10.34: Numerical Methods Applied to Chemical Engineering
Prof. K. Beers
Solutions to Problem Set 8: Brownian Dynamics and Monte Carlo Simulation

Mark Styczynski, Ben Wang

Problem 1. Brownian motion in an external field

Consider the Brownian motion in the x-direction of a spherical particle of radius R_p and density ρ_p moving in a Newtonian fluid of viscosity μ_f. Stokes' law gives a drag constant of $\zeta = 6\pi\mu_f R_p$. The particle experiences an external potential energy field U(x) such that the force imparted to the particle by the field is –(dU/dx). Let us say that the fluid itself is moving in the x-direction with a velocity v_f. With a random force $F_R(t)$ due to collisions with individual fluid molecules, Newton's second law of motion for the particle is

$$m\frac{dv_x}{dt} = -\zeta\left(v_x - V_f\right) - \frac{dU}{dx} + F_R(t)$$

As shown in class, the mass of a particle, $m = \frac{4}{3}\pi\rho_f R_p^3$, becomes negligibly small compared to the drag constant $\zeta = 6\pi\mu_f R_p$ when R_p is very small. This results in extremely short velocity correlation times $\tau_p = \frac{m}{\zeta}$. If we are concerned only with the observed motion of the particle on scales long compared to τ_p, we can neglect the inertial effects completely by taking the limit m → 0 while holding ζ constant. In this limit, the motion of the particle follows

$$0 = -\zeta\left(v_x - V_F\right) - \frac{dU}{dx} + F_R(t)$$

Upon rearrangement and multiplying by dt, using dx = $v_x$dt, this yields

$$dx = \left[V_f - \zeta^{-1}\frac{dU}{dx}\right]dt + \zeta^{-1}F_R(t)dt$$

As shown in class, in the case where v_f = 0, U(x) = 0, we get the correct statistical properties of the random displacement by simulating the SDE (Stochastic Differential Equation)

$$dx = \left(2D\right)^{\frac{1}{2}} dW_t$$

For a finite time step Δt, we have the simulation algorithm

$$x\left(t+\Delta t\right) - x\left(t\right) = \left(2D\right)^{\frac{1}{2}}\Delta W_t$$

ΔW_t is a random number generated at each time step with

$$\Delta W_t = \left(\sqrt{\Delta t}\right)\theta \qquad P\left(\theta\right) = \frac{1}{\sqrt{2\pi}}e^{\frac{-\theta^2}{2}}$$

ΔW_t is drawn from a normal distribution with a mean of zero and a variance of σ^2 = Δt.

Therefore, to agree with this limiting case, we write the SDE for the particle motion in a moving fluid and an external field as

$$dx = \left[V_f - \zeta^{-1} \left.\frac{dU}{dx}\right|_{x(t)} \right] dt + (2D)^{\frac{1}{2}} dW_t$$

Which yields the simulation algorithm

$$x(t+\Delta t) - x(t) = \left[V_f - \zeta^{-1} \left.\frac{dU}{dx}\right|_{x(t)} \right] (\Delta t) + (2D)^{\frac{1}{2}} \Delta W_t$$

As shown in class, the drag constant and the diffusivity are related by Einstein's relation

$$D = \frac{k_b T}{\zeta}$$

Part 1.A.

Considering the SDE above, we see that if we had no random force, we would have a deterministic velocity of the particle equal to $v_p = V_f - \zeta^{-1} \left.\frac{dU}{dx}\right|_{x(t)}$ *. So, the deterministic (non-random) part of the SDE appears to describe convective motion, and the random part (as we have seen) describes diffusive motion. This appears to suggest that the probability distribution p(t,x) follows a convection/diffusion equation*

$$\frac{\partial p}{\partial t} = -\frac{\partial}{\partial x}\left[v_p p(t,x) \right] + D \frac{\partial^2 p}{\partial x^2}$$

In fact, it is shown in the text that for a system described by the SDE

$$dx = a(t,x)dt + b(t,x)dW_t$$

The probability distribution is governed by a corresponding ***Fokker-Planck equation***

$$\frac{\partial p}{\partial t} = \frac{\partial}{\partial x}\left[a(t,x) p(t,x) \right] + \frac{1}{2}\frac{\partial^2}{\partial x^2}\left\{ \left[b(t,x) \right]^2 p(t,x) \right\}$$

Here, you are asked to perform a number of Brownian dynamics simulations to demonstrate that the probability distribution does indeed follow this convection/diffusion equation in the case of a constant fluid velocity v_f and in the absence of an external field, U(x) = 0.

If we release a particle at x = 0 at time t = 0, the initial condition is p(0,x) = δ(x).

1.A.1. *First, show that the solution of the convection/diffusion equation is*

$$p(t,x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{(2\sigma^2)}} \qquad \mu = V_f t \qquad \sigma^2 = 2Dt$$

Well, note that the question just asks you to "show" that the solution is such. Thus, I'll only be presenting here a demonstration that this is, indeed, the solution; this will be sufficient for full credit for this part. If you were feeling saucy and decided to actually give a proper derivation, then more power to you.

Looking at the equation

$$\frac{\partial p}{\partial t} = -\frac{\partial}{\partial x}\left[v_p p(t,x)\right] + D\frac{\partial^2 p}{\partial x^2} \qquad \text{where } v_p = V_f - \zeta^{-1}\left.\frac{dU}{dx}\right|_{x(t)} = V_f$$

We see that all we need to do is take the individual derivatives and make sure that everything cancels. We'll use some intuition in our grouping of terms: derivatives of p are likely to equal p times some factor, since there is an exponential term in p. So…

$$\frac{\partial p}{\partial t} = p(t,x)\frac{\partial}{\partial t}\frac{-\left(x - V_f t\right)^2}{4Dt} - p(t,x)\frac{1}{2t}$$

$$= p(t,x)\frac{\partial}{\partial t}\frac{-\left(x^2 - 2V_f tx + V_f^2 t^2\right)}{4Dt} - p(t,x)\frac{1}{2t}$$

$$= p(t,x)\left(\frac{x^2}{4Dt^2} - \frac{V_f^2}{4D} - \frac{1}{2t}\right)$$

$$\frac{\partial p}{\partial x} = p(t,x)\frac{\partial}{\partial x}\frac{-\left(x^2 - 2V_f tx + V_f^2 t^2\right)}{4Dt}$$

$$= \frac{-p(t,x)}{4Dt}\frac{\partial}{\partial x}\left(x^2 - 2V_f tx + V_f^2 t^2\right)$$

$$= \frac{-p(t,x)}{4Dt}\left(2x - 2V_f t\right) = p(t,x)\left(\frac{2V_f t - 2x}{4Dt}\right)$$

$$\frac{\partial^2 p}{\partial x^2} = \frac{\partial}{\partial x}\frac{\partial p}{\partial x} = \frac{\partial}{\partial x}\left[p(t,x)\left(\frac{2V_f t - 2x}{4Dt}\right)\right]$$

$$= p(t,x)\left(\frac{-1}{2Dt}\right) + p(t,x)\left(\frac{2V_f t - 2x}{4Dt}\right)^2$$

Substituting those into the convection/diffusion equation, we get

$$p(t,x)\left(\frac{x^2}{4Dt^2} - \frac{V_f^2}{4D} - \frac{1}{2t}\right) = -V_f\left[p(t,x)\left(\frac{2V_f t - 2x}{4Dt}\right)\right] + D\left[p(t,x)\left(\frac{-1}{2Dt}\right) + p(t,x)\left(\frac{2V_f t - 2x}{4Dt}\right)^2\right]$$

We can divide everything through by p(t,x) and simplify the last term to get

$$\frac{x^2}{4Dt^2} - \frac{V_f^2}{4D} - \frac{1}{2t} = \frac{2V_f x - 2V_f^2 t}{4Dt} - \frac{1}{2t} + \frac{4V_f^2 t^2 - 8V_f xt + 4x^2}{16Dt^2}$$

Then we clear out the denominators by multiplying through by 4Dt^2

$$x^2 - V_f^2 t^2 - 2Dt = 2V_f xt - 2V_f^2 t^2 - 2Dt + V_f^2 t^2 - 2V_f xt + x^2$$

Which can be rearranged to yield

$$x^2 - 2V_f^2 t^2 - 2Dt + 2V_f xt = x^2 - 2V_f^2 t^2 - 2Dt + 2V_f xt$$

Which is a true equation, since all terms cancel out to yield 0 = 0.

And yes, it was OK if you used Mathematica or some symbolic calculus program to do all of the manipulations for you, as long as you included that code or screenshots of that program after it did the calculations for you so that we know you didn't just pull it out of thin air.

1.A.2. *Next, perform a large number of Brownian dynamics simulations of individual particles for the case D = 1, V_f = 1. Using the MATLAB histogram routines, generate approximate probability distributions p(t,x) at t = 0.5, 1, 2, 3 from the random trajectories x(t) that you generate in the simulations. Plot these vs. the analytical p(t,x) to demonstrate that indeed the Brownian dynamics simulations result in the correct p(t,x). Store your program as username_HW8_1A.m.*

OK, so here we're just doing a very straightforward problem. You have already been given the simulation equations and the values of all of the appropriate variables. You can either remove the (dU/dx)/ζ term from your equations in your code, or just set ζ to some arbitrary non-zero number since the zero value of dU/dx will then remove its effect.

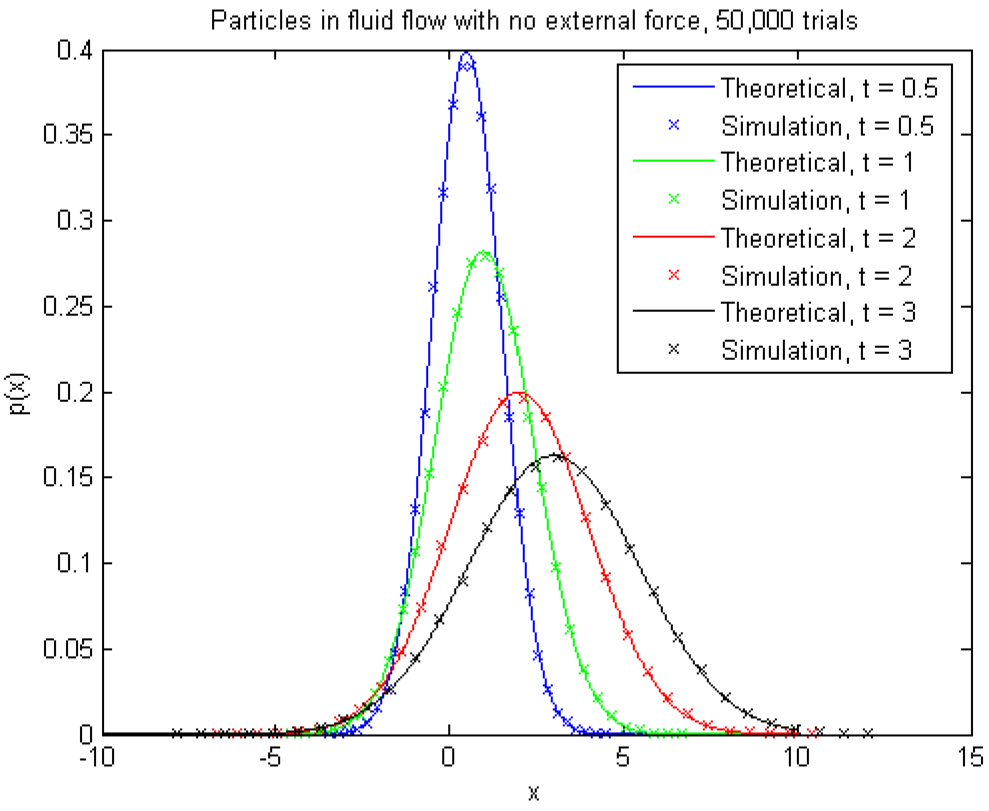
For Monte Carlo simulations, it is ideal to do a lot of trials. To get a fairly smooth curve, you need to damp out the random fluctuations caused by your sampling. To do that, you should run many trials. For instance, if you are using 50 bins for your histogram and you only run 1000 trials, that the average bin will have 20 occurrences. With such a low frequency, it is easy for random fluctuations to cause slight deviations from the expected distribution. At the same time, though, we would like our programs to run as quickly as possible, so we don't want to run too many trials.

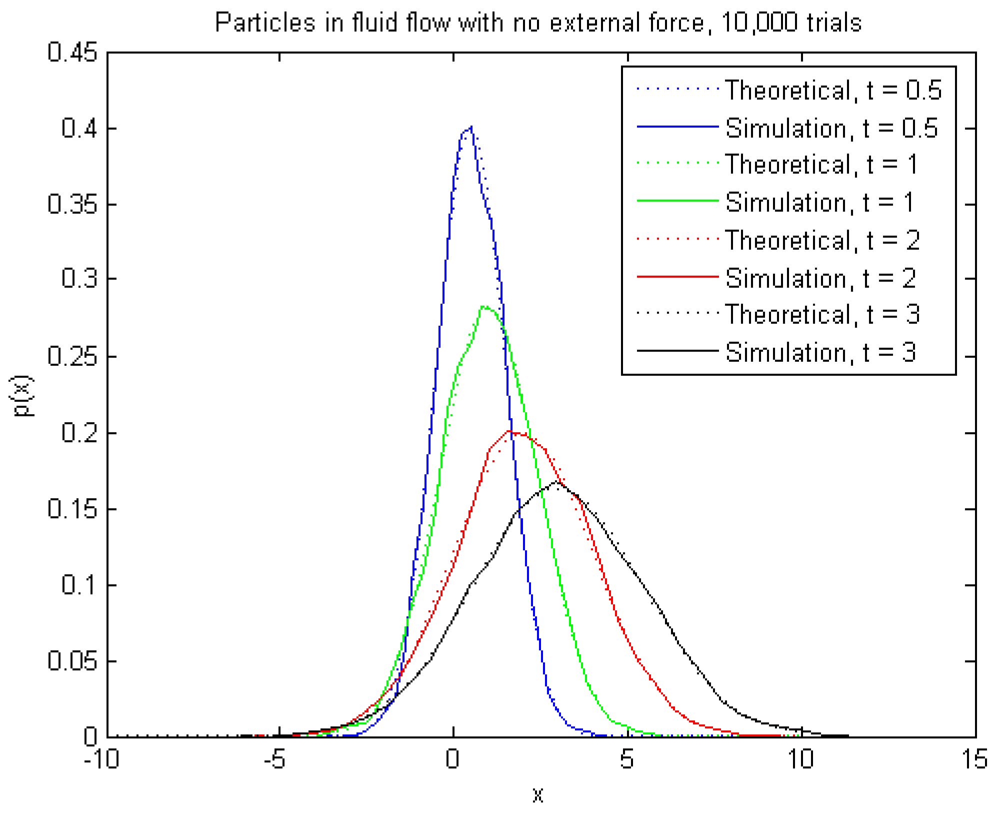
One thing to keep in mind for this task is that since we are running the commands in the same loop tens of thousands (or more) times, it behooves us to make only the most necessary commands be in the loop. For instance, if we needlessly re-assign a variable every time through the loop, that will take up extra time. If, on the other hand, we assign the variable outside of the loop, then we can cut down on the computational expense. Similarly, if you use a separate function to calculate your update, it is easiest if you don't bother to unpack your variables. What I mean by this is that if you pass in a bunch of parameters in a structure called Param, and each time you are in the function you say that x = Param.x, y = Param.y, theta = Param.theta, etc., then you are executing each of those thousands of times when they don't really need to be executed. Instead, you should just put Param.x, Param.y, Param.theta, etc., into your function. Though this makes your code harder to read, it makes the program run faster.

In fact, this is a consistent pattern throughout computational work: code that is easy to read is frequently suboptimal in execution time. Advanced computer languages and compilers attempt to circumvent these problems for you by intelligently optimizing your code when they assemble your program; in a case like this, though, it pays to note small ways to optimize your code and turn a 90-second execution time into a 10-second execution time. In my code, I have tried for a balance… by not outsourcing functions and including them "inline" in the loops, things may be a little faster than they otherwise might be, though perhaps with a little redundancy in my code. At the same time, though, I'm striving for legibility, even at the cost of storing extra variables.

Oh, one final thing to note: you may see that I start off by defining my "storage" array beforehand. This isn't strictly necessary, as Matlab can automatically extend the array for you. However, each time Matlab tries to extend an array, behind the scenes it is reallocating the array, a rather time-intensive process. This can produce a many-fold, even order-of-magnitude speedup for array that are particularly large.

That's it. The actual coding for this part isn't too tricky. Here's my sample code, and a graph that yours should look something like. The runtime for this program is about 30 seconds. The second figure shows how fewer trials will give the same trend, but with more noise.





```
% Mark Styczynski
% 10.34
% HW 8

% Problem 1.A.2.

clear all; close all;

% PDL> Set variables.
dUdx = 0;
D = 1;
Vf = 1;
deltaT = .001;
zeta = 1;  % Not needed for now, so set to arbitrary number
numTrials = 50000;

% PDL> Set initial state and allocate storage matrix.
x = 0;
storage = zeros(4,numTrials);

% PDL> Use tic and toc to keep track of my computation time
tic
% PDL> Perform a number of independent trials
for j = 1:numTrials,
    % PDL> Re-initialize each time.
    x = 0;
    % PDL> Run until desired times, then record sample.
    for i=1:.5/deltaT,
        deltaWt = randn()*sqrt(deltaT);
        x = x + (Vf-dUdx/zeta)*deltaT + (2*D)^.5*deltaWt;
    end

    storage(1,j) = x;

    for i=1:.5/deltaT,
        deltaWt = randn()*sqrt(deltaT);
        x = x + (Vf-dUdx/zeta)*deltaT + (2*D)^.5*deltaWt;
    end

    storage(2,j) = x;

    for i=1:1/deltaT,
        deltaWt = randn()*sqrt(deltaT);
        x = x + (Vf-dUdx/zeta)*deltaT + (2*D)^.5*deltaWt;
    end

    storage(3,j) = x;

    for i=1:1/deltaT,
        deltaWt = randn()*sqrt(deltaT);
        x = x + (Vf-dUdx/zeta)*deltaT + (2*D)^.5*deltaWt;
    end

    storage(4,j) = x;
end
toc

% PDL> Calculate theoretical curves.
tVals = [.5 1 2 3];
x = linspace(-10,10,10000);
for i=1:length(tVals),
    mu = Vf*tVals(i);
    sigma = sqrt(2*D*tVals(i));
    probPlot(i,:) = 1/sigma/sqrt(2*pi)*exp((-(x-mu).^2)./(2*sigma^2));
end

% PDL> Use histogram functions to produce probability distribution
%  function.
numBins = 30;
for i=1:4,
    [freq(i,:), bins(i,:)] = hist(storage(i,:),numBins);
    area(i) = trapz(bins(i,:),freq(i,:));
end

%PDL> Plot up results.
firstArray = ['b-';'g-';'r-';'k-'];
secondArray = ['bx';'gx';'rx';'kx'];
figure(1);
for i=1:4,
    plot(x,probPlot(i,:),firstArray(i,:));
    hold on;
    % Don't forget to normalize...
    plot(bins(i,:),freq(i,:)/area(i),secondArray(i,:));
end
xlabel('x')
ylabel('p(x)')
title('Particles in fluid flow with no external force, 50,000 trials')
legend('Theoretical, t = 0.5','Simulation, t = 0.5',...
    'Theoretical, t = 1','Simulation, t = 1',...
    'Theoretical, t = 2','Simulation, t = 2',...
    'Theoretical, t = 3','Simulation, t = 3');

% Alternate plot, if we had less runs.  Note it's noisier.
numBins = 30;
freq = [];
bins = [];
for i=1:4,
    [freq(i,:), bins(i,:)] = hist(storage(i,1:10000),numBins);
    area(i) = trapz(bins(i,:),freq(i,:));
end

firstArray = ['b:';'g:';'r:';'k:'];
secondArray = ['b-';'g-';'r-';'k-'];
figure(2);
for i=1:4,
    plot(x,probPlot(i,:),firstArray(i,:));
    hold on;
    % Don't forget to normalize...
    plot(bins(i,:),freq(i,:)/area(i),secondArray(i,:));
end
xlabel('x')
ylabel('p(x)')
title('Particles in fluid flow with no external force, 10,000 trials')
legend('Theoretical, t = 0.5','Simulation, t = 0.5',...
    'Theoretical, t = 1','Simulation, t = 1',...
    'Theoretical, t = 2','Simulation, t = 2',...
    'Theoretical, t = 3','Simulation, t = 3');
```

Part 1.B.

Now, set $v_f = 0$ and let us introduce a spatially-periodic potential

$$U(x) = E_a\left[\sin(x\pi)\right]^2$$

Of periodicity $U(x \pm m) = U(x)$, $m = 0, 1, 2, 3, \ldots$. This potential consists of a sequence of energy barriers of height E_a each separated by a distance of one.

Perform a Brownian dynamics simulation using periodic boundary conditions on the domain $0 \le x \le 1$ in which at any time the particle exits the domain, it is shifted by ±1 to bring it back inside the domain. Since the potential energy is also periodic, this shifting has no deleterious effect on the simulation, and it enables us to compare the measured probability distribution P(x) over the course of the simulation with the Boltzmann distribution

$$P(x) = \frac{e^{\frac{-U(x)}{k_b T}}}{\int_0^1 e^{\frac{-U(x)}{k_b T}} dx}$$

In your Brownian dynamics simulation, you should not start sampling the distribution of x until you have run the simulation for a while to "equilibrate" the system. Simulate the motion of a particle at $k_bT = 1$, $E_a = 1$, $D = 1$ and demonstrate that the Brownian dynamics simulation samples properly from the equilibrium distribution. That is, the probability distribution measured from the trajectory x(t) agrees with the Boltzmann distribution. Store your program as username_HW8_1B.m.

Well, the first thing to note is that using equation 9, $D = \frac{k_b T}{\zeta}$, we know that $\zeta = 1$.

Everything else is fairly straightforward; the parameters have been given to you. You do need to realize that dU/dx must be recalculated at each step because it varies with x. You also need to play with your equilibration time a little bit, as well as your time steps. For each of these, you should decide on their values similarly to how you decided on the number of grid points in a finite difference approximation: gradually change them until you see no difference in your solution. For instance, if your time step is too big (say even 0.01), then the same number of equilibration steps as a smaller time step (say 0.001) will not give you the correct result… instead, the curve will be not quite equilibrated. So, these must be adjusted appropriately until the expected behavior is observed. There are many correct values for these parameters that will achieve this, you just needed to find one set. Finding these appropriate values is the key part of this problem.

It may also be worth noting that the correct derivative dU/dx is

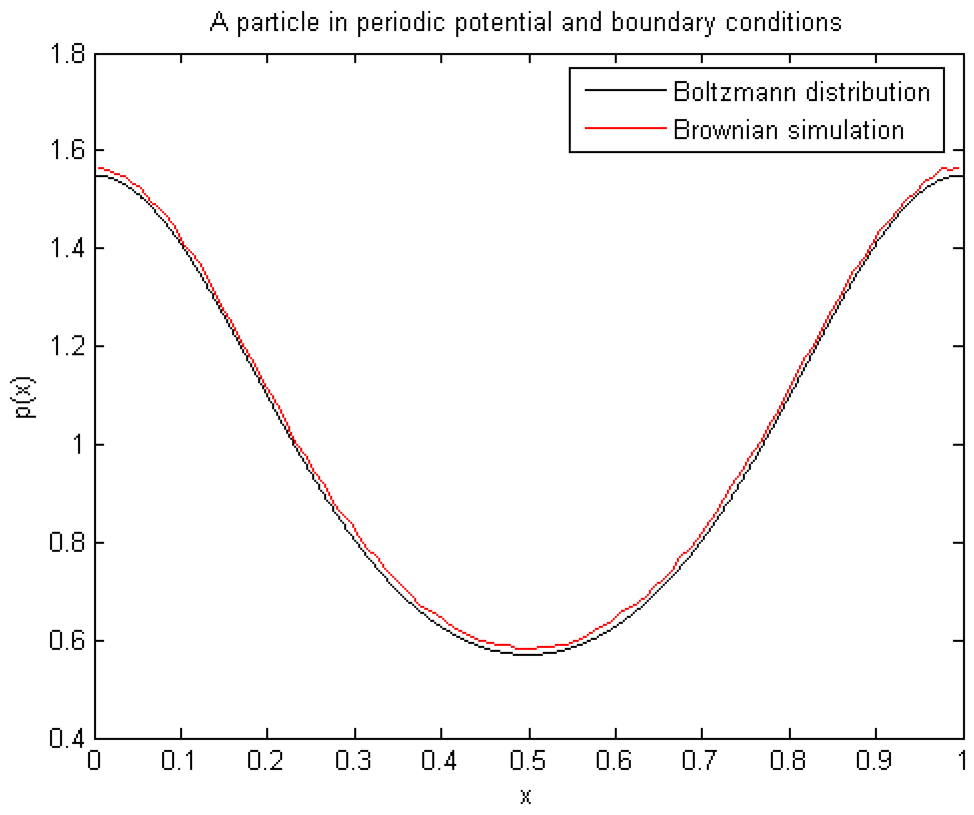
$$\frac{dU}{dx} = \frac{d}{dx}\left\{E_a\left[\sin(x\pi)\right]^2\right\} = 2E_a \sin(x\pi)\frac{d}{dx}\sin(x\pi)$$
$$= 2E_a\pi \sin(x\pi)\cos(x\pi)$$

When you integrate your curves, note that you must supply the x vectors to trapz; if you only supply the frequency or probability vector, you'll be some (significant) constant factor off in the scale of your probabilities, and the area would not integrate to 1.

I did multiple trials just to get a smoother graph; one trial would have given a little more jagged of a graph. Another alternative to smooth out would be to just sample for a lot longer; this would produce the same result.

Finally, for your graph… if your graph consistently follows the trend of the theoretical graph but is offset slightly above, this is acceptable (if not expected). This is because we are not able to properly normalize because there are no bins centered at 0 or 1; this means that when using trapz, we are essentially integrating from "almost 0" to "almost 1", and so we are missing some area. The area calculated by trapz is then smaller than the real area, and our curve will be shifted slightly up. This effect can be decreased by using more bins (so that we integrate closer to 0 and 1, capturing more of the area) or by extrapolating a point at 0 and 1 from the curve assuming a derivative equal to zero. But the bins are likely to be unevenly spaced by default, making that extrapolation a little more difficult and thus entirely optional. Brownie points for anyone who did that, though.

That's about it. Here's a figure and the code that generated it.



```
% Mark Styczynski
% 10.34
% HW 8

% Problem 1.B.

clear all; close all;

% PDL> Set variables.
D = 1;
Vf = 0;
deltaT = .001;
kbT = 1;
zeta = kbT/D;
Ea = 1;

% Base our number of points on the simualation time and the
%   deltaT.
```

```
numEquil = round(150/deltaT);
numPoints = round(150/deltaT);
numTrials=100;

x = 0;

% PDL> Define array here instead of constantly redefining and
%  making MATLAB reallocate it.
storage = zeros(numTrials,numPoints);

tic

% PDL> Perform a number of trials.
for j = 1:numTrials,
    x = 0;
    % PDL> Wait a while for the system to equilibrate.
    for i=1:numEquil,
        dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
        deltaWt = randn()*sqrt(deltaT);
        x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
        if (x ~= 1),
            x = mod(x,1);
        end
    end
    % PDL> Then sample from the system.
    for i=1:numPoints,
        dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
        deltaWt = randn()*sqrt(deltaT);
        x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
        if (x ~= 1),
            x = mod(x,1);
        end
        storage(j,i) = x;
    end
end
toc

% Rearrange the storage matrix so that it's just an array.
storage = reshape(storage,1,numTrials*numPoints);

% PDL> Calculate the theoretical curve.
x = linspace(0,1,1000);
probInitial = exp(-Ea*(sin(pi*x)).^2./kbT);
norm = trapz(x,probInitial);
probPlot = probInitial/norm;

% PDL> Use hist functions to create a probability density function.
numBins = 100;
[freq, bins] = hist(storage,numBins);
area = trapz(bins,freq);

figure(1);
plot(x,probPlot,'k-');
hold on;
plot(bins,freq/area,'r-');
```

```
xlabel('x')
ylabel('p(x)')
title('A particle in periodic potential and boundary conditions')
legend('Boltzmann distribution','Brownian simulation')
```

Part 1.C.

Consider again the same external potential, but now do not use the periodic boundary conditions. Instead, generate trajectories x(t) that are not shifted in space to remain in [0,1]. In the limit $E_a << k_bT$, the energy barriers are negligibly small and the particles essentially undergo "regular" diffusion. But, when E_a becomes comparatively large relative to k_bT, we expect the barriers to be difficult to overcome such that the particle trajectories become "trapped" between barriers for a long time until they are finally able to "jump" to the next energy well. If we then continue the simulation over very long periods of time such that each trajectory has experienced many jumps, and we measure at various times the mean squared displacement, $<x^2(t)>$, we can estimate the effective diffusivity, D_{eff}, in the presence of the barriers from the relation $<x^2(t)> = 2 D_{eff}t$ as $t \rightarrow \infty$.

Perform this calculation to measure D_{eff} when D = 1 for E_a = 0.1, 0.5, 1, 2, 3, 4, 5 when k_bT = 1 and plot ln D_{eff} vs. E_a. A reasonable prediction of how the effective diffusivity should be affected by the barrier height is

$$D_{eff} = De^{\frac{-E_a}{k_bT}}$$

Compare the results of your calculation to this functional form to see if it is an accurate description of the effect of energy barriers on long-time diffusive motion. Store your program as username_HW8_1C.m.

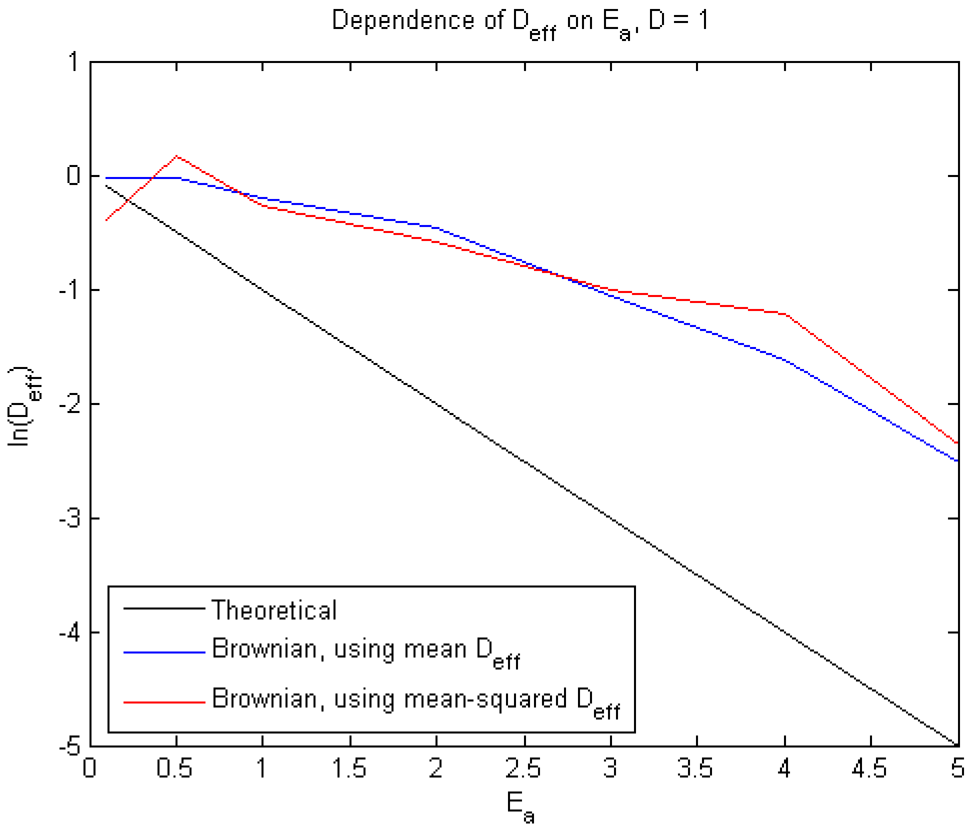
OK, so first we need to remove the periodic boundary condition from our code. We also need to know how to calculate $<x^2(t)>$; this is rather simple, actually. Though you could have created a probability distribution function using histogram routines, this was unnecessary. All you need to do is calculate the average of x^2, which in this case is the sum of all x^2 values divided by the number of samples taken. This is equivalent to taking $\sum_i x_i^2 P(x_i)$, because in this case the probability of each x_i is just one divided by the number of samples. This is actually more accurate than the histogram method since it does not put items into bins before calculating an average.

The equation that Dr. Beers gives, $D_{eff} = De^{\frac{-E_a}{k_bT}}$, can be reduced to

$\ln D_{eff} = \ln D + \ln e^{\frac{-E_a}{k_bT}} = -E_a$ since D = 1 and k_bT = 1.

We see that the plots do not agree; that is, the predicted effect of barrier height on effective diffusivity is not exactly the equation that Dr. Beers gave you. There is a different slope at the very least, and probably even a different functionality. As was mentioned in class, this may have to do with differences between activation energy/transition state theory and our methods of simulation.

Note that there are different methods to calculate the effective diffusivity given a set of $<x^2(t)>$ values; Dr. Beers mentioned in class that you can do a simple polynomial or least squares fit; I found it easier to divide the $<x^2(t)>$ values by the time at which they were taken and to then take the simple arithmetic mean of the resulting D_{eff} values. Either way is valid, and they should yield approximately equal results given equal sampling. However, it is a little bit harder to get good sampling for the line-fitting method, as the best fits will occur over a wider range of time. I ran the code for the same number of iterations for both methods, but that gives significantly fewer samples for the line-fitting method, so the curve looks more jagged. The main point is there, though: either method will produce approximately the same curve, which does not agree with the equation that Dr. Beers proposed.



```
% Mark Styczynski
% 10.34
% HW 8

% Problem 1.C.

clear all; close all;

% PDL> Set variables
D = 1;
```

```
Vf = 0;
deltaT = .001;
kbT = 1;
zeta = kbT/D;
numTrials = 100;
equilTime = 100;
EaList = [.1 .5 1 2 3 4 5];

% PDL> Set up our matrix for storage
fitStorage = zeros(length(EaList),6,numTrials);
samplerStorage = zeros(length(EaList),numTrials);

% PDL> Perform calculations for each activation energy
for k=1:length(EaList),
    tic
    Ea = EaList(k)
    x = 0;
    % PDL> Perform a number of different trials to get a
    %  representative value of <x^2>
    for j = 1:numTrials,
        x = 0;
        % PDL> Let the system equilibrate so t --> infinity
        for i=1:equilTime/deltaT,
            dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
            deltaWt = randn()*sqrt(deltaT);
            x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
        end
        % PDL> Now start recording specific time points
        storage(k,1,j) = x;
        % Let the system go for a little while before next time point.
        for i=1:10/deltaT,
            dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
            deltaWt = randn()*sqrt(deltaT);
            x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
            % PDL> Also record many consecutive time points, using
            %  the equation <x^2>/(2*t) = Deff.
            % So we are accumulating a bunch of x^2 values here...
            samplerStorage(k,j) = samplerStorage(k,j) +
x^2/2/(i*deltaT+equilTime);
        end
        % And then we divide by the number of values to get the
average.
        samplerStorage(k,j) = samplerStorage(k,j)/(10/deltaT);
        storage(k,2,j) = x;
        % Perform a few more iterations to find more time points.
        for i=1:10/deltaT,
            dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
            deltaWt = randn()*sqrt(deltaT);
            x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
        end
        storage(k,3,j) = x;
        for i=1:10/deltaT,
            dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
            deltaWt = randn()*sqrt(deltaT);
            x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
        end
        storage(k,4,j) = x;
        for i=1:10/deltaT,
            dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
            deltaWt = randn()*sqrt(deltaT);
            x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
        end
        storage(k,5,j) = x;
        for i=1:10/deltaT,
            dUdx = Ea*2*sin(pi*x)*cos(pi*x)*pi;
            deltaWt = randn()*sqrt(deltaT);
            x = x + (Vf-dUdx/zeta)*deltaT + sqrt(2*D)*deltaWt;
        end
        storage(k,6,j) = x;
    end
    toc
end


for k=1:length(EaList),
    for i=1:6,
        % PDL> Calculate <x^2> for the slope-fitting method.
        DeffT(k,i) = mean(storage(k,i,:).^2)/2;
        % Alternative way of calculating average x^2 using
histograms...
        % numBins = 30;
        % [freq(k,i,:), bins(k,i,:)] = hist(storage(k,i,:),numBins);
        % area(k,i) = trapz(bins(k,i,:),freq(k,i,:));
        % Deff2(i) =
trapz(bins(i,:),freq(i,:)/area(i).*(bins(i,:).^2))/2/(i+equilTime-1);
    end
    % PDL> Find Deff as slope of Deff*t vs. t
    timeArray = linspace(equilTime,equilTime+50,6);
    fit = polyfit(timeArray,DeffT(k,:),1);
    DeffFit(k) = fit(1);
end


% PDL> Plot up results.
figure(1)
plot(EaList,-EaList,'k-')
hold on
plot(EaList,log(mean(samplerStorage')),'b-')
plot(EaList,log(DeffFit),'r-')
xlabel('E_a')
ylabel('ln(D_e_f_f)')
title('Dependence of D_e_f_f on E_a, D = 1')
legend('Theoretical','Brownian, using mean D_e_f_f', ...
    'Brownian, using mean-squared D_e_f_f','Location','SouthWest')
```

Problem 2. Metropolis Monte Carlo Simulation

Consider again the 1-D system with the periodic potential energy

$U(x) = E_a\left[\sin(x\pi)\right]^2$

Write a program to sample the NVT equilibrium distribution of x using Metropolis Monte carlo for the case $E_a = 1$, $k_bT = 1$, and show that the results agree with the Boltzmann distribution

$$P(x) = \frac{e^{\frac{-U(x)}{k_bT}}}{\int_0^1 e^{\frac{-U(x)}{k_bT}} dx}$$

Where again you use periodic BC to maintain the particle within $0 \le x \le 1$.

Here we implement a Metropolis Monte Carlo method to sample from the NVT distribution. To make an iterative move from our current point, we need to add some random number to our value of x. In class Dr. Beers suggested that we sample from a normal distribution; in the book and in some sample code he used a uniform distribution. Either of those should suffice for this case. Of course, be sure that if you use the uniform distribution with rand() that you recenter it around zero by subtracting 0.5 from whatever value it gives you.

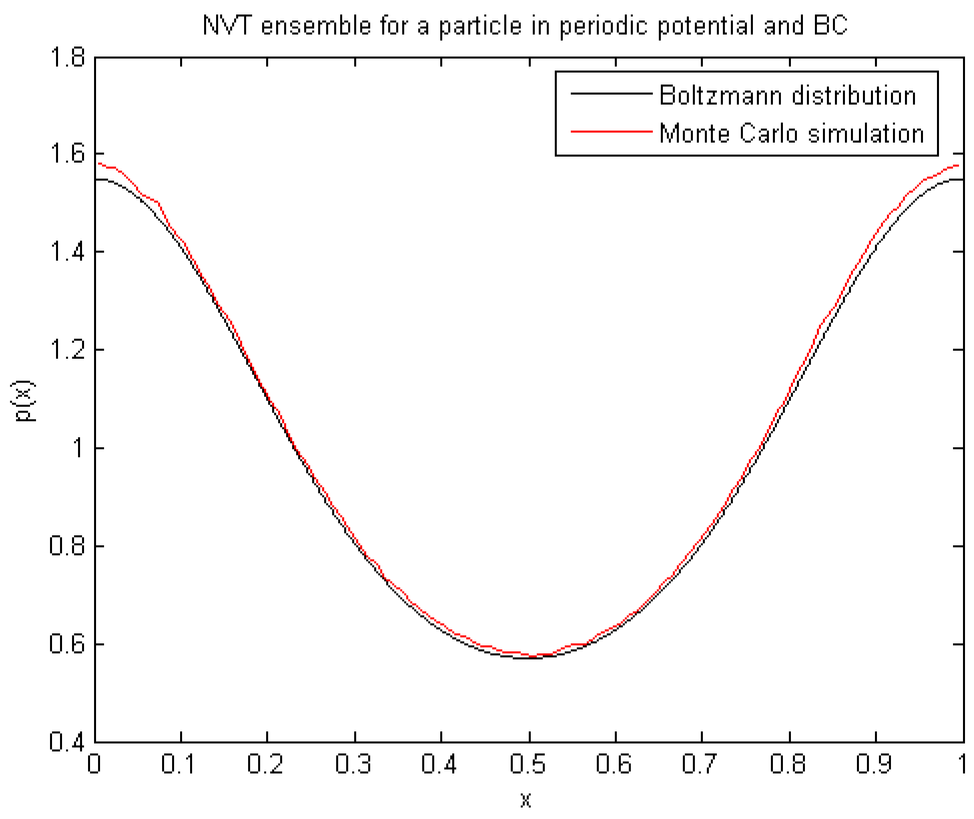
As we know from the book, the update steps for Metropolis Monte Carlo are accepted dependent upon the ratio of the probability of the proposed state to the probability of the current state. Since these two probabilities have the same normalization factor, this can be directly calculated as

$$\alpha = \frac{e^{\frac{-U(x_{new})}{k_bT}}}{e^{\frac{-U(x_{curr})}{k_bT}}} = e^{\frac{-[U(x_{new})-U(x_{curr})]}{k_bT}}$$

We then select a number, u, from a uniform distribution over [0,1]. If $u \le \min\{1, \alpha\}$, then we accept the proposed step and sample from it. Otherwise, we stay at the current step and sample from it.

Note that the same issues with integration for the purposes of normalizing the probability distribution function still apply, so it's OK if our graph is offset from the Boltzmann distribution slightly upward.

Here is the resulting figure and the code to produce it. The program takes about 20 seconds to run.



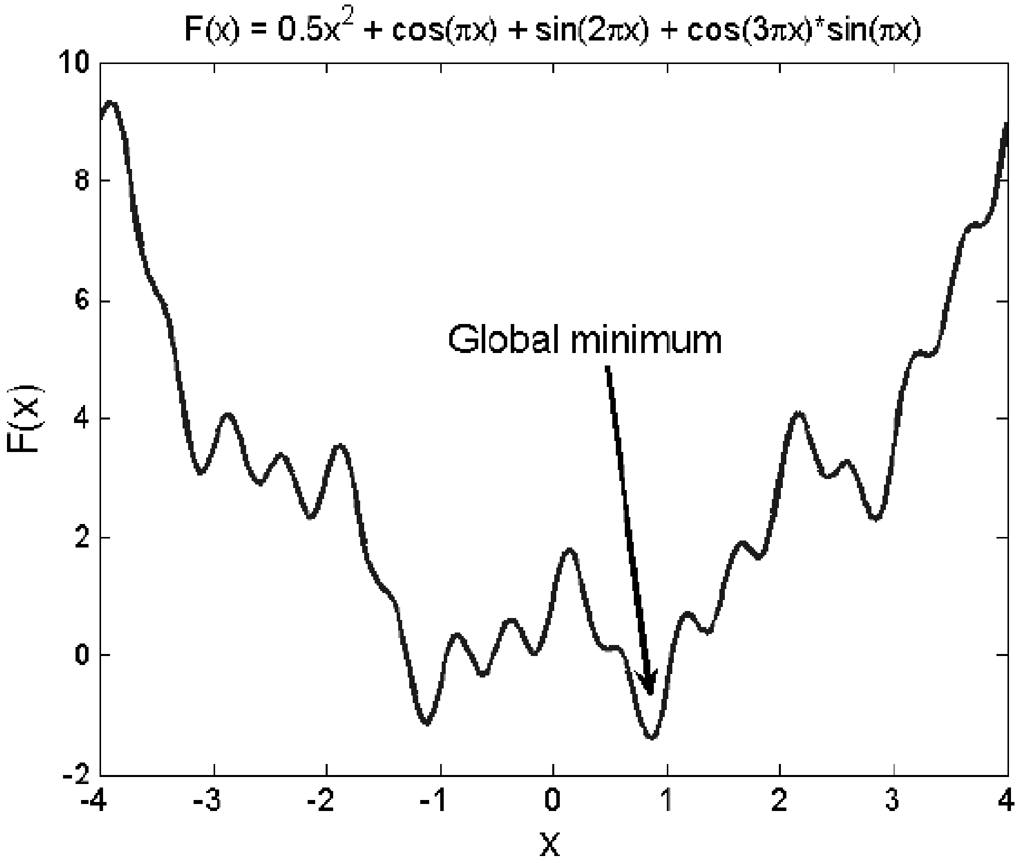
```
% Mark Styczynski
% 10.34
% HW8
% Problem 2. Metropolis Monte Carlo simulation

clear all; close all;

% PDL> Set variables.
Ea = 1;
kbT = 1;
stdDevFactor = .2;
numEquil = 1e7;
numSteps = 1e7;
x = 0;
freq = 1;

% Create an array to store the x values, to speed things up.
storage = zeros(1,floor(numSteps/freq));
expFactor = Ea/kbT;
j=0;
tic

% PDL> First let the system equilibrate to make sure we're at a
%  state that is relatively likely.
for i=1:numEquil,
    % Get a new point.
    new_x = x+(rand()-0.5)*stdDevFactor;
    % Apply the BCs.
    if (new_x ~= 1),
        new_x = mod(new_x,1);
    end
    % Calculate the acceptance probability.
    probRatio = exp(-expFactor*((sin(new_x*pi))^2-(sin(x*pi))^2));
    acceptProb = min(1,probRatio);
    acceptRand = rand();
    if acceptProb >= acceptRand
        x = new_x;
    end
end
toc
tic
% Now we start sampling.
for i=1:numSteps,
    new_x = x+(rand()-0.5)*stdDevFactor;
    % Apply BCs
    if (new_x ~= 1),
        new_x = mod(new_x,1);
    end
    % Calculate probabilitiy.
    probRatio = exp(-expFactor*((sin(new_x*pi))^2-(sin(x*pi))^2));
    acceptProb = min(1,probRatio);
    acceptRand = rand();
    % Decide if we move or not.
    if acceptProb >= acceptRand
        x = new_x;
    end
    % Sample either way.
    storage(floor(i/freq)) = x;
end
toc

% PDL> Calculate theoretical distribution.
x = linspace(0,1,1000);
probInitial = exp(-Ea*(sin(pi*x)).^2./kbT);
norm = trapz(x,probInitial);
probPlot = probInitial/norm;

% PDL> Calculate probability distribution function using
%  histogram routines.
numBins = 100;
[freq, bins] = hist(storage,numBins);
area = trapz(bins,freq);
figure(1);
plot(x,probPlot,'k-');
hold on;
plot(bins,freq/area,'r-');
xlabel('x')
ylabel('p(x)')
title('NVT ensemble for a particle in periodic potential and BC')
legend('Boltzmann distribution','Monte Carlo simulation',...
    'Location','Best')
```

Problem 3. Simulated annealing

Consider the cost function

$F(x) = 0.5x^2 + \cos(\pi x) + \sin(2\pi x) + \cos(3\pi x)\sin(\pi x)$

Plotted in the figure below.



Clearly, the cost function has many local minima, and it would be very difficult to find the global minimum using the deterministic techniques that we developed in chapter 5. Unfortunately, such irregular cost functions are not uncommon, especially when attempting to compute the minimum energy geometry of a molecule or a crystal.

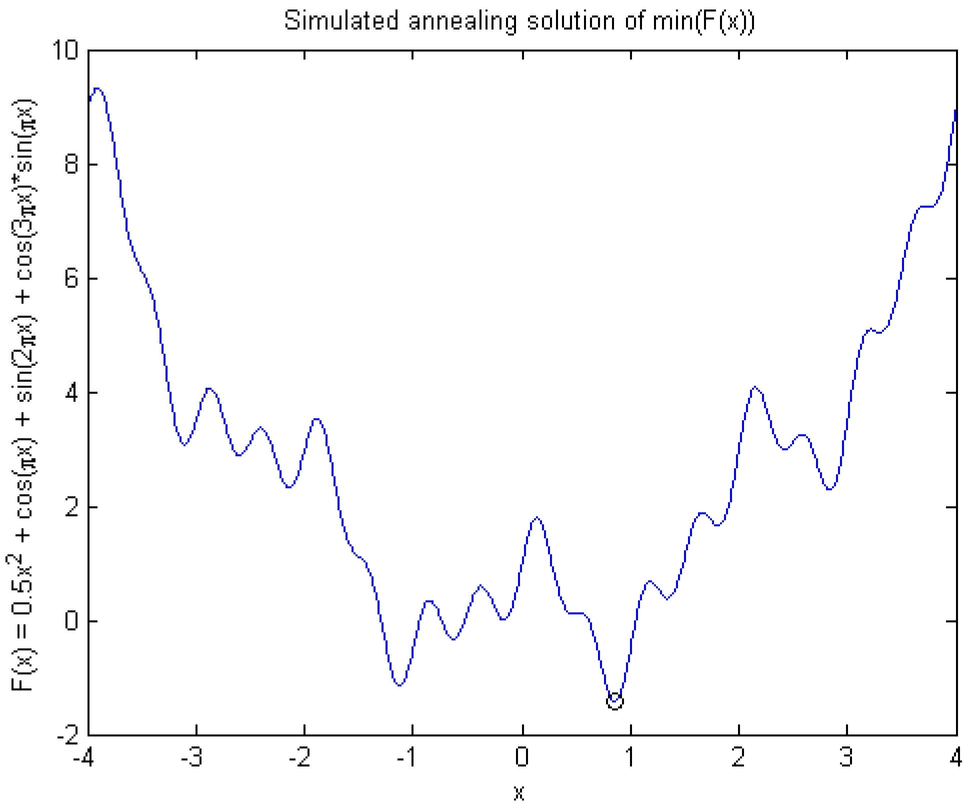
Write a program that uses simulated annealing to identify the global minimum from a random initial guess. Store your program as username_HW8_P3.m and provide directions for its use.

Yes, if you used Dr. Beers's sample code from the website, you will get credit for this part. However, if something went wrong somewhere in your program, you are much more likely to be given partial credit for work that you did versus work that you borrowed (with appropriate citations) from the website.

As for the specific problem… as detailed in the book, simulated annealing is essentially a sequential set of MCMC problems designed to find the minimum of your function. (See above for a brief explanation of the MCMC method.) Slowly, the "temperature" parameter is lowered; as this happens, the routine is less likely to accept moves that result in higher function values, but still leaves some non-zero chance. This method helps to avoid (though does not eliminate) getting stuck in local minima.

It is important to note that simulated annealing still can get stuck in local minima. In addition, it may be somewhat dependent on your initial guess (particularly if your maximum displacement/move/jump size is relatively small). As such, it is best to run it multiple times and see what the result is. Whatever single lowest function value you find in any run is then the answer that you should return.

And as you can mostly see from the graph you were initially given, the correct minimum for this system is x = 0.8566, F(x) = -1.4120. Here's that figure again, and some fresh code to find the solution.



```
% Mark Styczynski
% 10.34
% HW8
% Problem 3. Simulated annealing

% PDL> Set variables.
numRuns = 10;
numSteps = 1e6;
initKbT = 100;
maxDev = .5;
```

```
% PDL> Set an initial guess; this is just a placeholder to
%  make sure that the references in the loop are OK.
bestX = 0;
bestF = .5*bestX^2+cos(pi*bestX)+sin(2*pi*bestX)+ ...
    cos(3*pi*bestX)*sin(pi*bestX);

tic

% PDL> Run a number of times to be more sure we get the global
%  minimum.
for k=1:numRuns,
    % I decided to use random initialization points in a known range.
    x = (rand()-.5)*8;
    f = .5*x^2+cos(pi*x)+sin(2*pi*x)+cos(3*pi*x)*sin(pi*x);
    % PDL> Perform a number of MCMC runs with different temperatures.
    for j=1:numSteps,
        % PDL> Degrade to kbT (almost) 0 linearly... note that we
        %  do not actually want to get to 0.
        kbT = initKbT*(numSteps+1-j)/numSteps;
        % PDL> Propose a new x
        newX = x + (rand() - 0.5)*maxDev;
        newF = .5*newX^2+cos(pi*newX)+sin(2*pi*newX)+ ...
            cos(3*pi*newX)*sin(pi*newX);
        % PDL> Find the probability of accepting it
        probRatio = exp(-(newF - f)/kbT);
        acceptProb = min(1,probRatio);
        acceptRand = rand();
        % PDL> Accept or reject
        if (acceptProb >= acceptRand)
            x = newX;
            f = newF;
        end
    end
    % PDL> See if we've found a new minimum.
    opts = optimset('Display','Off');
    [x, f] = fminunc(@(x) ...
    .5*x^2+cos(pi*x)+sin(2*pi*x)+cos(3*pi*x)*sin(pi*x), x, opts);
    if (f < bestF),
        bestF = f;
        bestX = x;
    end
end
toc

bestX
bestF


% PDL> Plot up the function and our result.
plotX = linspace(-4,4,1000);
plotF = .5*plotX.^2 + cos(pi*plotX) + sin(2*pi*plotX) + ...
    cos(3*pi*plotX).*sin(pi*plotX);
figure(1)
plot(plotX,plotF,'b-')
hold on
plot(bestX,bestF,'ko')
```

```
xlabel('x')
ylabel('F(x) = 0.5x^2 + cos(\pix) + sin(2\pix) + cos(3\pix)*sin(\pix)')
title('Simulated annealing solution of min(F(x))')
```

<u>Problem 4. 4.A.2</u>

Compute the value of the following definite integral using both **dblquad** *and Monte Carlo integration.*

$$I_D = \int_1^2 \int_0^{\sqrt{x}} \left[(x-1)^2 + y^2 \right] dy dx$$

Monte Carlo integration is explained in your book, particularly on pages 241-243 and 504. There are two different methods by which we can do this: one using an iterative, Markov chain approach, and one using a simple uniform sampling approach. Most of you likely used the latter, but I'll present both here for completeness.

The uniform sampling approach is explained in chapter 4. Essentially, we just define a hypercube that completely contains the space to be integrated and multiply the function to be integrated, f, by an indicator function, Ω. The indicator function takes a value of one if a sample point is in the space to be integrated and zero otherwise. We then draw values at random from a uniform distribution across the entire hypercube and take the sum of all samples' function values times their indicator function. That is, we take f*Ω for all samples and accumulate a sum. This sum is then divided by the number of samples, yielding an "average" value for the function f over the hypercube. In the limit that we have an infinite number of samples, we will have a sample from every point and our average will be correct. We then need only multiply by the space of the hypercube to get the appropriate integral.

The alternative approach, briefly explained on page 504 (beware typos, though), is to use an iterative method. Here, as before in MCMC methods, we propose a new step each time. Now, our probability function is defined as

$$P(\underline{x}) = \frac{H_\Omega\left(\underline{x}^{[j]}\right)}{V_S}$$

Where V_S is the space being integrated (not necessary right now, as it will cancel in our calculations), and H_Ω is an indicator function with a value of 1 if we are in the space to be integrated and 0 otherwise. Thus, assuming we start in the space to be integrated, then the value of the ratio of $P(x_{\underline{proposed}})/P(\underline{x}_{current})$ is always either 1 or 0… that is, we either definitely accept the sample if it is in the space, or definitely reject it if it is not. If we reject it, we just sample the current point again. (See problem 2 for a quick run-through of MCMC methods.) This time, the average function value that we get will only be for the space to be integrated; in this case, we multiply this average by the volume of the space to be integrated, not the volume of some hypercube. (Of course, this does mean that we have to know that volume, which may in some cases be non-trivial to calculate.)

And of course, the answer hasn't changed from the previous assignment… you should be getting 1.0612 for the area (or at least approximately that, depending on your sample size). Here's the code to do all of that… and now there's only one more assignment left.

```
function marksty_HW8_P4()

clear all; close all;

% PDL> Use code from the previous homework assignment.
dblquad_area = dblquad(@p4a2fun,1,2,0,sqrt(2))
manually_caclulated = 2/7*2^(7/2) - 2/3*2^(5/2) + 2/3*2^(3/2) - 2/7

% PDL> Set variables.
stdDevFactor = .1;
numEquil = 1e6;
x = [1.5 1];
hFunVal = hSigmaFun(x);
fTotal = 0;
sCounter = 0;
tic

% PDL> Set storage vector.
storage=zeros(numEquil,2);

% PDL> First do MCMC method.
% PDL> First let the system equilibrate.
for i=1:numEquil,
    new_x = x+(rand(1,2)-[0.5 0.5]);
    newHFun = hSigmaFun(new_x);
    % PDL> Calculate probabilities.
    % Don't need to worry about the denominator here.
    probRatio = newHFun/hFunVal;
    acceptProb = min(1,probRatio);
    acceptRand = rand();
    if acceptProb >= acceptRand,
        x = new_x;
        hFunVal = newHFun;
    end
end
% PDL> Now we sample from it.
for i=1:numEquil,
    new_x = x+(rand(1,2)-[0.5 0.5]);
    newHFun = hSigmaFun(new_x);
    % PDL> Calculate probabilities.
    % Don't need to worry about the denominator here.
    probRatio = newHFun/hFunVal;
    acceptProb = min(1,probRatio);
    acceptRand = rand();
    if acceptProb >= acceptRand,
        x = new_x;
        hFunVal = newHFun;
    end
    fTotal = fTotal + (x(1)-1)^2 + x(2)^2;
```

```
end
toc

% PDL> Calculate the average and the space being sampled
averageF = fTotal/numEquil;
volS = quad(@(x) sqrt(x),1,2);

% PDL> Multiply to get the integral
MCintegral = averageF*volS

% PDL> The slightly easier Monte Carlo integration...
% S is a rectangle from 1 to 2 in x and 0 to sqrt(2) in y
newFTotal = 0;
for i=1:numEquil,
    % PDL> Draw numbers from a uniform distribution
    x(1) = 1 + rand();
    x(2) = sqrt(2)*rand();
    newFTotal = newFTotal + hSigmaFun(x)*((x(1)-1)^2 + x(2)^2);
end

% PDL> Calculate the average and the space being sampled
aveF = newFTotal/numEquil;
newVolS = sqrt(2);

% PDL> Multiply to get the integral
newMCintegral = aveF * newVolS

end

% PDL> Function to integrate, from previous homework assignment.
function integrand = p4a2fun(x,y)

integrand = (x - 1).^2 + y^2;
multiplier = (y <= sqrt(x));
integrand = integrand.*multiplier;

end

% PDL> H_sigma function for doing both Monte Carlo methods.
function answer = hSigmaFun(x)
% PDL> Require sample to be in integrated space.
  if ((x(1) <= 2) && (x(1) >= 1) && (x(2) >= 0) && (x(2) <=
sqrt(x(1)))),
      answer = 1;
  else
      answer = 0;
  end
end
```