Statistical Mechanics: HW 5

Truman Ellis

April 19, 2011

We use the Verlet algorithm to integrate the equations of motion of a single particle of mass $m = 1$ moving on the Muller potential. I start our simulations at the lowest energy potential, which I found at $(x, y) = (-0.5582, 1.4417)$ which had a Muller potential of -146.7. I randomly set the initial velocity direction, but fix the magnitude to give a total energy of either -39 or -45. I found that a timestep of $\Delta t = 4 \times 10^{-3}$ consistently gave an energy error of less than one percent and that 200,000 timesteps allowed the simulation to reach an equilibrium average for x, y, and xy.

The particle with a total energy of $E = -39$ had enough kinetic energy to occasionally break out of the deepest energy well and explore some other part of the domain. When it did, it always got caught in a neighboring well for some time before breaking out of that one and reentering the original well. I would let this process repeat a sufficient number of times until a reliable average could be calculated. In most cases, however, there was sufficient symmetry in the problem for the particle to follow a repeating path in which it never broke out of the energy well. Several sample trajectories are illustrated in Figures (1) - (7). The particle with total energy $E = -45$ never had enough kinetic energy to break out of the energy potential barrier, and thus never got to explore other parts of the problem domain. In each run, it would just achieve some path symmetry and repeat this pattern until the end of the simulation. There appeared to be fewer trajectory patterns for this lower energy particle. Figures (8) - (11) illustrate several sample patterns.

We would expect that for a given energy, the expected x, y, and xy values would be the same no matter which starting point and which direction the initial velocity was directed towards, but as the plots demonstrate, this is not true. Even though each simulation originates at the same point, a simple matter of reorienting the initial velocity vector changes the calculated averages, sometimes dramatically. I believe this is primarily because the particle gets stuck in a pattern and never fully explores the approximation space as it should. Thus, molecular dynamics does not appear to be an adequate solution for calculating micro-canonical averages. Perhaps if many particles were initialized at random points in the domain with random velocity orientations, we would do better. I think a much more efficient approach would be to use some sort of Monte Carlo technique. This would break the symmetry in the particle paths and allow a particle to more fully explore the problem space.

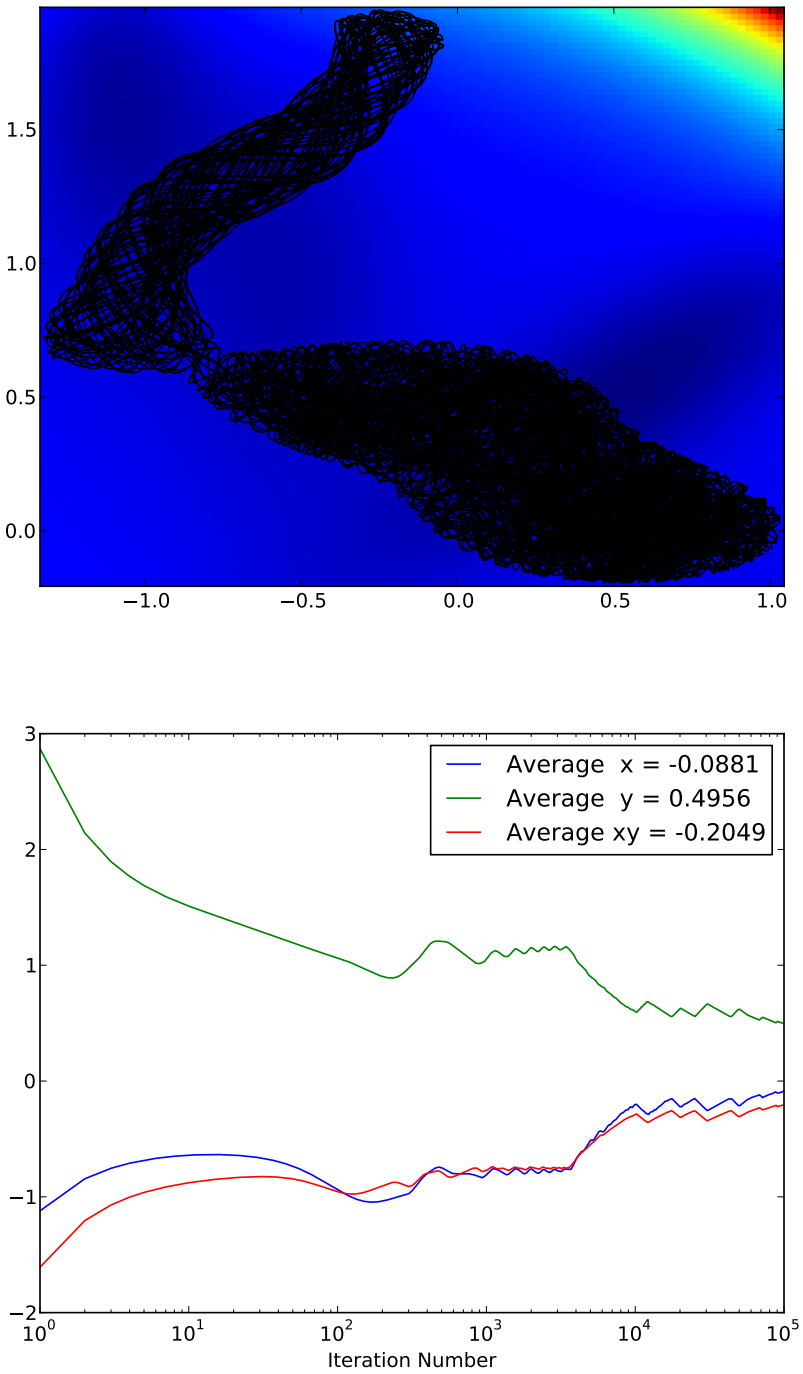

Figure 1: $E = -39$: Simulation with 100,000 smaller timesteps $\Delta t = 1 \times 10^{-3}$ to illustrate particle path. The plot of the averages does not quite look sufficiently stabilized in the simulation.

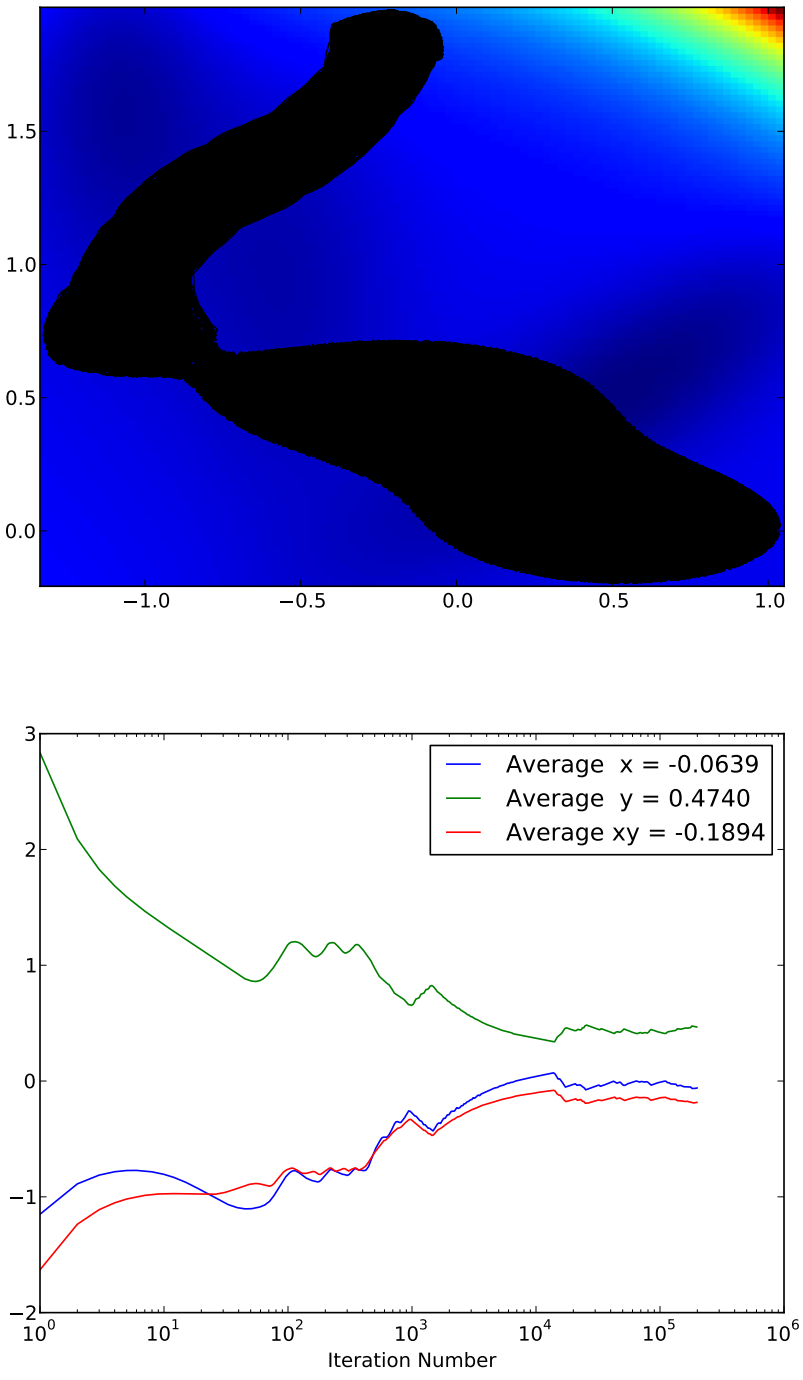

Figure 2: $E = -39$: Similar simulation with 200,000 simulations and a larger timestep. The plot of averages appears to have come to equilibrium. The particle has covered the space much more efficiently than the previous simulation.

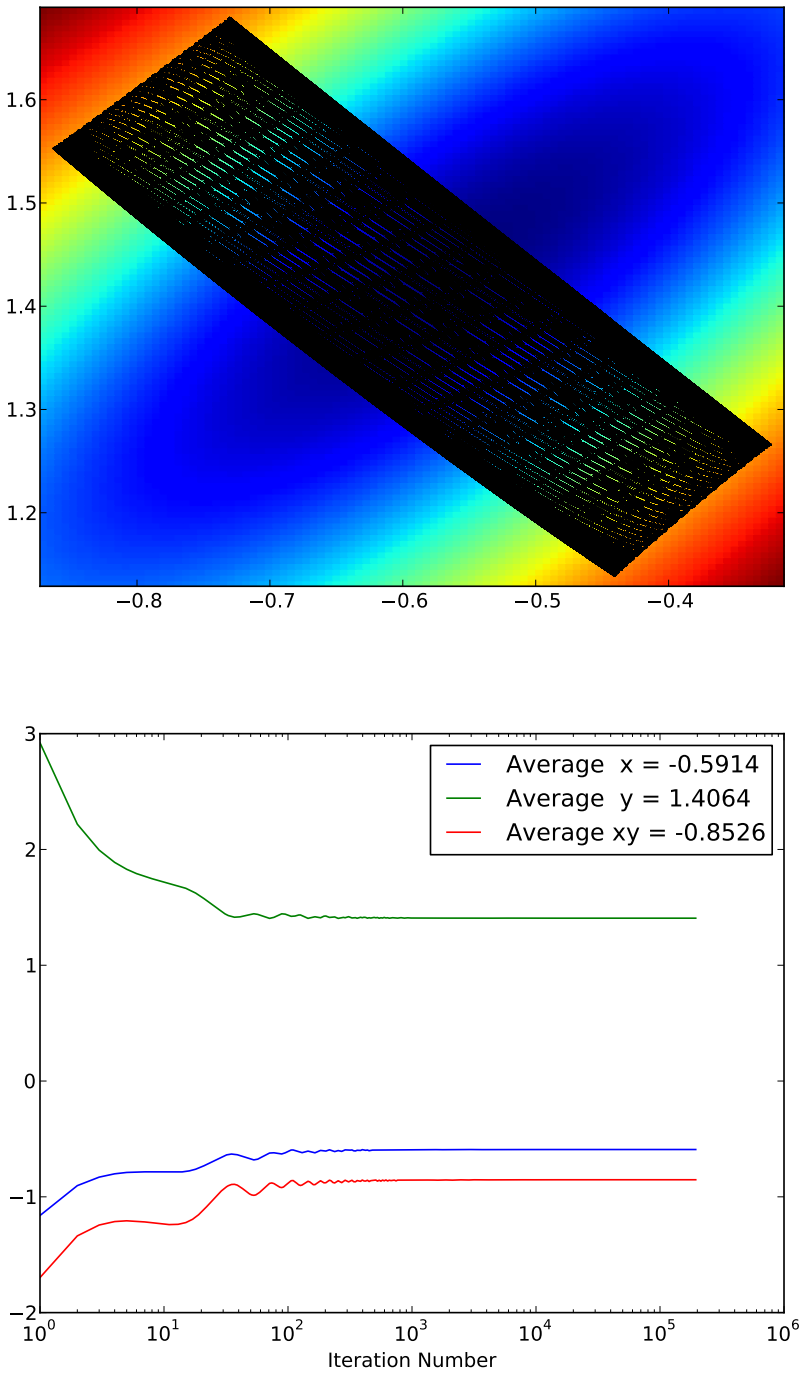

Figure 3: $E = -39$: Sometimes the particle finds symmetries in the potential field and ends up repeating the same path over and over. The small gaps in the path demonstrate this here.

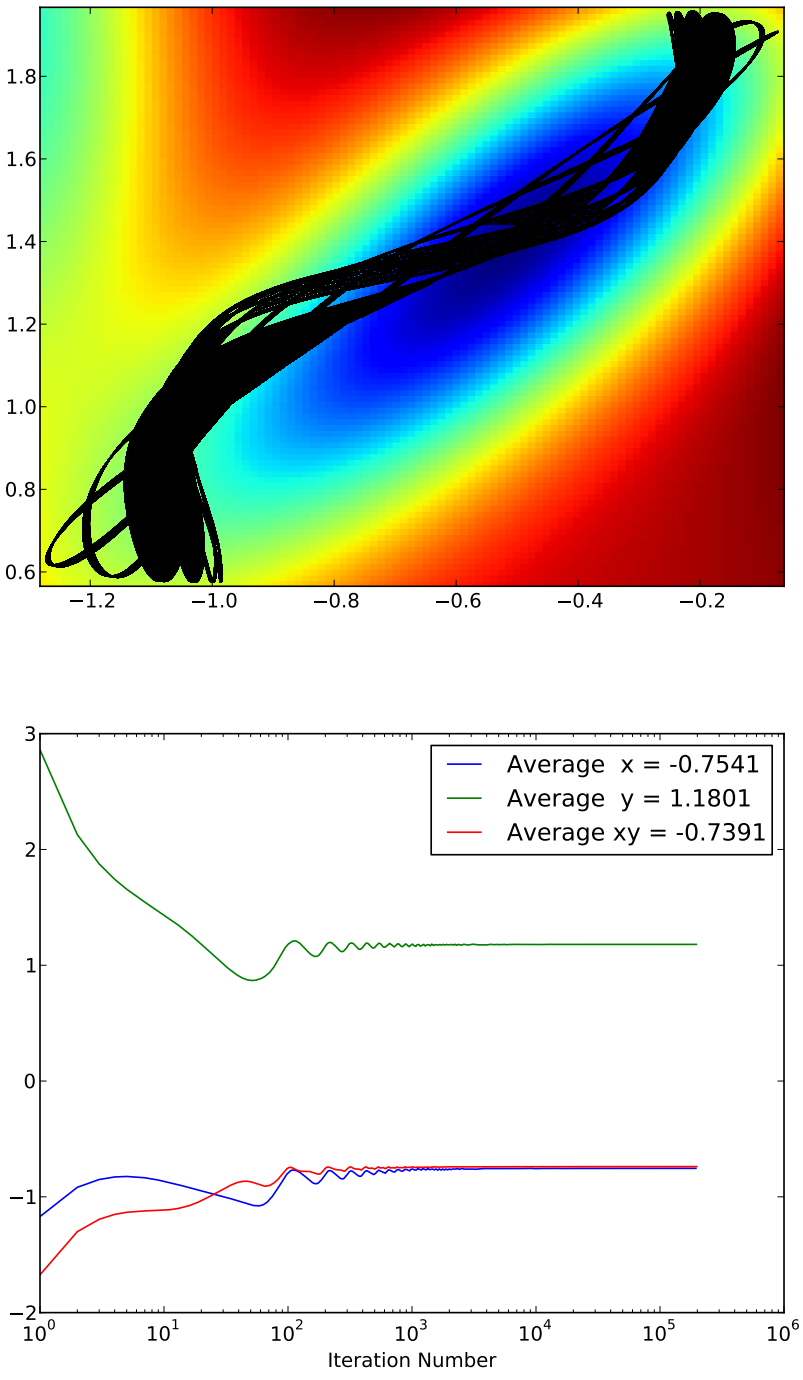

Figure 4: $E = -39$: Here is a more extreme case of the previously noted behavior. Notice the distinctive ribbon path that the particle follows.

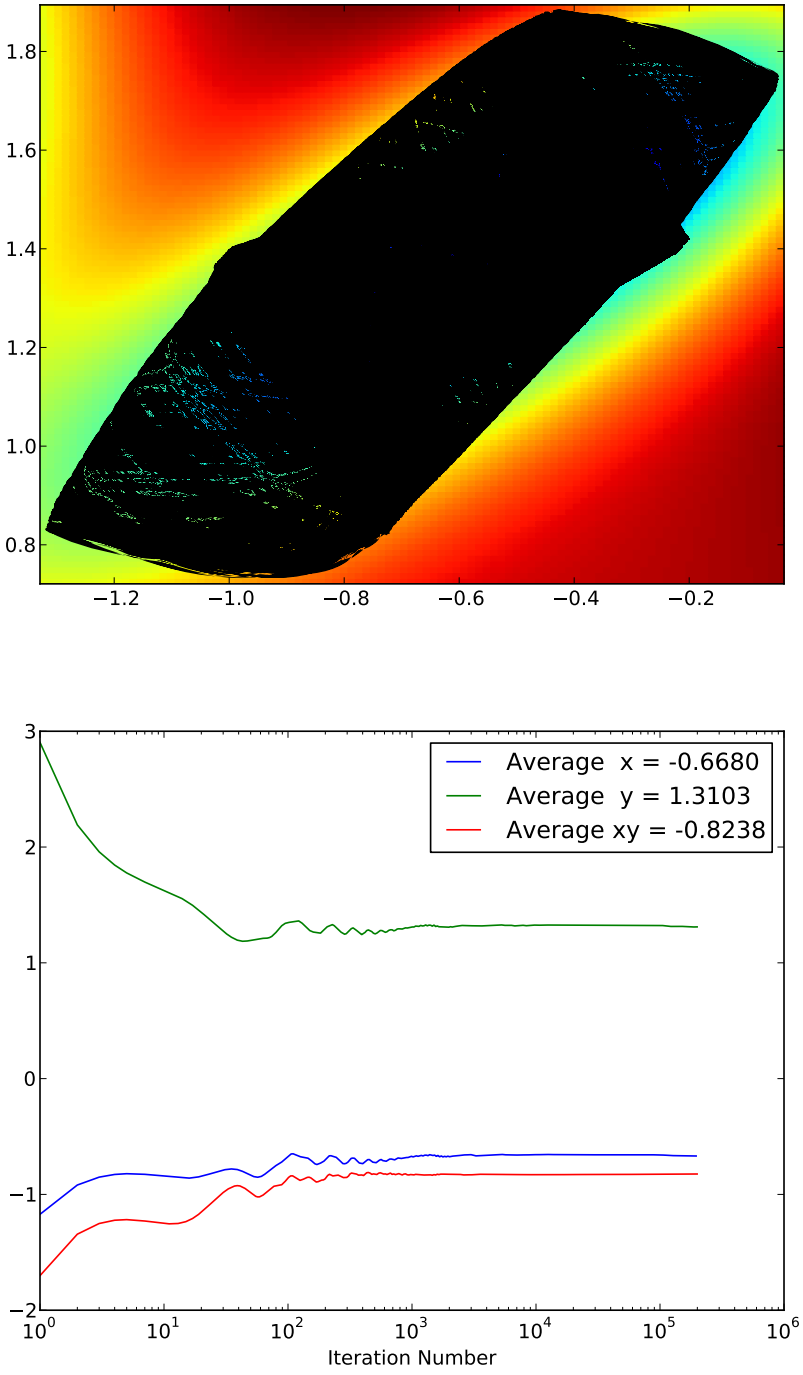

Figure 5: $E = -39$: Other times the particle falls into a seemingly random pattern, but is still never able to leave the well.

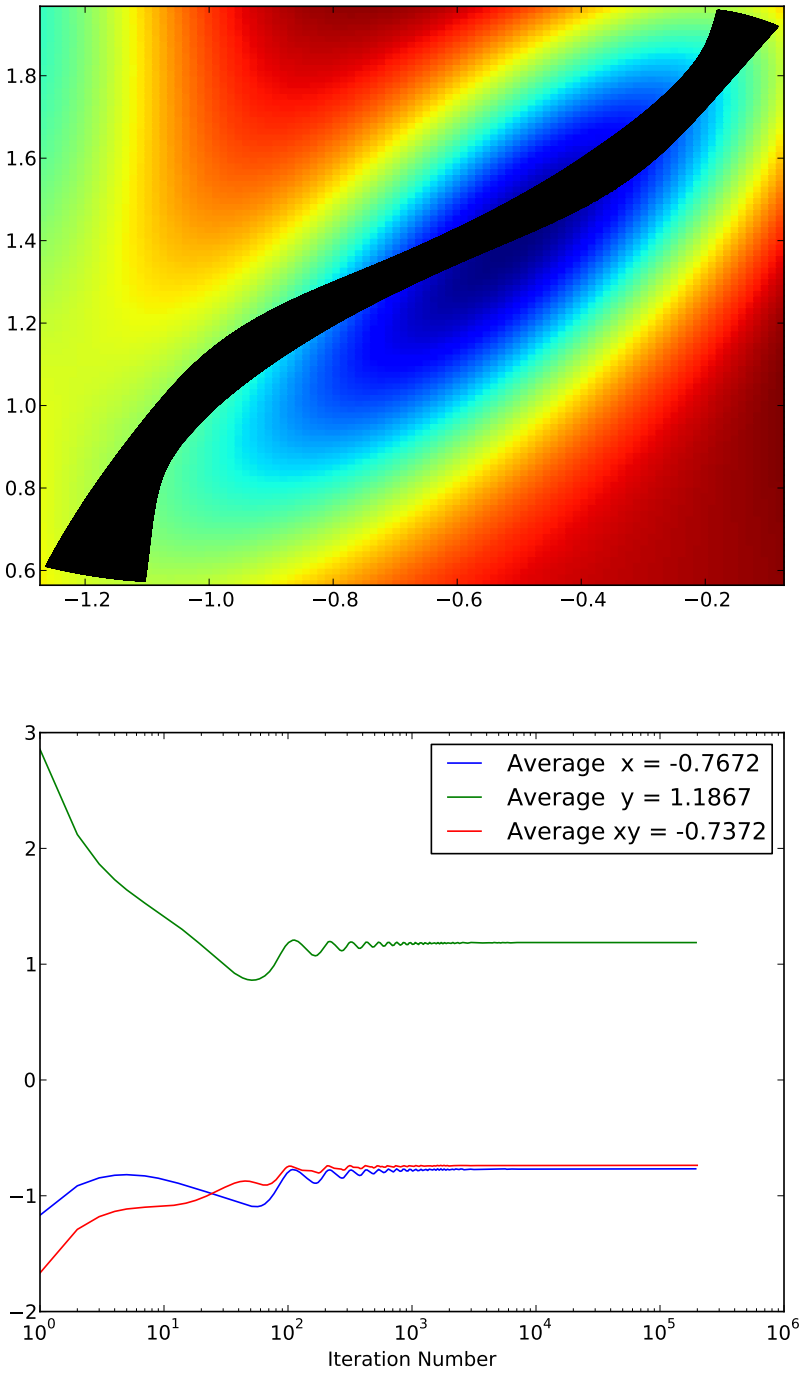

Figure 6: $E = -39$: In other cases we get a small ribbon that gets sampled.

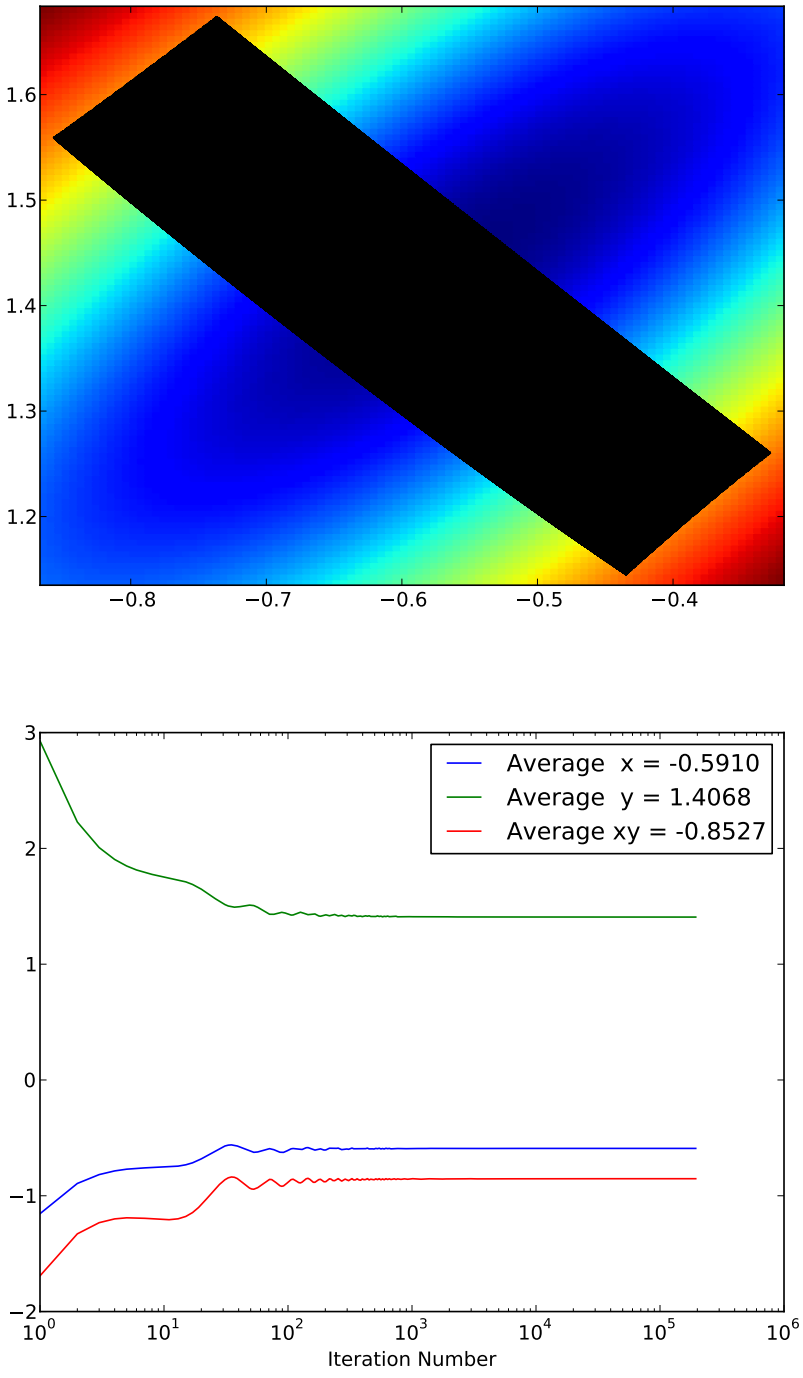

Figure 7: $E = -39$: Another common pattern was to get rectangular domains of sampled points.

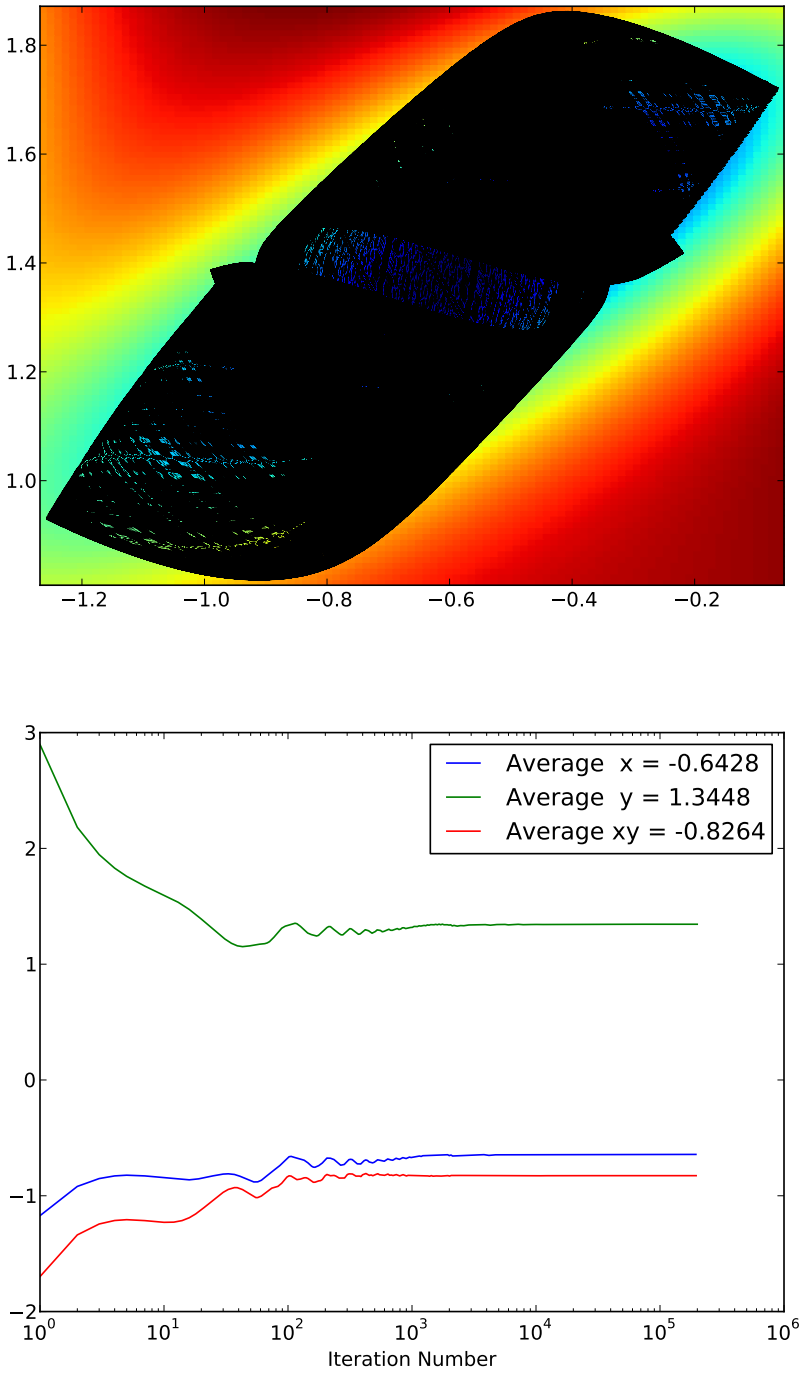

Figure 8: $E = -45$: This pattern looks similar to 5, except the particle does not travel as far.

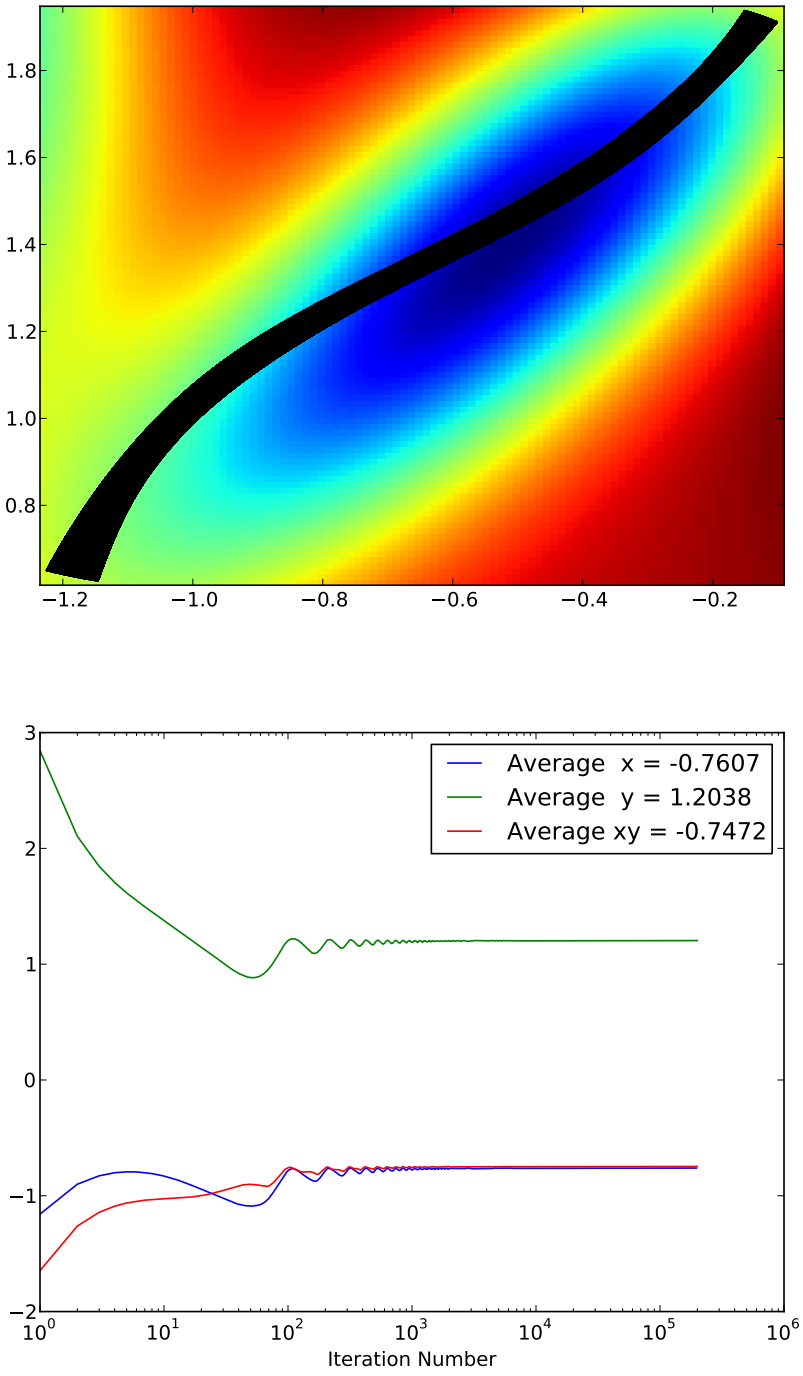

Figure 9: $E = -45$: As before, sometimes we get narrow ribbons of sampled points.

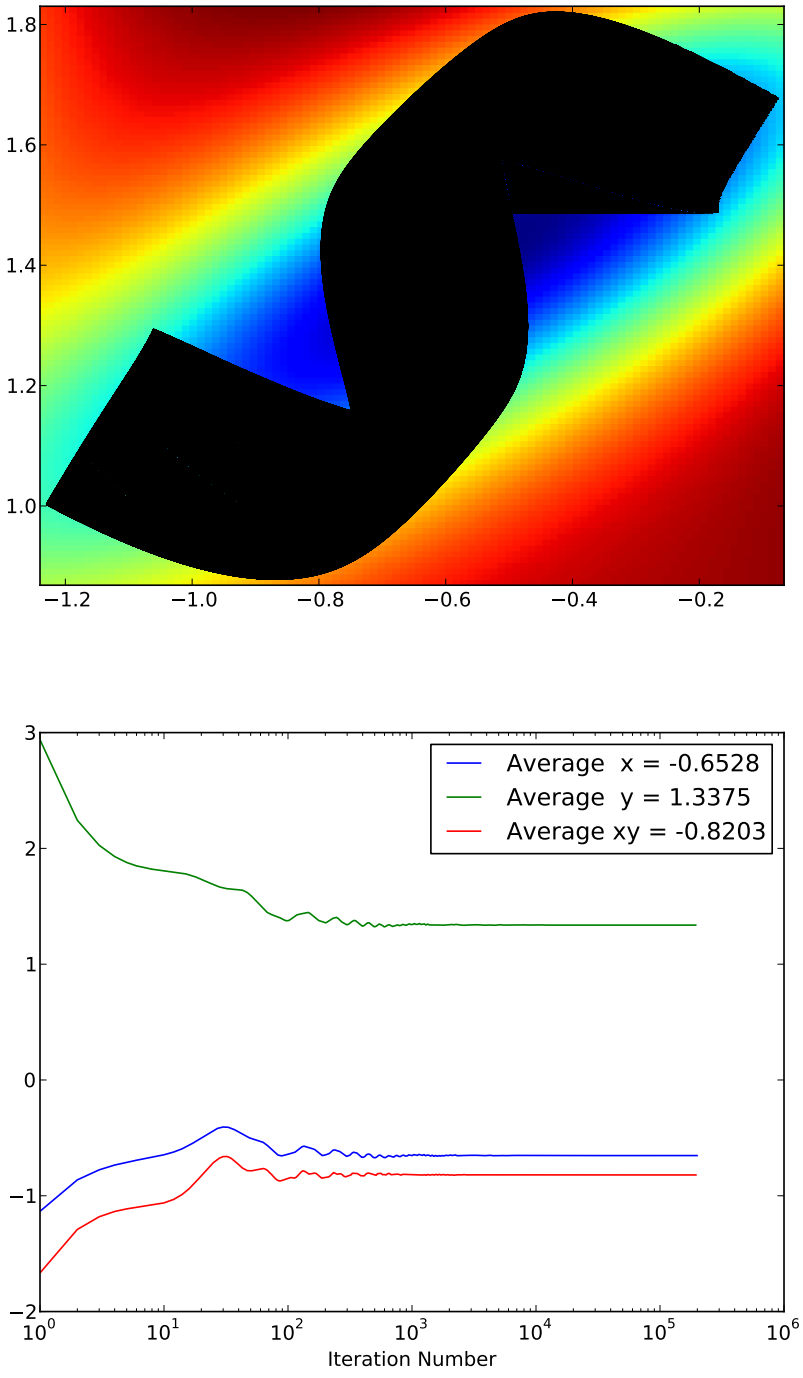

Figure 10: $E = -45$: Other times we get S shapes.

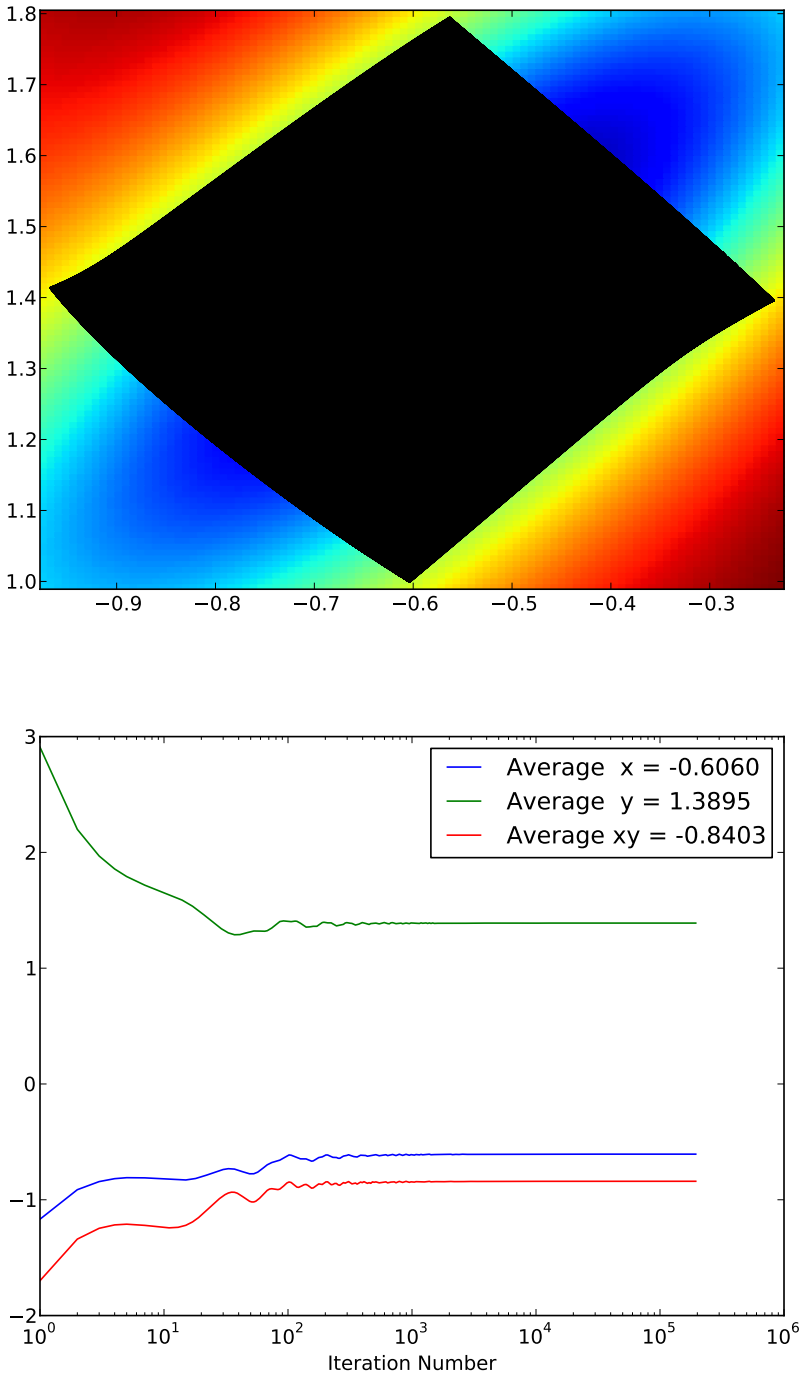

Figure 11: $E = -45$: And as before, sometimes we get squares or rectangles.

HW5.py

```
# Statistical Mechanics Homework 5
# Spring 2011
# Written by Truman Ellis

from pylab import *
close('all')

def gauss1(x, y):
    dg = zeros(2)
    d2g = zeros((2,2))
    g = -200*exp(-(x-1)*(x-1)-10*y*y)
    dg[0] = -2*(x-1)*g
    dg[1] = -20*y*g
    d2g[0,0] = (-2+4*(x-1)*(x-1))*g
    d2g[1,1] = (-20 + 400*y*y)*g
    d2g[0,1] = 40*(x-1)*y*g
    d2g[1,0] = d2g[0,1]
    return (g, dg, d2g)

def gauss2(x, y):
    dg = zeros(2)
    d2g = zeros((2,2))
    g = -100*exp(-x*x-10*(y-.5)*(y-.5))
    dg[0] = -2*x*g
    dg[1] = -20*(y-.5)*g
    d2g[0,0] = (-2+4*x*x)*g
    d2g[1,1] = (-20 + 400*(y-.5)*(y-.5))*g
    d2g[0,1] = 40*x*(y-.5)*g
    d2g[1,0] = d2g[0,1]
    return (g, dg, d2g)

def gauss3(x, y):
    dg = zeros(2)
    d2g = zeros((2,2))
    g = -170*exp(-6.5*(x+.5)*(x+.5)+11*(x+.5)*(y-1.5)-6.5*(y-1.5)*(y-1.5))
    dg[0] = (-13*(x+.5)+11*(y-1.5))*g
    dg[1] = (-13*(y-1.5)+11*(x+.5))*g
    d2g[0,0] = ((-13*(x+.5)+11*(y-1.5))**2-13)*g
    d2g[1,1] = ((-13*(y-1.5)+11*(x+.5))**2-13)*g
    d2g[0,1] = (11+(-13*(x+.5)+11*(y-1.5))*(-13*(y-1.5)+11*(x+.5)))*g
    d2g[1,0] = d2g[0,1]
    return (g, dg, d2g)

def gauss4(x, y):
    dg = zeros(2)
    d2g = zeros((2,2))
    g = 15*exp(.7*(x+1)*(x+1)+.6*(x+1)*(y-1)+.7*(y-1)*(y-1))
    dg[0] = (1.4*(x+1)+0.6*(y-1))*g
    dg[1] = (1.4*(y-1)+0.6*(x+1))*g
    d2g[0,0] = ((1.4*(x+1)+.6*(y-1))**2+1.4)*g
    d2g[1,1] = ((1.4*(y-1)+.6*(x+1))**2+1.4)*g
    d2g[0,1] = (.6+(1.4*(x+1)+.6*(y-1))*(1.4*(y-1)+.6*(x+1)))*g
    d2g[1,0] = d2g[0,1]
    return (g, dg, d2g)

def muller((x, y)):
    (g1, dg1, d2g1) = gauss1(x, y)
    (g2, dg2, d2g2) = gauss2(x, y)
    (g3, dg3, d2g3) = gauss3(x, y)
    (g4, dg4, d2g4) = gauss4(x, y)
    return g1 + g2 + g3 + g4

def dmuller((x, y)):
    (g1, dg1, d2g1) = gauss1(x, y)
    (g2, dg2, d2g2) = gauss2(x, y)
    (g3, dg3, d2g3) = gauss3(x, y)
    (g4, dg4, d2g4) = gauss4(x, y)
    return dg1 + dg2 + dg3 + dg4

def VerletIntegrator(Xi, Vi, m, dt):
    dUi = dmuller(Xi)
    Xip1 = Xi + Vi*dt - 0.5*dt*dt/m*dUi
    dUip1 = dmuller(Xip1)
    Vip1 = Vi - 0.5*dt/m*(dUi + dUip1)

    return (Xip1, Vip1)

# Initialize particle at lowest energy point
m = 1.
E = -39
X0 = (-0.55822362835932715, 1.4417258468422827)
U0 = muller(X0)
# Calculate required velocity to produce E0
V2 = 2/m*(E-U0)
```

```
# Randomly set Vx and Vy as fractions of V2
Vx2 = rand()*V2
Vy2 = V2 - Vx2
# Also randomize signs of velocities
V0 = (sign(.5-rand())*sqrt(Vx2), sign(.5-rand())*sqrt(Vy2))

N = 200000
dt = 4e-3
X = zeros((N,2))
V = zeros((N,2))
Xave = zeros(N)
Yave = zeros(N)
XYave = zeros(N)
X[0] = X0
V[0] = V0
Xave[0] = X0[0]
Yave[0] = X0[1]
XYave[0] = X0[0]*X0[1]
for n in range(1,N):
    (X[n], V[n]) = VerletIntegrator(X[n-1], V[n-1], m, dt)
    Xave[n] = average(X[0:n+1,0])
    Yave[n] = average(X[0:n+1,1])
    XYave[n] = average(X[0:n+1,0]*X[0:n+1,1])

xmin = min(X[:,0])-.01
xmax = max(X[:,0])+.01
ymin = min(X[:,1])-.01
ymax = max(X[:,1])+.01
# Compute background pseudocolor plot of Muller potential
nx = 100
ny = 101
mu = zeros( (nx,ny) )
x = linspace(xmin, xmax, 100)
y = linspace(ymin, ymax, 101)
for i in range(0,nx):
    for j in range(0,ny):
        mu[i,j] = muller((x[i],y[j]))

figure(1)
pcolor(x,y,mu)
hold(True)
plot(X[:,0],X[:,1],'k', linewidth=1)
axis([xmin,xmax,ymin,ymax])

figure(2)
hold(True)
plot(Xave)
plot(Yave)
plot(XYave)
legend(('Average x = %.4f'%(Xave[-1],),'Average y = %.4f'%(Yave[-1],),'Average \
    xy = %.4f'%(XYave[-1],)),loc='best')
show()

#Calculate energy error
Ef = muller(X[-1]) + .5*m*sum(V[-1]**2)
EnergyError = abs((E - Ef)/E)
print 'Energy error: %.6f'%(EnergyError,)
```