

# Lab 10 - Sand[piles] and Earth[quakes]

Thursday 2 November, 2006 - Due: Thursday 9 November

REVISED

In this lab, you will study aspects of “self-organized critical phenomena”, which have been used to study general “universal” properties of some interesting complex physical systems. Here, you’ll focus on two one-dimensional models: the sandpile and the earthquake.

*As always, please include in your writeup any sections of Python code you write.*

♣ Read thoroughly. Every sentence has been very carefully constructed to guide you through this lab.

## 1 Visualizing the Toppling Over of Sandpiles

In lecture, you were introduced to a model of building a [Bak, Tang, Wiesenfeld] sandpile, subject to a simple set of rules: While adding a grain of sand at the origin, if the origin’s stack becomes unstable by being too high compared to its neighboring stack, it may partially collapse and distribute some grains to its neighbors—an avalanche. Possibly surprisingly, in spite of this seemingly complex dynamical behavior, the sandpile, by itself, tends towards a statistically time-independent state.

Open [lab10.sandpile.py](#) and run it.

Let’s add a little VPython code to help us visualize the actual process of building the sandpile.

Use [lab10\\_sandpile\\_mod.py](#) to grab snippets of code. Use the `#` comment markers to help you insert this code. I’ll discuss this in lab.

- ( )★ Write (copy-paste) some Vpython code to plot the sandpile height profile.**
- (Q1)★ Take a couple of screen-captures showing the running log-log plot and the sandpile height profile.**
- (Q2)★ When all of the grains have been dropped, take a screen-capture of the final log-log plot and the final sandpile height profile. Copy-paste the log-log data in the shell-window. Using a suitable subset of those data points, estimate of the slope of the best-fitting line. Report your value for the “power-law”-exponent for topples in the sandpile.**
- (Q3)★ Repeat the previous question for a different values of `sites` .  
Choose a larger value, capture, and analyze.  
Choose a smaller value, capture, and analyze.**

## 2 A simple physical model of an earthquake

Open [lab10.BKearthquake.py](#) and run it.

What you see will yield a continuous mechanical model of an earthquake.

No dynamics is being displayed because it’s not in there yet. That’s your job.

Based on the discussion in Giardano’s Computational Physics

There is an array of masses, whose **neighbors are joined by springs with spring constant**  $k_c$ . By drawing a free-body diagram of one mass, you can see that

$$\begin{aligned} f_{\text{springs}} &= f_{\text{due to right spring},x} + f_{\text{due to left spring},x} \\ &= -k_c(x_i - x_{i+1}) - k_c(x_i - x_{i-1}) \end{aligned} \quad (1)$$

Since, in this model, **the masses at each end are only connected to one spring**, the system is subject to so-called “**free boundary conditions**”, which you can code by saying

- The left-end mass “has **itself**” to its left.  
(That is, the left neighbor of “0” is “0”)
- The right-end mass “has **itself**” to its right.  
(That is, the right neighbor of “ $N - 1$ ” is “ $N - 1$ ”)

[What this does is that it effectively removes that spring since its term that looks like  $-k_c(x_i - (x_i)) = -k_c(0) = 0$ .]

Furthermore, each mass is connected by string to an upper plate, which will slide with constant velocity  $v_0$ . The connection to the upper plate makes each string-mass pair an **ideal pendulum**, which can be treated like a horizontal spring with spring constant  $k_p$ , where the displacement is the horizontal component of the displacement from the pendulum support above.


$$\begin{aligned} f_{\text{pendulum on mass } m_i} &= -k_p(x_i - x_{i,\text{support}}) \\ &= -k_p(x_i - (x_i + v_0t)) \\ &= -k_p(x_i - v_0t) \end{aligned} \quad (2)$$

The masses also ride on a rough plate, which applies **frictional forces** on the block.

- If the object is **at rest** (i.e., with speed  $|v| = 0$ ) relative to the surface, the **static friction force** can apply *any opposing force necessary to prevent impending motion, up to a maximum value*  $F_0$ :

$$f_{\text{static friction on mass } m_i} \leq F_0 \quad (3)$$

- So, if the friction force (i.e., what is needed to balance the other applied forces) doesn’t exceed the maximum value, the net-force is zero. Since this means the acceleration is zero, we must have that the object remains at rest (i.e., the resulting velocity is zero).
- ...otherwise, there is a nonzero net-force (since in this case  $f_{\text{static friction on mass } m_i} = F_0$ ), and therefore nonzero acceleration. So, the resulting velocity is determined by this nonzero acceleration.

- If [or, better,  **else:** Since] the object is **moving** relative to the surface, the **kinetic friction force** can apply a force which “opposes the direction [sign] of the velocity”. One model for this opposing kinetic friction force is

$$f_{\text{kinetic friction on mass } m_i} = -\frac{F_0 \text{ sign}(v_i)}{1 + \left| \frac{v_i}{v_f} \right|}$$




When programming, consecutive statements “if A then B” followed by “if not-A then C” is logically different from “if A then B else C”.

where  $v_f$  is a parameter that determines the strength of the velocity-dependence for this friction force. However, we will simply model this as

$$= -\frac{F_0 \operatorname{sign}(v_i)}{2} \quad \text{by choosing } v_f = v_i \quad (4)$$

One important feature (for the sake of the simulation) is that if the velocity changes sign, we should set the resulting velocity to zero. (The reason for this is related to the non-infinitesimal timestep `dt`. With a finite `dt`, the simulation can miss the zero-velocity [which must be there if the velocity changes sign], which could lead to some instabilities in the simulation.)

 added in revision

Your job is to write Python code in the `update_kinematics` function to give the system these dynamics.

Let me suggest that you use these constants near the top of your program

```
KC=250.
KP=40.
F0=50.
v0=.01
```

We can think of setting  $m_i = 1$  for all  $i$ . So, we can ignore using a variable for the masses.

Let me start you off with this snippet:

within the `while(true):` loop in `update_kinematics`

```
####
####
##
##
for i in arange(len(x)):
    xnew[i] = x[i]+v[i]*dt

    .....
    ...(more of your code here)
    .....
```

which does the first step in the Euler method by updating the position variables for each block. For [object-oriented] convenience, we use `len(x)`, the number of elements (“the length”) of the `x`-array, for the number of blocks in the model.

**(\*) You need to use the numbered equations above (and the section on the boundary conditions) to write does the steps for the velocity variables.**

Here’s another hint to get you going. Define two variables

```
iplus=i+1
iminus=i-1

.....
...(more of your code here)
.....
```

Finally, when you are all done with those steps, you have to “commit updates” of the running position and velocity variables:

```
        if loop_again==false:
            break          ##do... while(loop_again)
    ##
    ####

    #commit updates
    for i in arange(len(x)):
        block[i].x=float(xnew[i])
        block[i].vel=float(vnew[i])
```

(Q3)★ Take some screen captures. Try to use the scatter plot of the earthquake magnitudes to determine the power-law exponent. Use a line of best fit with an appropriate set of data points.