 User Interface Component

Graph and explanation.

3.2 Simulation Component

Graph and explanation.

3.3 Visualization Component

Graph and explanation.

4 Algorithm Description

4.1 User Interface Component

Flowchart and explanation.

4.2 Simulation Component

Flowchart and explanation.

4.3 Visualization Component

Flowchart and explanation.

5 Demo and Testing Result

5.1 Screenshots

User interface and every operation.

5.2 Testing Procedure, Data and Result

Explanation, testing data and result table, and result analysis

6 Conclusion

The discussion of your experiment and answers the questions listed above.

What is your research result? What are your experience and lesson on this project?