Computational Physics With Python

Dr. Eric Ayars

California State University, Chico

Chapter 6

Monte Carlo Techniques

Monte Carlo is a small town on the French Riviera most famous for its casino. Because of this fame in the field of random events, the name "Monte Carlo techniques" is given to random (or pseudo-random) methods of solving problems on a computer.

Here's an example. Say you have an odd shape, and you want to find its area. One Monte Carlo technique for finding this area would be to surround the shape with a shape of known area (a rectangle, perhaps) and then generate a large number of random points within the known area. The area of the shape is (approximately) the fraction of the points that fall within the shape, times the known area surrounding the shape. (See figure 6.0.)

For this method to work, it's necessary that our random numbers meet three criteria:

1) They must be uniformly distributed. Obviously, if the points were distributed about some normal curve, then we'd see more points in the central area of figure 6.0 and we'd get a poor answer.

2) They must be uncorrelated. For example, if our random numbers in figure 6.0 were correlated in such a way that a low one was most often followed by a high one, and vice-versa, then the points in the figure would trend towards the upper left and lower right of the figure, and this would skew our results.

3) We need a *lot* of points. Our percent error by this method is going to scale roughly as $\frac{1}{\sqrt{N}}$, so for a mere three significant figures in the answer, we need a million data pairs.

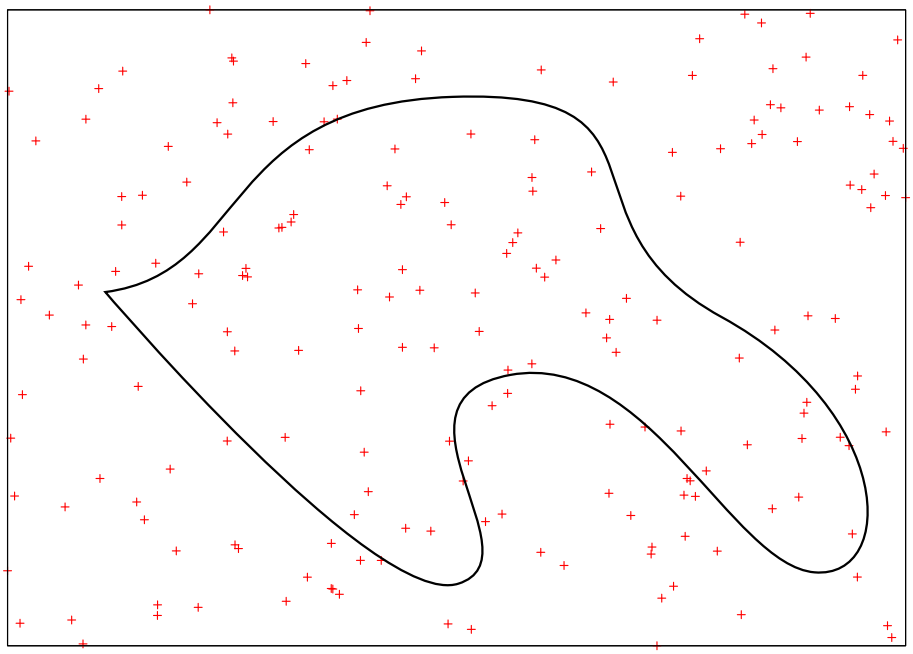

Figure 6.0: A Monte Carlo method of finding the area of an odd-shaped figure. The area of the shape is approximately 71/200 × the area of the bounding rectangle.

6.0 Random Numbers

Generating random numbers is inherently difficult. For starters, there's the question of what is random in the first place! (See figure 6.1.) Add to this the problem that people generally can't create randomness on their own. If you asked the average student to draw 200 random dots on a rectangle, they'd end up much more evenly separated than the points on figure 6.0. Adding to the difficulty with Monte Carlo techniques is the issue of trying to generate something random with computers, which are inherently non-random devices. There are physical sources of random numbers, such as the time between decay events in a radioactive sample, but it's hard to collect enough of these fast enough for what we need.

We generally give up on truly random numbers, and use "Pseudo-Random" sequences of numbers instead. If the sequence of numbers meets the criteria on page 123, then they'll do the job even if they aren't technically random.

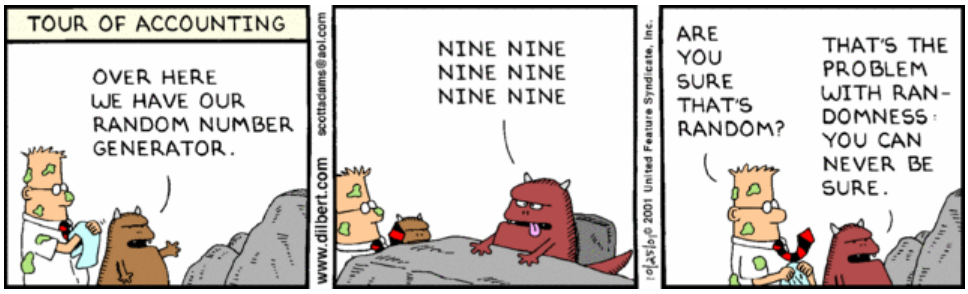

Figure 6.1: ©Scott Adams

One example of a pseudo-random number generator is the linear congruential generator:

$$x_{n+1} = (ax_n + b) \mod c \tag{6.1}$$

This pseudo-random number generator takes a "seed" x_0 and generates a series of numbers which, depending on the choices of a, b, and c, can meet the necessary criteria for Monte Carlo techniques.

Example 6.0.1
Let $a = 5$, $b = 3$, and $c = 8$. For a starting "seed" $x_0 = 1$, this generates the sequence 0, 3, 2, 5, 4, 7, 6, 1. At this point, the sequence repeats itself.

As you can see in example 6.0.1, c needs to be fairly large in order to generate a long sequence: in fact the longest possible sequence for the linear congruential generator is of length c.

One set of parameters that was used heavily in the 70's and early 80's is "Randu": $a = 65539$, $b = 0$, and $c = 2^{31}$. These parameters have the advantage of being easy for the computer to calculate, so it was possible to generate large sequences quickly. There's a problem, though, which was not discovered until much later: the Randu sequence is highly correlated! If you use Randu to plot random points in three dimensions, every point falls on one of 15 parallel planes. As you may well imagine, this belated discovery has cast doubt on many previously-calculated Monte Carlo results, and people are still trying to evaluate the damage.

Python comes with a pseudo-random number generator called the "Merseinne Twister". This is a very fast generator, and it has been extensively

tested by the mathematical community and given a clean bill of health.[1] Rather than use our own pseudo-random number generator and risk pitfalls such as Randu, we'll use this one.

To access these random-number routines, import the "random" package:

```
import random

random.random()          # generates a random float between 0 and 1
random.uniform(a,b)      # generates a random float on the range [a,b).
random.choice(list)      # returns a random element from the list
random.gauss(mu,sigma)   # returns points with gaussian distribution
                         # centered on mu, with width sigma
random.randint(a,b)      # random integer on range [a,b]
```

These are the most useful functions within random, at least for the purposes of this course. There are other functions as well: see the Python documentation.

6.1 Integration

Let's go back now and take a closer look at the first integration example in this chapter. The first question you should be asking is, "Why would anyone do this?" After all, in order for the program to determine whether a random point is inside or outside of the shape, the program will have to have an equation for the shape. If the program has an equation for the shape, shouldn't it be possible to find the area by numerically integrating, using the techniques from section 2.1? And shouldn't direct numerical integration be faster, by several orders of magnitude, than randomly selecting several million points?

These objections are entirely valid, for problems with low dimensionality. At higher dimensions, though, direct numerical integration becomes more difficult. The number of function calculations for numeric integration grows exponentially with dimension. Monte Carlo integrations also increase in difficulty with dimension, but the increase is linear. This means that an integral that takes 10 function calls to calculate in one dimension would take on the order of 10^{12} function calls in 12 dimensions. The same integrals might take a million function calls in one dimension for Monte Carlo integration, and a mere 12 million in 12 dimensions. Monte Carlo wins by a large margin.

[1] The only caveat is that the Merseinne Twister is not appropriate for cryptographic use, as it is possible to calculate the exact state of the system given a large enough sample of the output. This makes no difference to this course!

"But where," you may well ask, "does one get a 12-dimensional integral?" It takes only two particles. Each particle has three position components (x, y, z) and three velocity components (v_x, v_y, v_z). With two particles, that's 12 degrees of freedom.

Monte Carlo methods are still slow, though. (Actually, the problems to which we apply Monte Carlo methods are slow.) It's definitely worthwhile to carefully consider any symmetries in the problem you're facing and use them to your advantage. For example, if the shape you are integrating is symmetric about some axis, find the area on one side only, then double the result. Make sure your known area is not bigger than it needs to be, and by all means devise your test for whether the points are within the area or not to be as efficient as possible.

Example 6.1.1
Use Monte Carlo integration to find the volume of a hemisphere of radius 1.

We can use the symmetry of the problem to find the volume of one quarter (the positive $\{x, y, z\}$ quarter) of the hemisphere, then multiply by 4 to get our final answer. For the known volume surrounding our unknown piece, let's use the unit cube: x, y, and z range from 0–1. This range of values corresponds exactly to the evenly-distributed random numbers given by random.random(). Here's code to do the job:

```
from random import random
from math import sqrt

# Since the radius is 1, we can leave it out of most calculations.

N = 1000000                    # number of random points in unit cube
count = 0                      # number of points within sphere
for j in range(N):
    point = (random(), random(), random())
    if point[0]*point[0] + point[1]*point[1] + point[2]*point[2] < 1:
        count = count+1

Answer = float(count)/float(N)
# Make sure to use float, otherwise the answer comes out zero!
# Also note that in this case the volume of our "known" volume (the unit
# cube) is 1 so multiplying by that volume was easy.

Answer = Answer * 4            # Actual volume is 4x our test volume.
```

```
print '''The volume of a hemisphere of radius 1
is %0.4f +/- %0.4f.''' % (Answer, 4*sqrt(N)/float(N))
```

The program gives a volume of 2.095 ± 0.004, which is consistent with the known value $\frac{2}{3}\pi r^3$.

6.2 Problems

6-0 Calculate $\int_0^\pi \sin(x)\,dx$ using Monte Carlo techniques. Report your uncertainty in the result, and compare with the known result.

6-1 Find the volume of the intersection of a sphere and a cylinder, using Monte Carlo techniques. The sphere has radius 1 and is centered at the origin. The cylinder has radius 1/2, and its axis is perpendicular to the x axis and goes through the point ($\frac{1}{2}$, 0, 0). (See figure 6.2.) Report your uncertainty, also.

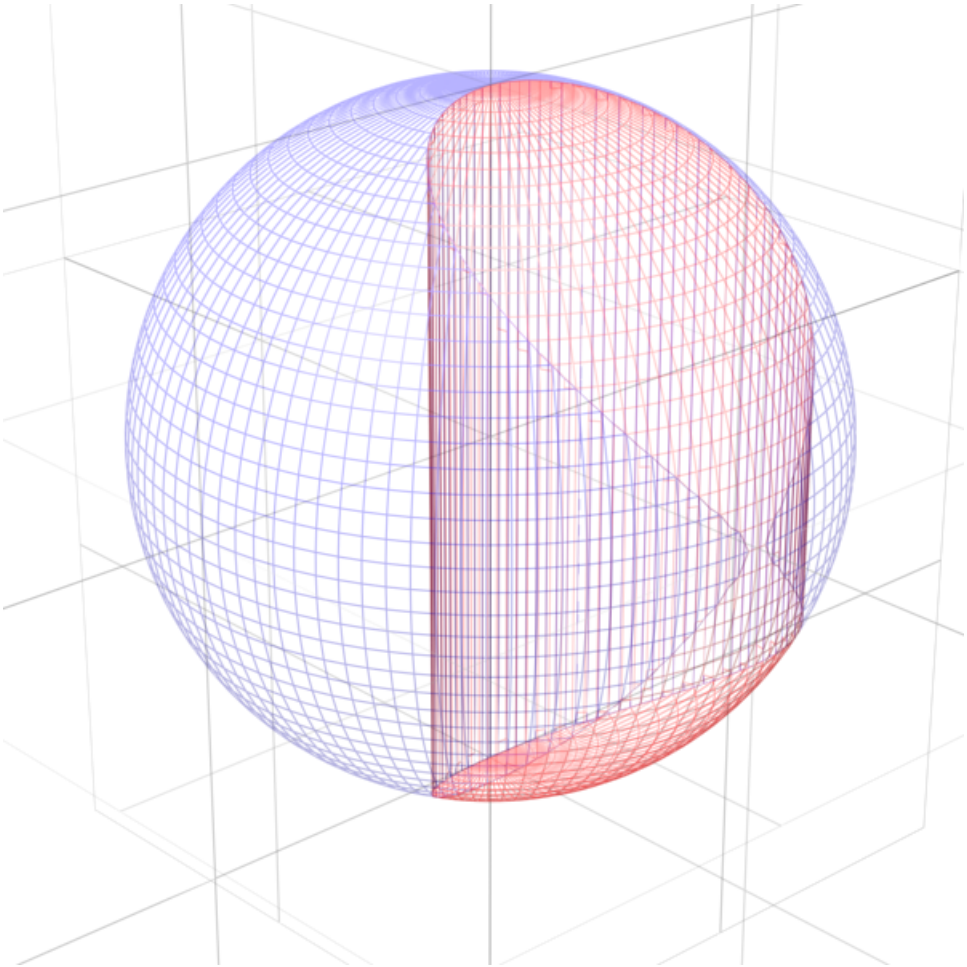

Figure 6.2: Figure for problem 1

6-2 The "volume" of a 2-sphere $x^2 + y^2 = r^2$ (AKA a "circle") is $(1)\pi r^2$. The volume of a 3-sphere $x^2 + y^2 + z^2 = r^2$ is $\left(\frac{4}{3}\right)\pi r^3$. The equation for a 4-sphere is $x^2+y^2+z^2+w^2 = r^2$. We can guess, by extrapolation from the 2-dimensional and 3-dimensional cases, that the "volume" of a 4-sphere is $\alpha\pi r^4$. Use Monte Carlo integration to estimate α.

Chapter 7

Stochastic Methods

Take a still beaker of water, and drip one drop of dye into the center. Over time the dye spreads out, and eventually distributes itself evenly throughout the water. How would we model this behavior computationally?

The direct method would be to calculate $\vec{x}$ and $\vec{p}$ for each molecule in the beaker, but this poses some computational difficulty. Assuming 10^{23} molecules, with six degrees of freedom each, and a 4-byte floating-point number for each degree of freedom, this direct method would require 2.4×10^{12} Tb just to store the state of the system at any instant.[1] In addition to the storage problem, the direct method has limited usefulness even if it could be implemented: We could calculate the time it would take for the dye molecules to evenly spread throughout the water, but then if we wanted to know how long it would take to distribute through twice the volume we'd have to do it again! The direct method doesn't give us any *understanding* of what's going on.

Instead of the direct method, we use *stochastic* methods. The fundamental idea behind these methods is that *large ensembles act in an "average" way even if individual elements are random.* We lose the details about each molecule, and only learn the behavior of the ensemble.

7.0 The Random Walk

We'll start with the simplest method, the random walk in one dimension. Start with a drunken frat boy standing on the sidewalk outside of the Bear at about 2AM on a Saturday morning. He can take a step east, or a step

[1] The "volume" of the internet was estimated to be merely 7.5×10^3 Tb in the fall of 2006.

west. For the sake of this model, we'll assume that the probability of either direction is 0.5, and the step lengths are all the same. The question we'd like to answer is, "how fast, on average, does this random walk move the frat boy away from the Bear?"

Here's a program that models his motion:

```
#!/usr/bin/env python

"""
Fratboy.py
Program to model 1-D random walk
"""

from random import choice
from pylab import *

N = 200       # number of steps

# set up storage space
x = zeros([N])
t = range(N)

# Do the walk
for i in range(1,N):
    if choice(['forward', 'back']) == 'back':
        # take a step back
        x[i] = x[i-1] - 1
    else:
        # take a step forward
        x[i] = x[i-1] + 1

RMS = array([sqrt(i*i) for i in x])

plot(t,x,'b-')
plot(t,RMS, 'g-')

show()
```

This program only tells us *one* walk, though. We need to know the *average* behavior of a drunk frat boy. To calculate this average behavior, we take several thousand non-interacting drunk frat boys, or —since non-interacting drunk frat boys don't exist[2] — we run the simulation several thousand times

[2] As has been determined experimentally, drunk frat boys in groups tend to all move in the same general direction, and burn furniture in the middle of the street. While random

to determine the average RMS displacement. Here's a program that does just that:

```
#!/usr/bin/env python

"""
fratboy_average.py
Program to model AVERAGE 1-D random walk
"""

from random import choice
from pylab import *
from scipy.optimize import curve_fit

def power(x,a,b):
    return a*x**b

steps = 200      # number of steps
boys = 2000      # number of fratboys

# set up storage space
x = zeros([steps])
t = range(steps)
x_sum = zeros([steps])
x2_sum = zeros([steps])

# Do the walks
for j in range(boys):
    for i in range(1,steps):
        if choice([0,1]):
            x[i] = x[i-1] - 1
        else:
            x[i] = x[i-1] + 1
    # add x, x^2 to running sums
    for i in range(steps):
        x_sum[i] = x_sum[i] + x[i]
        x2_sum[i] = x2_sum[i] + x[i]**2

# rescale averages
x_avg = [float(i)/float(boys) for i in x_sum]
RMS = [sqrt(float(i)/float(boys)) for i in x2_sum]

xlabel("Time (Step number)")
```

in some sense, this is not the random behavior we're interested in studying in a physics course.

```
ylabel("Average and RMS position (Steps)")
plot(t,x_avg, 'b-')

plot(t,RMS, 'g-')


# Check least-squares fit, see what the power dependence is.
# I assume that the RMS displacement goes as D = A*t^B
popt, pcov = curve_fit(power, t, RMS)

print "Power fit: y(t) = A * t^B:"
print "A = %f +/- %f." % (popt[0], sqrt(pcov[0,0]))
print "B = %f +/- %f." % (popt[1], sqrt(pcov[1,1]))

# Plot the curve fit on top of that last graph
plot(t, power(t, popt[0], popt[1]))
show()
```

An RMS displacement graph for this program is shown in figure 7.0. The RMS displacement goes as the square root of the number of steps. There's a good mathematical reason for this: The displacement after n steps is given by

$$x_n = \sum_{i=1}^{n} s_i$$

where s_i is the step length for step i, which in this case could be either ± 1. The square of the displacement is given by

$$x_n^2 = \left(\sum_{i=1}^{n} s_i \right) \left(\sum_{j=1}^{n} s_j \right) = \sum_{i=1}^{n} s_i \sum_{j=1}^{n} s_j \ .$$

We can break the double sum into two portions:

$$x_n^2 = \sum_{i=j} s_i s_j + \sum_{i \neq j} s_i s_j \ .$$

The second term ($i \neq j$) averages to zero, since the direction of the step is random. We're left with just the first term, which simplifies nicely to

$$\left\langle x_n^2 \right\rangle = \sum_{i=1}^{n} s_i^2 = n \left\langle s^2 \right\rangle \ .$$

In our present case $s_i = \pm 1$, so

$$\langle x_n \rangle = \sqrt{n}$$

as is consistent with the program output.

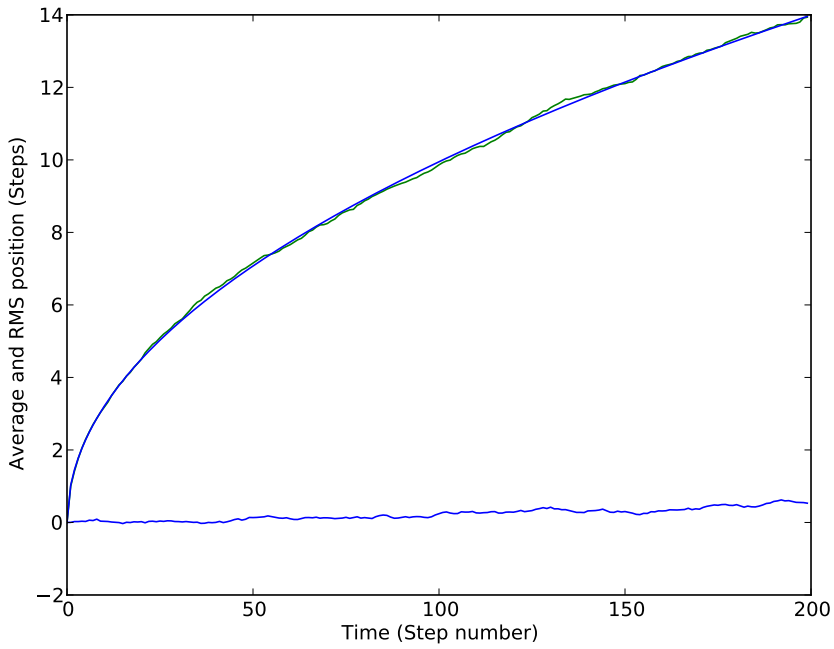

Figure 7.0: Average RMS Displacement for a one-dimensional random walk with a uniform stepsize of 1. This graph shows the average for 2000 walks. A least-squares fit is also shown for $x(t) = At^B$, where $A = 0.994 \pm 0.002$ and $B = 0.4991 \pm 0.0005$.

7.1 Diffusion and Entropy

We can use the random walk to model the diffusion of the drop of dye in water from the beginning of this chapter. For ease of display, the following program does only 2-D diffusion rather than 3-D, but the concept is the same.

```
#!/usr/bin/env python
"""
diffusion.py

Random-walk model of diffusion in a 2-d environment.
Starts with 400 particles in a square grid centered at (100,100).
At each step, the program picks each particle and moves it (or not)
```

```
one integer step in the x and y directions. If the move would
take the particle beyond the boundary of space (200x200), then
the particle bounces off the wall and moves the other direction.

The program plots the positions of all particles after each step.

Call the program with one argument: the number of steps to take.
"""

import sys
from pylab import *
from random import randint  # randint(a,b) picks a random integer in
                           # the range (a,b), inclusive.

# Allow animation
ion()

# set up graph window
figure(figsize=(10,10))

# Define droplet coordinates (all droplets) to be at point 100,100.
atoms = ones([400,2])*100

# show initial configuration
line, = plot(atoms[:,0], atoms[:,1], 'ro')
xlim(0,200)
ylim(0,200)
draw()
wait = raw_input("Press return to continue")

# How many steps to take?
N = int(sys.argv[1])

for i in range(N):
    # Go through all atoms
    for j in range(400):

        # Move each atom (or not) in the x and/or y direction.
        atoms[j,0] += randint(-1,1)
        atoms[j,1] += randint(-1,1)

        # Check for boundary collision
        x,y = (atoms[j,0], atoms[j,1])
        if x == 200:
            atoms[j,0] = 198
```

```
            elif x == 0:
                atoms[j,0] = 2
            if y == 200:
                atoms[j,1] = 198
            elif y == 0:
                atoms[j,1] = 2

        # See how things look now.
        line.set_xdata(atoms[:,0])
        line.set_ydata(atoms[:,1])
        draw()

wait = raw_input("Press return to exit")
```

It takes awhile (4-5 thousand iterations), but the "dye molecules" in this simulation eventually spread evenly throughout the container. (The simulation runs much faster if you don't replot at every step, of course!)

One characteristic of the simulation is that the system becomes more disordered with time. At the beginning, all of the molecules are on a single point at the center of the simulation, and at the end they're scattered about like legos on my son's bedroom floor after a hard day of playing. We quantify this "amount of disorder" as *entropy*.

In previous physics classes, you probably dealt only with *change* in entropy,

$$\Delta S = \frac{\Delta Q}{T} \, .$$

For a system with discrete states such as this diffusion simulation, it's possible to define the entropy exactly

$$S = -k_B \sum_i P_i \ln P_i \tag{7.1}$$

where P_i is the probality of finding a particle in state i. We can divide the simulation up into smaller segments (see figure 7.1) to estimate the values P_i.

Each grid square in figure 7.1 represents one state i. The probability (or relative frequency) of particles in each state is the number of particles per square, divided by the total number of particles. For example, in the final state (on the right in figure 7.1) the value of P_i for the top left state is 5/400, since there are 5 particles in that particular box and 400 particles total.

The entropy of the initial state is zero for this particular choice of states. The relative frequency for each state is zero for all but the center state, since

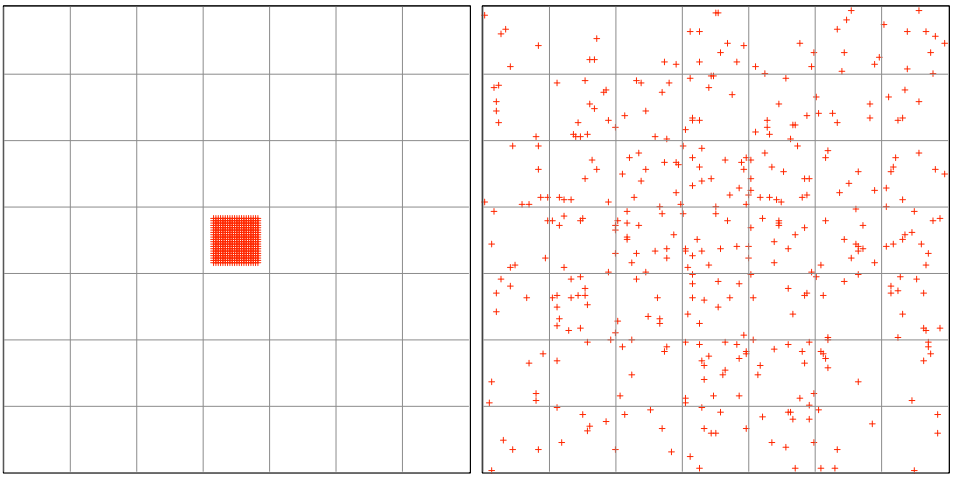

Figure 7.1: Output of the program diffusion.py, initially and after 5,000 steps. The light grid overlaid on both graphs is for calculation of entropy by equation 7.1. In the initial state $S = 0$, and in the final $S = 195$.

they're all empty. So $P_i = 0$ in equation 7.1 for all but the center square. In the center square, $P_i = 1$ since all of the particles are there, and $\ln 1 = 0$ so the total entropy is $S = 0$.

The entropy of the final state is calculated in the same way, but of course the number of particles in each box is not zero. Writing a program to do this is left as an exercise, in problem 2.

7.2 Problems

7-0 We showed in section 7.0 that the RMS displacement of the drunk fratboy goes as $\sqrt{\text{time}}$. That was done for a one-dimensional walk, though. Diffusion of dye molecule in a liquid would be a three-dimensional random walk, instead of one-dimensional. What is the time dependence of the RMS displacement for a 3-D random walk?

7-1 What happens in the one-dimensional random walk if the stepsize is also random? Model this situation where instead of randomizing the direction of each step, you pick a step with a random length on the range [-1, 1]. What is the time dependence of the RMS displacement in this case?

7-2 Modify the program diffusion.py (in the /export/312/shared/ directory) so that it makes a plot of entropy vs. time.