```python
from numpy import *
from vpython import *
from random import *
from time import *

# constants
m = 1.0
kp = 40.0
kc = 250.0
f_0 = 50.0
v_0 = 0.01

# changeable constants
# number of blocks
n = 500
# long timestep
dt1 = 0.03
# short timestep
dt2 = 0.003
# final time
t_f = 1000.0
# number of runs
n_events = 10
# window size
window_w = 400
window_h = 400
# plus or minus factor to initial positions
rand = 0.001
# use uniformly distributed start or not
equil = False

# returns force of friction
def calculate_friction(f_opposed, v_i_t):
    # if block not moving
    if (v_i_t == 0.0):
        # if force on block less than equal to f_0
        if (f_opposed <= f_0):
            return -f_opposed
        # if force on block greater than f_0
        else:
            return sign(-f_opposed) * f_0
    else:
        # calculate kinetic friction
        return (-((f_0 * sign(v_i_t)) / (1.0 + abs(v_i_t))))

# return array of zeros
def initialize_positions_uniform(positions):
    return positions

# return array of positions adding or subtracting rand from each one
def initialize_positions_random(positions, rand):
    for i in range(n):
        switch_value = randint(0, 1)
        if (switch_value == 0):
            positions[i] = rand
        else:
            positions[i] = -rand
    return positions

# return array of random masses in hardcoded range
def initialize_masses_random(masses):
    for i in range(n):
        masses[i] = uniform(0.5, 2.0)
    return masses

# return array of random kcs in hardcoded range
```

```python
def initialize_kcs_random(kcs):
    for i in range(n):
        kcs[i] = uniform(200.0, 300.0)
    return kcs

# return array of random kps in hardcoded range
def initialize_kps_random(kps):
    for i in range(n):
        kps[i] = uniform(30.0, 50.0)
    return kps

# return value for force on block i
def calculate_force(i):
    # if first block then no block to the left
    if (i == 0):
        f_opposed = kcs[i] * (positions[i + 1] - positions[i])
    # if last block then no block to the right
    elif (i == n - 1):
        f_opposed = kcs[i] * (positions[i - 1] - positions[i])
    else:
        f_opposed = kcs[i] * (positions[i + 1] + positions[i - 1] - 2.0 * positions[i])
    # add force from leaf spring
    f_opposed += kps[i] * (v_0 * t_i - positions[i])
    # calculate and add friction
    ff = calculate_friction(f_opposed, velocities[i])
    return f_opposed + ff

# returns value for velocity of block i
def calculate_velocity(i, f_i):
    # calculate velocity of next step
    v_i = (f_i * dt) / masses[i] + velocities[i]
    # if velocity is not zero and it changes direction
    if ((velocities[i] != 0.0) and (sign(velocities[i]) != sign(v_i))):
        return 0.0
    return v_i

# returns value for position of block i
def calculate_position(i, v_i_t1):
    return (v_i_t1 * dt) + positions[i]

# plots the distribution
def plots(magnitudes):
    # set up buckets
    diff = 0.5
    max_m = int(max(magnitudes)) + 1
    min_m = int(min(magnitudes)) - 1
    magnitude_counts = [0] * (int((max_m - min_m) / diff) + 1)
    # iterate through magnitudes and put in buckets
    for i in magnitudes:
        index = int((i - min_m) / diff)
        magnitude_counts[index] = magnitude_counts[index] + 1
    # print info
    print(magnitude_counts)
    print(sum(magnitude_counts))
    # plot on logarithmic y scale
    distribution = graph(align = 'left', xmin = min_m, xmax = max_m, width = window_w, height = window_h, \
                    title = 'Distribution of magnitudes', xtitle = 'Magnitude', ytitle = 'Number of events')
    g_distribution = gdots(color=color.cyan)
    for i in range(len(magnitude_counts)):
        if (magnitude_counts[i] == 0):
            g_distribution.plot(pos = [min_m + i * diff, magnitude_counts[i]])
        else:
            g_distribution.plot(pos = [min_m + i * diff, math.log(magnitude_counts[i])])

# get start time
start = time()
```

```python
magnitudes = []

# iterate through runs
for k in range(n_events):

    # initialized constants
    # array of positions for each block
    positions = resize(array([0.0]), n)
    # array of velocities for each block
    velocities = resize(array([0.0]), n)
    # array of forces for each block
    forces = resize(array([0.0]), n)
    # array of masses for each block
    masses = resize(array([0.0]), n)
    masses = initialize_masses_random(masses)
    # array of kc factors for each block
    kcs = resize(array([0.0]), n)
    kcs = initialize_kcs_random(kcs)
    # array of kp factors for each block
    kps = resize(array([0.0]), n)
    kps = initialize_kps_random(kps)

    t_i = 0.0
    moment = 0.0
    dt = dt1

    # initialize positions
    if (equil):
        positions = initialize_positions_uniform(positions)
    else:
        positions = initialize_positions_random(positions, rand)

    # keeps track of switching dt
    first = False
    # keeps track of dt having been switched until quake start
    second = False

    # for this amount of time
    while (t_i < t_f):
        # copy arrays so as not to overwrite them
        temp_positions = positions.copy()
        temp_velocities = velocities.copy()
        temp_forces = forces.copy()

        # keeps track of all velocities being zero
        zeros = True

        # iterate through blocks
        for i in range(n):
            f_i = calculate_force(i)
            v_i_t1 = calculate_velocity(i, f_i)

            # if block is moving
            if (v_i_t1 != 0.0):
                # if using long timestep
                if (dt == dt1):
                    # go back
                    t_i = t_i - dt
                    # switch to short timestep
                    dt = dt2
                    first = True
                    zeros = False
                    break
                # marks quake start
                if (second):
                    second = False
```

```
                zeros = False

            x_i_t1 = calculate_position(i, v_i_t1)

            # put calculated values in arrays
            temp_forces[i] = f_i
            temp_velocities[i] = v_i_t1
            temp_positions[i] = x_i_t1

        # if not switch (when go back in time)
        # update arrays
        if (not first):
            forces = temp_forces
            velocities = temp_velocities
            # update moment
            moment += sum(velocities) * dt
            positions = temp_positions

        # after switched
        else:
            first = False
            second = True

        # marks that quake is over
        if (zeros and not second):
            if (moment != 0.0):
                magnitudes.append(math.log(moment))
                moment = 0.0
            # switch to long timestep
            dt = dt1

        t_i += dt

    # keep track of time
    if (k % 1 == 0):
        print(k, ': ', time() - start)

plots(magnitudes)
print('Finish: ', time() - start)
```