# Computational Physics Final Project

Sage Li and Eathan Xu

April 2024

## 1 Introduction

For our final project, we investigated the dynamics of a ball bouncing on a vibrating table. The height of the vibrating table will follow a periodic function. In this project, we will compare two different periodic functions: A sinusoidal and a periodic triangle. The ball will be dropped from an initial height, and will follow standard projectile motion. When the ball hits the table, we will recalculate a rebound velocity based on the ball's impact velocity, the table's velocity, and the coefficient of restitution. Using this rebound velocity, we recalculate the motion of the ball. The process of the ball hitting the table and rebounding will repeat until the end of the simulation time.

The objective of studying the problem of a bouncing ball on an oscillating table is to explore different regimes of stability and chaos. As we have seen in class, we can visualize these different regimes through bifurcation diagrams. We plan on creating our own bifurcation diagrams by varying the frequency of the table's oscillation and recording the peak height of the ball. Additionally, we will animate the ball rebounding off the table with two plots. One plot will showcase the heights of both the table and the ball as a function of time, while another plot will showcase a physical representation of the interactions between the table and the ball.

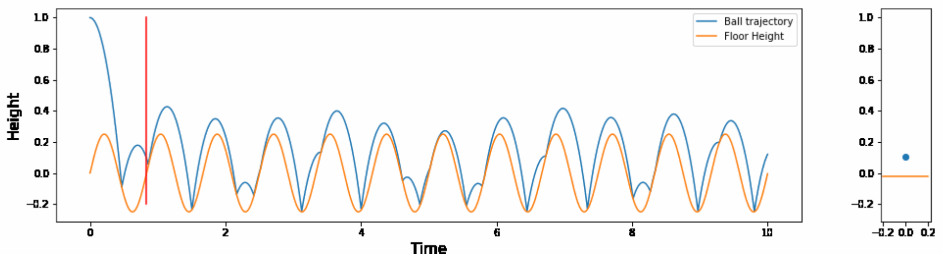
## 2 Problem-Solving Method

Our methodology for solving the problem was straight-forward. For each time step of the simulation, we will calculate the ball's height and the table's height. If the ball's height falls below the table's height, a collision has occurred. We will update the ball's velocity to a new rebound velocity. If no collision occurs, then the ball will continue its standard free fall projectile motion path. We will repeat this until the end of the simulation time. Additionally, we will generate bifurcation diagrams by running our simulation for various table frequencies and tracking the peak height attained by the ball.

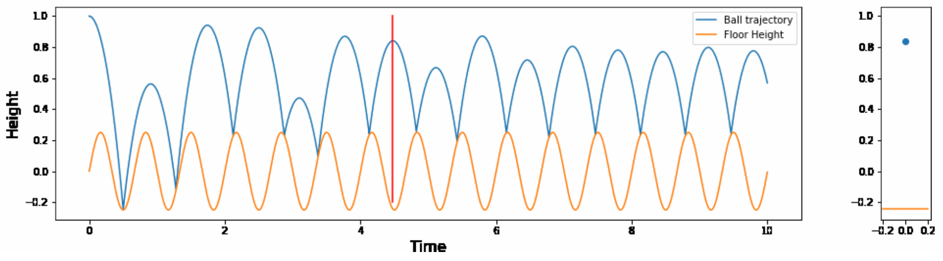
## 3 Results Discussion

### 3.1 Sinusoidal Table Height

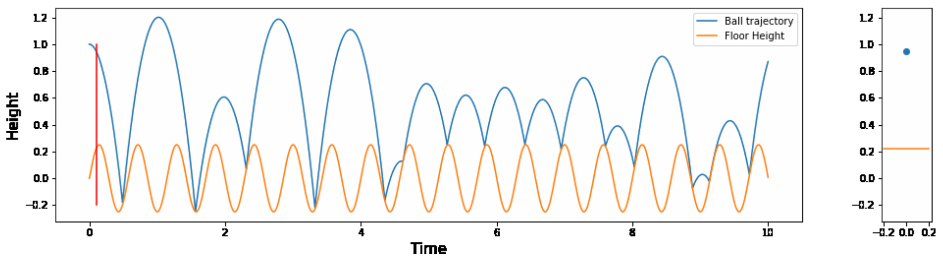
Our results showcase interesting dynamics within the bifurcation diagrams for both the sinusoidal and triangular periodic table heights. For the case of the sinusoidal, we identified three different regimes. We called the first one the "hugging" regime. With our initial conditions, this occurred at low frequency values, typically below 1.25. We call this regime hugging as the table's



(a) An example of a "hugging" regime.



(b) A stable regime.



(c) A chaotic regime.

Figure 1: Observed regimes in a sinusoidal oscillating table.

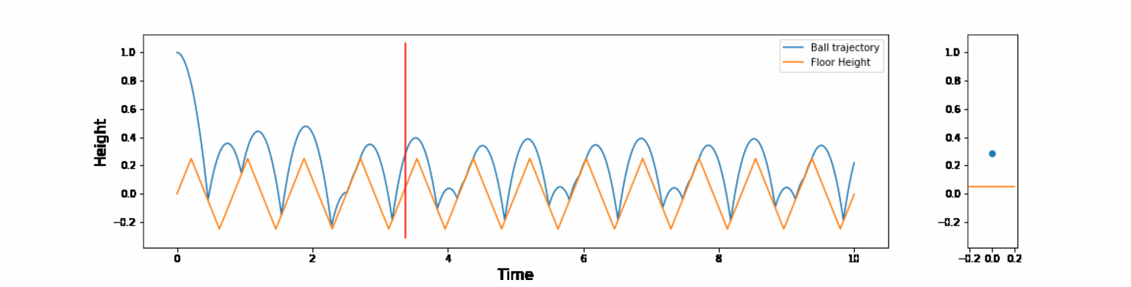
oscillation do not provide enough energy back into the ball to sustain higher bounces. This causes the ball's height to hug the table as the simulation runs. There seems to be some structure in this regime, but it is generally hard to distinguish.

The next is the stable regime. This regime occurred at frequencies around 1.25 to 1.7. Here, balls will be able to sustain their heights as the table oscillates throughout the simulation time. Additionally, this regime showcases a branching of the bifurcation diagram. We also see a few transient structures around the 1.55 1.6 frequency range, which almost seem to resemble bifurcation diagrams of their own.

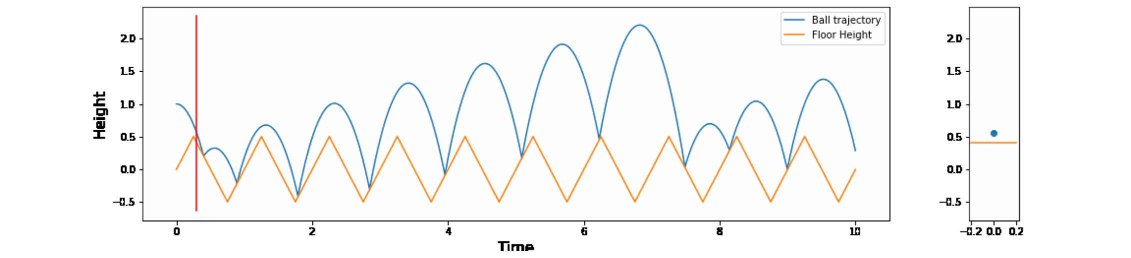
The last regime is the chaotic regime. This occurred at frequency values above 1.7. In this regime, small changes to initial parameters- like table oscillation frequency, impact the behavior of the ball a lot.

## 3.2 Periodic Triangular Table Height

As with the sinusoidal case, we find three different regimes through the bifurcation diagram generated. The first regime corresponds to another hugging regime. The ball will hug the table as



(a) An example of a "hugging" regime.



(b) An increasing bounce height regime.

Figure 2: Observed regimes in a triangularly oscillating table.

the table does not provide enough energy to rebound the ball higher.

The second regime is the stable regime. For the triangular case in particular, the ball will gain height from each rebound before losing it and starting over. The height of the ball is easily predictable as it gains and loses height periodically.

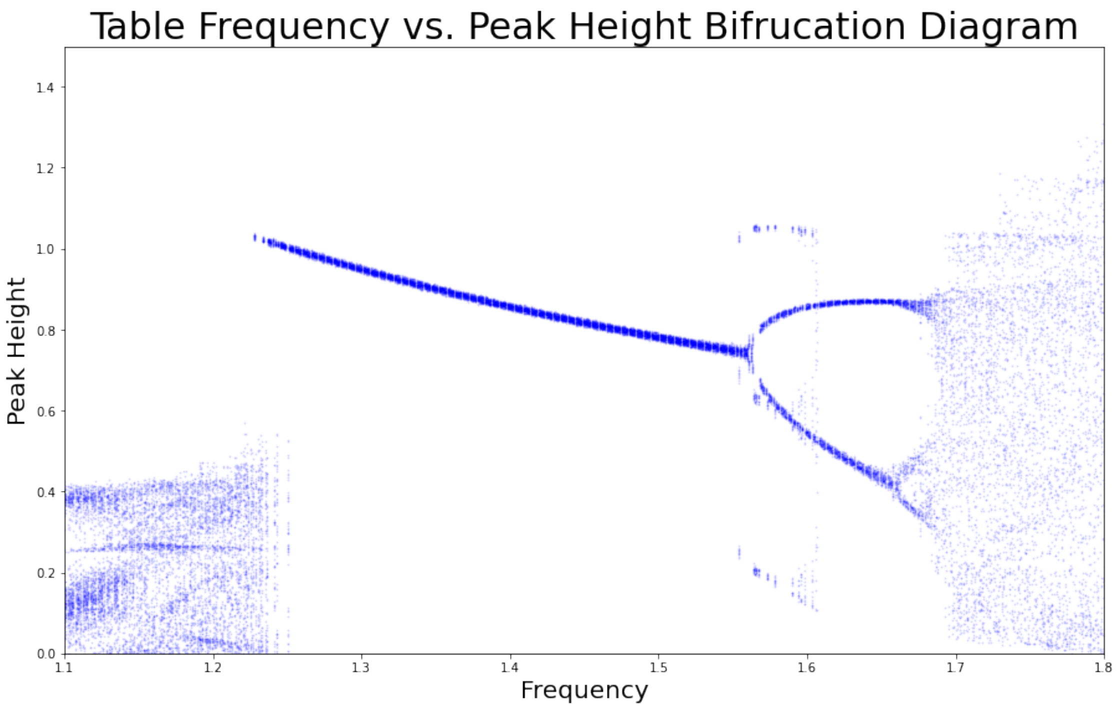
The last regime is the chaotic regime. Just as we had seen with the sinusoidal case, small changes in the initial conditions drastically influence the dynamics.

## 4 Code Discussion

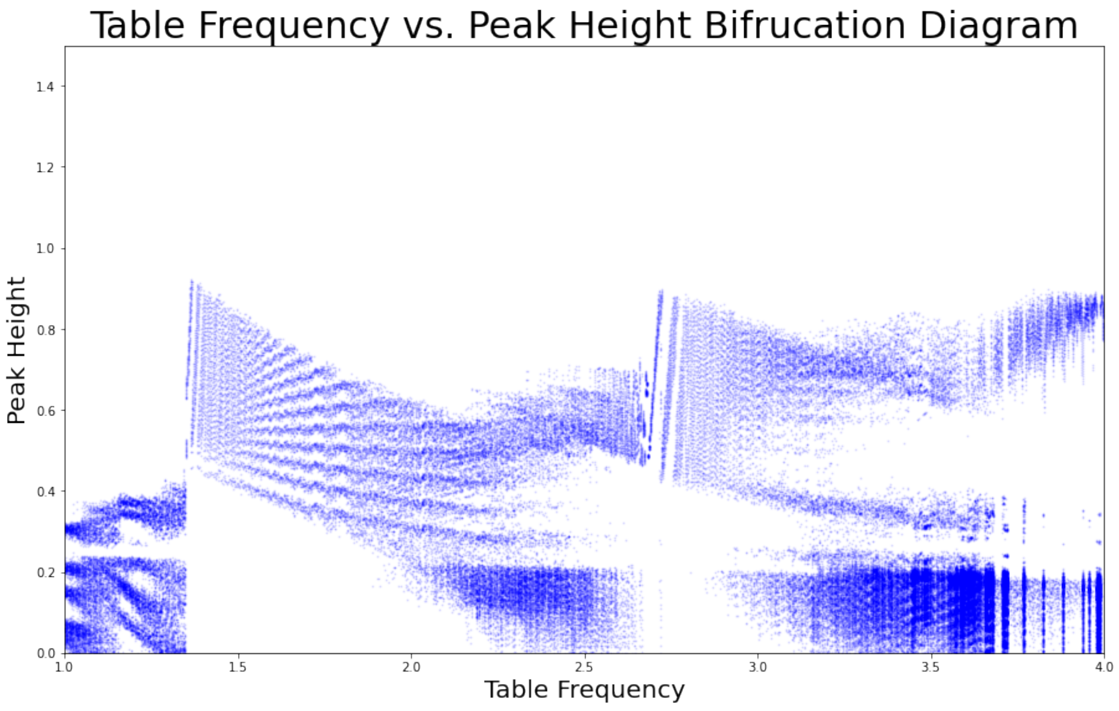
We chose to code our system in Python. We first set some initial conditions for both the table and the ball. Additionally, we choose our coefficient of restitution to be 0.8. We create arrays to represent the time elapsed, the height of the ball, the velocity of the ball, the height of the table, and an auxiliary array to keep track of the peak height of the ball. We also set a value of our timestep $dt$ as we plan on using Euler's method to update the positions of the ball and the table.

We create a while loop that will run to the length of the simulation time. For each timestep of the simulation, we calculate if a collision between the ball and table occur. This was done by simply checking if the ball's height will be less than the table's height. If it is, then we need to recalculate the velocity of the ball to its new rebound velocity. If there is no collision, then the ball will simply follow projectile motion. Additionally, if a collision occurs, we check the ball's current velocity to ensure that the sign of the rebound velocity is properly computed. All relevant values are stored in their respective arrays for plotting once the max simulation time is reached.

To generate the bifurcation diagrams, we create an array of possible frequency values to test. For each possible frequency value, we run the simulation of the ball and oscillating table up to an arbitrary maximum time. Once this time limit is reached, we find the peak height that the ball reached and store them into an array. With both the frequency array and the peak array, we are



(a) Sinusoidal Oscillations.



(b) Triangular Oscillations.

Figure 3: Bifurcation Diagrams for Triangularly and Sinusoidally Oscillating Table Heights. Notice the different regimes of dynamics in both.

able to plot both of these to create our bifurcation diagrams.

The last thing done for this problem was an animation of the system. As stated in the introduction, we created two sets of animations for the problem: One to display the heights of the table and the ball as a function of time and a corresponding physical representation of the dynamics at play. This was simply done by creating an update function that would utilize our calculated values from the simulation ran previously. This update function can be used with matplotlib to generate a gif of the plots.

## 5 Challenges

One challenge we faced was deciding on the method to find the collision time and height of the ball and the table. The ball's equation of motion follows a polynomial, while the equation of the table for the sinusoidal case is a sin function. The solution to this system of equations cannot be solved analytically as it is transcendental. We attempted to use scipy's fsolve to find the roots of this equation, but abandoned this approach in favor for Euler's method.

Computation time was consistently an issue throughout, animations and plots took upwards of five minutes to generate each- which added up when adjusting parameters.

Additionally, the logic for the collisions were a little complicated to sort out, and some failed attempts shown in Figure 4 can be seen.

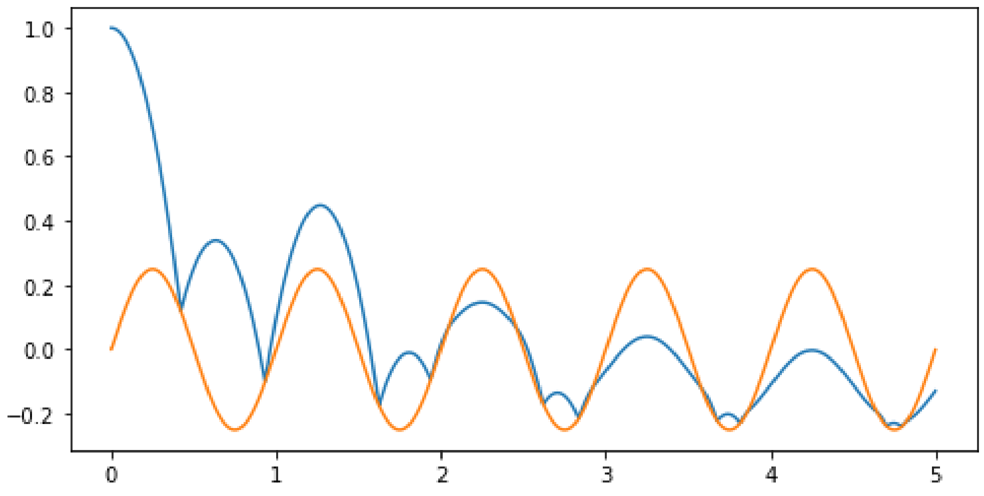


Figure 4: Failed Collision Logic

## 6 Future Directions

Given unlimited time and resources, we would like to increase the resolution and range of our bifurcation diagrams- as it stands right now, generating each of the animations and bifurcation plots took several minutes individually even with keeping code as well optimized as possible. We could also look into multi-threading the computation so that more computational power could be utilized. As mentioned previously, some transient structures appeared in the bifurcation diagram for the sinusoidal function. These might be resolvable if we had more computing power.

Another aspect to investigate further is the interpolation of the collision location if we continued to use Euler integration- in our code, one time step $dt$ before it hits the platform, we assume it bounces off of that location, which could introduce some error, maybe even explaining the transient structures. Instead, we can assume that the ball and platform moves linearly from $t$ to $t + dt$, and calculate the bounce as if it had happened at that point. Another approach for the simulation would have been to solve for the intersection of the two curves using some numerical solver like scipy's fsolve and take the intersection to be there.

We would also like to investigate the bifurcation of different parameters- like table oscillation amplitude and ball starting position. From reading related literature, there appears to be a method to reduce table amplitude and frequency into one parameter, so both could be investigated simultaneously.

## 7 Conclusions

We observe the dynamics of a ball bouncing on a sinusoidal and triangularly oscillating table. We were successful in creating a simulation to calculate the height of the ball throughout each time step of the simulation time. Additionally, we were able to successfully generate bifurcation diagrams for both systems and determine and explore the dynamics of different regimes for each system.

## 8 Contributions

Sage Li: Python coding for simulations, animations, and bifurcation and script writing.

Eathan Xu: Simulation Coding, Script Writing, Video Editing, Report Writeup

## 9 Code

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from scipy import signal
from scipy.fftpack import fft, fftshift

%matplotlib inline
def triangle(x, A, freq):
    wavelength = 1 / freq
    loc = x % wavelength
    if loc >= 0 and loc < wavelength / 4:
        return 4 * A * loc
    elif loc >= wavelength / 4 and loc < (3 * wavelength) / 4:
        return 2 * A - (4 * A * loc)
    elif loc > (3 * wavelength) / 4:
        return -4 * A + 4 * A * loc

def d_triangle(x, A, freq):
    wavelength = 1 / freq
    loc = x % wavelength
    return 4 * A if 0 <= loc < 0.25 or 0.75 <= loc else -4 * A
```

```
def triangle_range(x_lim, A, freq):
    triang_vals = []
    for x in x_lim:
        triang_vals.append(A * triangle(freq * x, 1, 1))
    return np.array(triang_vals)

def d_triangle_range(x_lim, A, freq):
    d_triang = []
    for x in x_lim:
        d_triang.append(A * d_triangle(freq * x, 1, 1))
    return np.array(d_triang)

x_lim = np.linspace(0, 10, 10000)
plt.plot(x_lim, triangle_range(x_lim, 2, 0.5))
plt.plot(x_lim, d_triangle_range(x_lim, 2, 0.5))

# %matplotlib qt
%matplotlib inline

t_max = 10
dt = 0.0025
g = 9.81
x_i = 1

e = 0.8 # coefficient of restitution

A = 0.25 # table oscillation amplitude
w = 1.75 # table oscillation frequency

t = 0
time_array = [0]
ball_pos = [x_i]
ball_vel = [0]
floor_pos = [0]
peaks = []

counter = 0 # counter for peak finding
while t < t_max:
    # instantaneous velocity of the floor
    floor_vel = A * 2 * w * np.pi * np.cos(w * t * 2 * np.pi)

    # checking for collision
    if ball_pos[-1] + ball_vel[-1] * dt < floor_pos[-1] + floor_vel * dt:
        # find the max height in the trajectory from the previous
        collision to current impact
        peaks.append(np.max(ball_pos[-counter:]))
        counter = 0
        if ball_vel[-1] > 0:
            ball_vel.append((ball_vel[-1] - g * dt + floor_vel) * e)
        elif ball_vel[-1] < 0:
            ball_vel.append((-ball_vel[-1] - g * dt + floor_vel) * e)
    else:
        ball_vel.append(ball_vel[-1] - g * dt)

    # updating the position of the floor
    floor_pos.append(A * np.sin(w * t * 2 * np.pi))

    ball_pos.append(ball_pos[-1] + ball_vel[-1] * dt)
    t += dt
    counter += 1
    time_array.append(t)

plt.figure(figsize=(15, 4))
plt.plot(time_array, ball_pos)
plt.plot(time_array, floor_pos)

# print(peaks)

plt.close()
fig, ax = plt.subplots(1, 2, figsize=(16, 4), gridspec_kw={'width_ratios': [15, 1]})
speed = 10
to_plot = np.arange(0, t_max / dt + speed, speed, dtype=int)

def update(n):
    ax[0].clear()
    ax[1].clear()
    i = n
    t = i * dt
    ax[0].plot(time_array, ball_pos, label='Ball␣trajectory', color='C0')
    ax[0].plot(time_array, floor_pos, label='Floor␣Height', color='C1')
    ax[0].plot([t, t], [-0.2, 1], color='r')
    ax[0].legend()
    ax[0].set_xlabel('Time', size=15)
    ax[0].set_ylabel('Height', size=15)

    ax[1].plot([0], [ball_pos[i]], marker='o', color='C0')
    ax[1].plot([-0.2, 0.2], [floor_pos[i], floor_pos[i]], color='C1')
    ax[1].set_ylim(ax[0].get_ylim())

    print(f'generating␣frame␣{i␣/␣10}␣/␣{t_max␣/␣dt}', end='\r')

ani = animation.FuncAnimation(fig, update, frames=to_plot, blit=False, interval=20)

ani.save('test.gif', writer='pillow')
```

```
# %matplotlib qt
%matplotlib inline

t_max = 10
dt = 0.0025
g = 9.81
x_i = 1

e = 0.8 # coefficient of restitution

A = 0.25 # table oscillation amplitude
w = 1.36 # table oscillation frequency

t = 0
time_array = [0]
ball_pos = [x_i]
ball_vel = [0]
floor_pos = [0]
peaks = []

counter = 0 # counter for peak finding
while t < t_max:
    # instantaneous velocity of the floor
    floor_vel = d_triangle_range([t], A, w)

    # checking for collision
    if ball_pos[-1] + ball_vel[-1] * dt < floor_pos[-1] + floor_vel * dt:
        # find the max height in the trajectory
        from the previous collision to current impact
        peaks.append(np.max(ball_pos[-counter:]))
        counter = 0
        if ball_vel[-1] > 0:
            ball_vel.append((ball_vel[-1] - g * dt + floor_vel) * e)
        elif ball_vel[-1] < 0:
            ball_vel.append((-ball_vel[-1] - g * dt + floor_vel) * e)
    else:
        ball_vel.append(ball_vel[-1] - g * dt)

    # updating the position of the floor
    floor_pos.append(triangle_range([t], A, w))

    ball_pos.append(ball_pos[-1] + ball_vel[-1] * dt)
    t += dt
    counter += 1
    time_array.append(t)

plt.figure(figsize=(15, 4))
plt.plot(time_array, ball_pos)
plt.plot(time_array, floor_pos)

# print(peaks)

plt.close()
fig, ax = plt.subplots(1, 2, figsize=(16, 4), gridspec_kw={'width_ratios': [15, 1]})
speed = 10
to_plot = np.arange(0, t_max / dt + speed, speed, dtype=int)

def update(n):
    ax[0].clear()
    ax[1].clear()
    i = n
    t = i * dt
    ax[0].plot(time_array, ball_pos, label='Ball␣trajectory', color='C0')
    ax[0].plot(time_array, floor_pos, label='Floor␣Height', color='C1')
    ylower, yupper = ax[0].get_ylim()
    ax[0].plot([t, t], [ylower, yupper], color='r')
    ax[0].legend()
    ax[0].set_xlabel('Time', size=15)
    ax[0].set_ylabel('Height', size=15)

    ax[1].plot([0], [ball_pos[i]], marker='o', color='C0')
    ax[1].plot([-0.2, 0.2], [floor_pos[i], floor_pos[i]], color='C1')
    ax[1].set_ylim(ax[0].get_ylim())

    print(f'generating␣frame␣{i␣/␣10}␣/␣{t_max␣/␣dt}', end='\r')

ani = animation.FuncAnimation(fig, update, frames=to_plot, blit=False, interval=20)

ani.save('test.gif', writer='pillow')

t_max = 100
dt = 0.01
g = 9.81
x_i = 1

e = 0.8 # coefficient of restitution

A = 0.25 # table oscillation amplitude

resolution = 0.001
omegas = np.arange(1, 2 + resolution, resolution)

all_peaks = []
for w in omegas:
    print(f'Now␣computing:␣omega␣=␣{w:.4f}', end='\r')
    t = 0
    time_array = [0]
    ball_pos = [x_i]
    ball_vel = [0]
    floor_pos = [0]
    peaks = []

    counter = 0 # counter for peak finding
    while t < t_max:
        # instantaneous velocity of the floor
        floor_vel = d_triangle_range([t], A, w)

        # checking for collision
        if ball_pos[-1] + ball_vel[-1] * dt < floor_pos[-1] + floor_vel * dt:

            if t > 75:
                # find the max height in the trajectory from the previous collision to current i
                peaks.append(np.max(ball_pos[-counter:]))
                counter = 0

            if ball_vel[-1] > 0:
                ball_vel.append((ball_vel[-1] - g * dt + floor_vel) * e)
            elif ball_vel[-1] < 0:
                ball_vel.append((-ball_vel[-1] - g * dt + floor_vel) * e)
        else:
            ball_vel.append(ball_vel[-1] - g * dt)

        # updating the position of the floor
        floor_pos.append(triangle_range([t], A, w))

        ball_pos.append(ball_pos[-1] + ball_vel[-1] * dt)
        t += dt
        counter += 1
        time_array.append(t)

    all_peaks.append(peaks)
print('Done')

%matplotlib inline
plt.close()
plt.figure(figsize=(15, 9))
for i, peaks in enumerate(all_peaks):
    plt.scatter(omegas[i] * np.ones(len(peaks) - 1),
    peaks[1:], marker='o', color='b', alpha=0.1, s = 1)

plt.xlim(1.1, 1.8)
plt.ylim(0, 1.5)
plt.xlabel('Table␣Frequency', size=20)
plt.ylabel('Peak␣Height', size=20)
plt.title('Table␣Frequency␣vs.␣Peak␣Height␣Bifrucation␣Diagram', size=30)
```