project1final

November 10, 2024

0.0.1 Introduction

The structure of molecules is often simplified by thinking about each atom as a point mass connected by springs to its neighbors, which is meant to represent the intermolecular bonds. The vibration of these molecules is then easy to describe in this picture as just being the point masses experiencing some kind of spring oscillation. Given that molecules are often large systems, solving dynamics of these sort of simplified systems is very tedious to do by hand, sense it involves considering the influence of the many different particles' oscillation on your point of interest, and carrying out this process for each point in the molecule. Given the simplicity of the conceptual model, however, the dynamics of this system is simple to solve for computationally, since it just involves a bunch of masses on springs.

0.0.2 Research Goal and Question:

The primary goal of this work is to first write code that will allow the simulation of any (n, m) two-dimensional lattice where some of its atoms experience some initial velocity. This will allow us to visualize how this initial velocity/energy introduced into the system spreads over the system. Because there is no damping of these oscillators, the total energy of the system will stay conserved (up to numerical error), and will spread across the lattice.

With a simulation like this in hand, we then wish to determine how changing the free variables of the system, the spring constant k between each bond (which determines the strength of their interaction) and the spring length L changes the dynamics of the system. We will answer this question by running our simulation for multiple different k and L values, taking the power spectrum of each "atom" in the systems, and inspecting how the peaks of the generated spectra change over varying k and L values. The peaks of a power spectrum will tell us about what frequency values each point in the system prefers to oscillate at, so this will tell us how the atomic oscillations change as we vary the interaction strength k and length L.

[ ]:
```
%matplotlib inline
from pylab import *
from scipy.integrate import odeint
from scipy.linalg import norm
from matplotlib import animation
from IPython.display import HTML # enable animation in Jupyter notebook
from scipy import *
from matplotlib.animation import FuncAnimation, FFMpegWriter
import numpy as np
```

I will first generate plots describing the dynamics for a single 16 by 16 lattice, where I'll choose $L = 1$ and $k = 5$.

[15]:
```
L = 1
k = 5

number_of_columns = 16
number_of_rows = 16

def f_loop(s, t):

    s = s.reshape((2, number_of_columns, number_of_rows, 2)) # reshaping the␣
↪state of my system so I can index it with [i,j]
    positions = s[0]
    velocities = s[1]

    forces = np.zeros((number_of_columns,number_of_rows,2)) # initializing␣
↪forces experienced by all particles to be zero

    # I am defining an exterior column and row of atoms that don't move (these␣
↪stay fixed so whole system doesn't move)
    for j in range(1,number_of_rows-1): # iterate over all rows EXCEPT for the␣
↪first and last rows (the fixed rows)
        for i in range(1, number_of_columns-1): # iterate over all columns␣
↪EXCEPT for the fixed left and right side
            my_pos = positions[i,j]
            # find the position of the point to your right
            rightneighbor_pos = positions[i+1, j]
            leftneighbor_pos = positions[i-1, j]
            topneighbor_pos = positions[i, j+1]
            bottomneighbor_pos = positions[i, j-1]

            # compute the displacement vector between your position and each of␣
↪your neighbors (top, bottom, left and right)
            rx1vec = rightneighbor_pos - my_pos
            rx1 = norm(rx1vec)
            rhatx1 = rx1vec/rx1
            rx2vec = leftneighbor_pos - my_pos
            rx2 = norm(rx2vec)
            rhatx2 = rx2vec/rx2
            ry1vec = topneighbor_pos - my_pos
            ry1 = norm(ry1vec)
            rhaty1 = ry1vec/ry1
            ry2vec = bottomneighbor_pos - my_pos
            ry2 = norm(ry2vec)
            rhaty2 = ry2vec/ry2
```

```
            # compute the force experienced by your particle due to the␣
↪relative position of each neighbor and the resulting spring force
            F1 = k*(rx1 - L)*rhatx1
            F2 = k*(rx2 - L)*rhatx2
            F3 = k*(ry1 - L)*rhaty1
            F4 = k*(ry2 - L)*rhaty2
            # the total force experienced by your point at a given time will be␣
↪a sum of all 4 forces generated by your neighbors
            Ftot = F1+F2+F3+F4

            # update the force on your particle as the sum of all contributions␣
↪above
            forces[i,j] = Ftot

    # because odeint works with arrays, we need to reshape the state of our␣
↪system from matrices back into an array
    return concatenate((velocities.reshape(-1),forces.reshape(-1)))
```

[16]:
```
initial_lattice_points = np.array([[[i+1, j] for j in range(number_of_rows)]␣
↪for i in range(number_of_columns)])

# Generate initial lattice velocities with shape (number_of_columns,␣
↪number_of_rows, 2)

# set all of them to be zero (system is in equilibrium)
initial_lattice_velocities = np.zeros((number_of_columns, number_of_rows, 2))
# Then assign some initial velocity to a specific point, e.g., point (3,3)
#initial_lattice_velocities[2, 1] = [3, 0]
#initial_lattice_velocities[2, 2] = [0,-1]
initial_lattice_velocities[3, 3] = [1,-1]

# Combine all initial lattice points and velocities into a single intial state,␣
↪which is currently a tensor:
s0 = np.stack((initial_lattice_points, initial_lattice_velocities), axis=0)
```

[17]:
```
num_timesteps = 50
t_len = 40
ts = linspace(0,t_len,num_timesteps)

# Read in the initial state of the system into f_loop, where I'm converting my␣
↪intial state tensor into a vector so that it can be properly used in odeint
# ... then feed the initial state of the full lattice into odeint to solve for␣
↪the dynamics of the full system over time (over t_len)
ans = odeint(f_loop,s0.reshape(-1),ts)
```

[ ]:

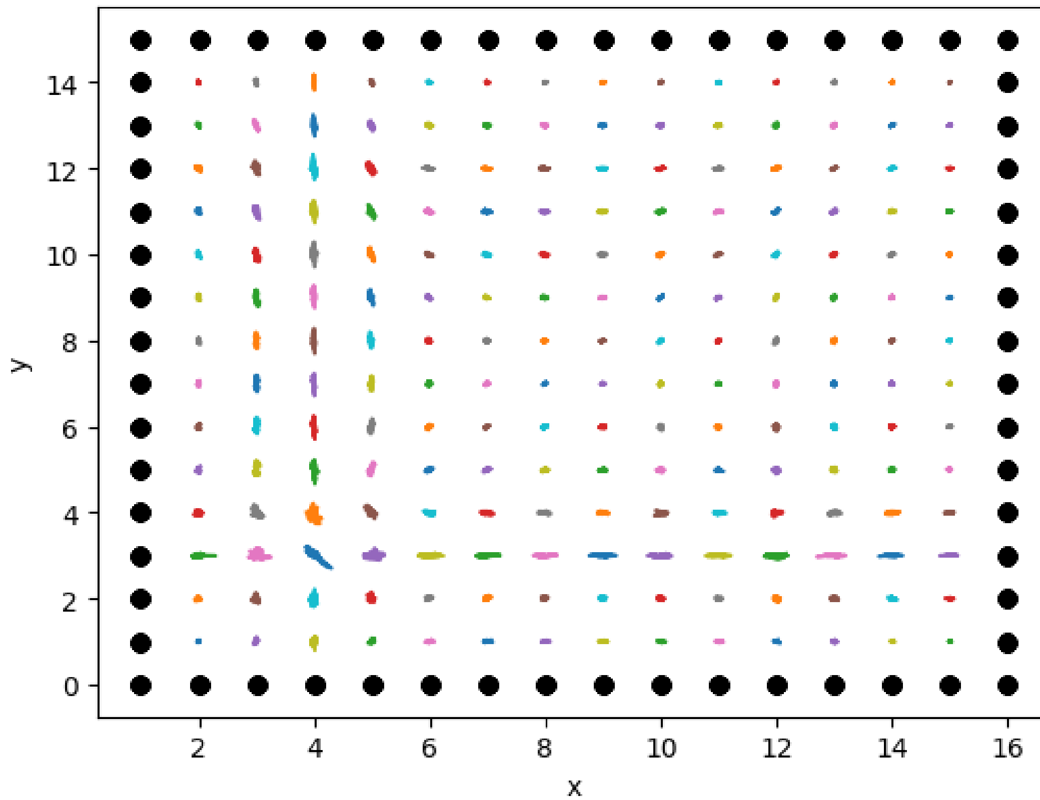
```

```

[18]:

```
val = 2*number_of_columns * number_of_rows
positions = ans[:, :val].reshape(-1, number_of_columns, number_of_rows, 2)  #␣
↪Reshape to (num_timesteps, number_of_columns, number_of_rows, 2 contains␣
↪both the x AND y coordinates)
```

[19]:

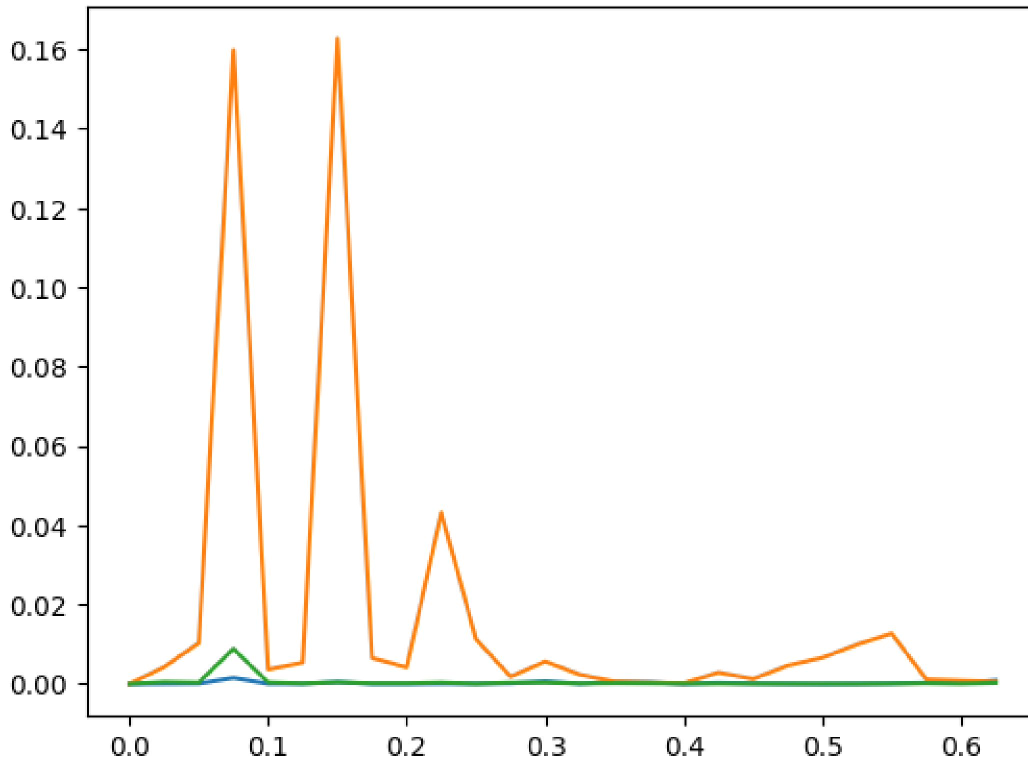
```
# now plot the full dynamics of the system over x position and y position

for i in range(number_of_rows):
    for j in range(number_of_columns):
        # Check if the point has moved
        if np.any(np.diff(positions[:, i, j, :], axis=0)):  # Check for any␣
↪movement. If none, plot the state of the system over time as a path over x␣
↪and y
            plt.plot(positions[:, i, j, 0], positions[:, i, j, 1],␣
↪label=f'Point ({i}, {j}) Line')
        else:
            # If there's no movement, plot as scatter point to represent the␣
↪fixed columns and rows that keep our lattice in one place
            plt.scatter(positions[:, i, j, 0], positions[:, i, j, 1],␣
↪label=f'Point ({i}, {j}) Scatter', color = "k")

plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



[20]:
```
# I'm first going to pick the x and y positions over time for specific points
↪and generate power spectra for each of them

point11 = positions[:,1,1]
test = point11[:,1]
f, Pden = signal.periodogram(test, num_timesteps/t_len)

point33 = positions[:,3,3]
test2 = point33[:,1]
f2, Pden2 = signal.periodogram(test2, num_timesteps/t_len)

point77 = positions[:,6,6]
test3 = point77[:,1]
f3, Pden3 = signal.periodogram(test3, num_timesteps/t_len)
```

[21]:
```
plt.plot(f, Pden)
plt.plot(f2, Pden2)
plt.plot(f3, Pden3)
#plt.yscale("log")
#plt.xlim(0,10)
```

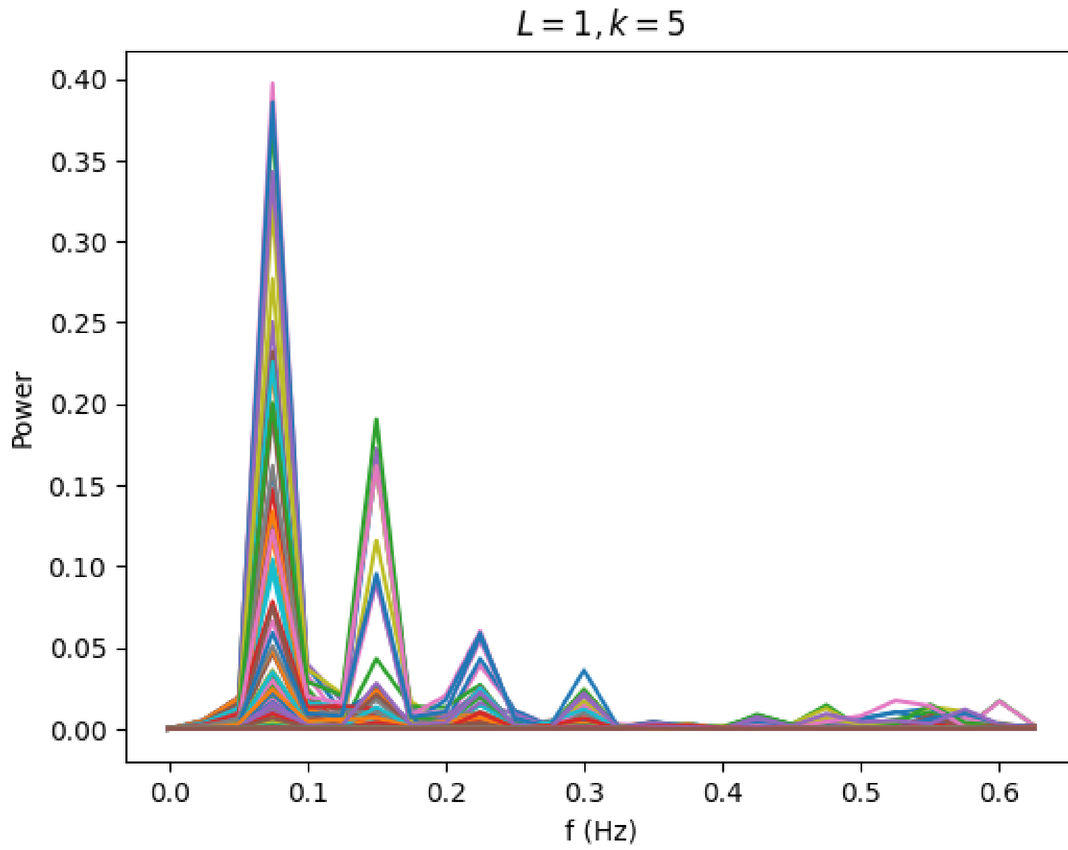
[21]: [<matplotlib.lines.Line2D at 0x1230e3110>]



It looks like there is a set of preferred frequencies that the these particles like to oscillate at. To analyze this further, if we plot the spectra generated by ALL points, we may notice a pattern.

[22]:
```
for j in range(1,number_of_rows-1):
    for i in range(1, number_of_columns-1):
        point = positions[:,i,j]
        test = point[:,1]
        f, Pden = signal.periodogram(test, num_timesteps/t_len)
        plt.plot(f, Pden)
plt.xlabel("f (Hz)")
plt.ylabel("Power")
plt.title("$L=1, k=5$")
```

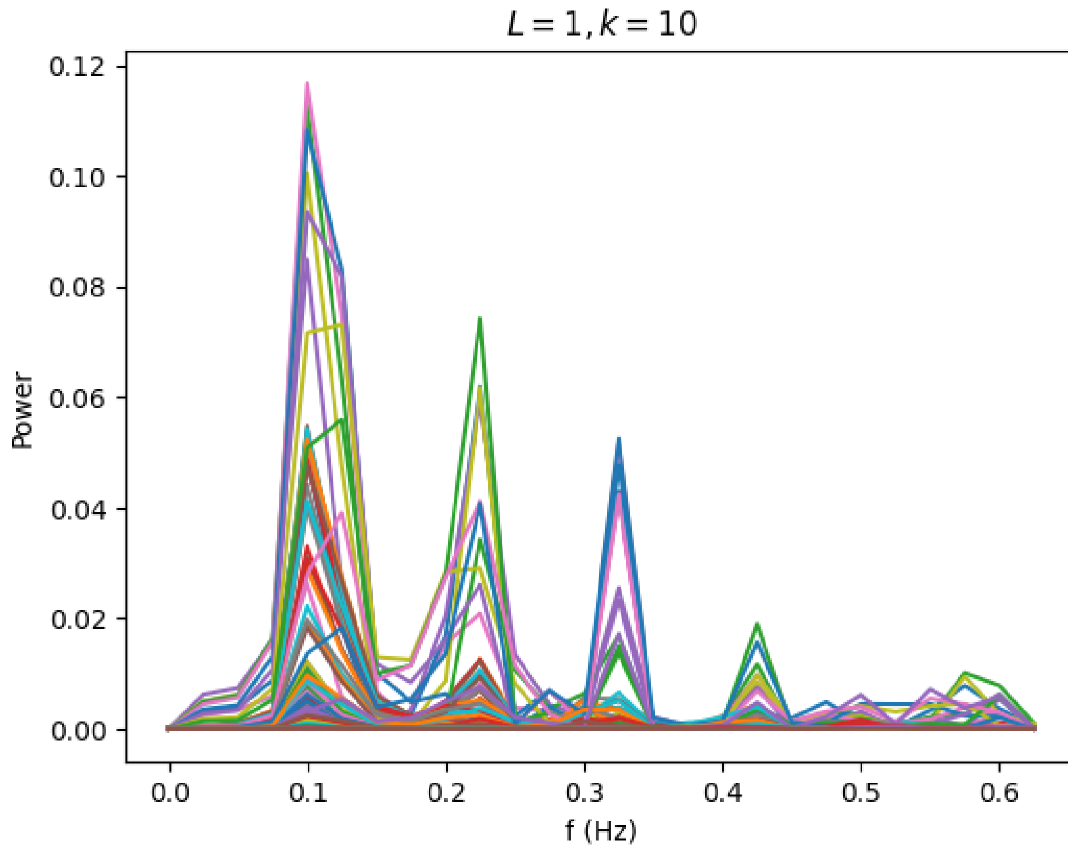
[22]: Text(0.5, 1.0, '$L=1, k=5$')



Now I rerun the above code for a system with a k value of 10 and the same L value to see how this compares to the previous plot.

[24]:
```
for j in range(1,number_of_rows-1):
    for i in range(1, number_of_columns-1):
        point = positions[:,i,j]
        test = point[:,1]
        f, Pden = signal.periodogram(test, num_timesteps/t_len)
        plt.plot(f, Pden)
plt.xlabel("f (Hz)")
plt.ylabel("Power")
plt.title(f"$L={L}, k={k}$")
```

[24]:
```
Text(0.5, 1.0, '$L=1, k=10$')
```



The preferred frequencies of the system have shifted to higher values when we double the k value. Recall that $\omega^2 = \frac{k}{m} = 4\pi^2 f^2$ so $k \propto f^2$, which is the behavior we see when comparing these two plots.

To get a better idea of how the system responds to changing k and L, we finally run the above code and generate the resulting power spectra of each particle for a variety of k and L values:

[ ]:

```
```

[ ]:

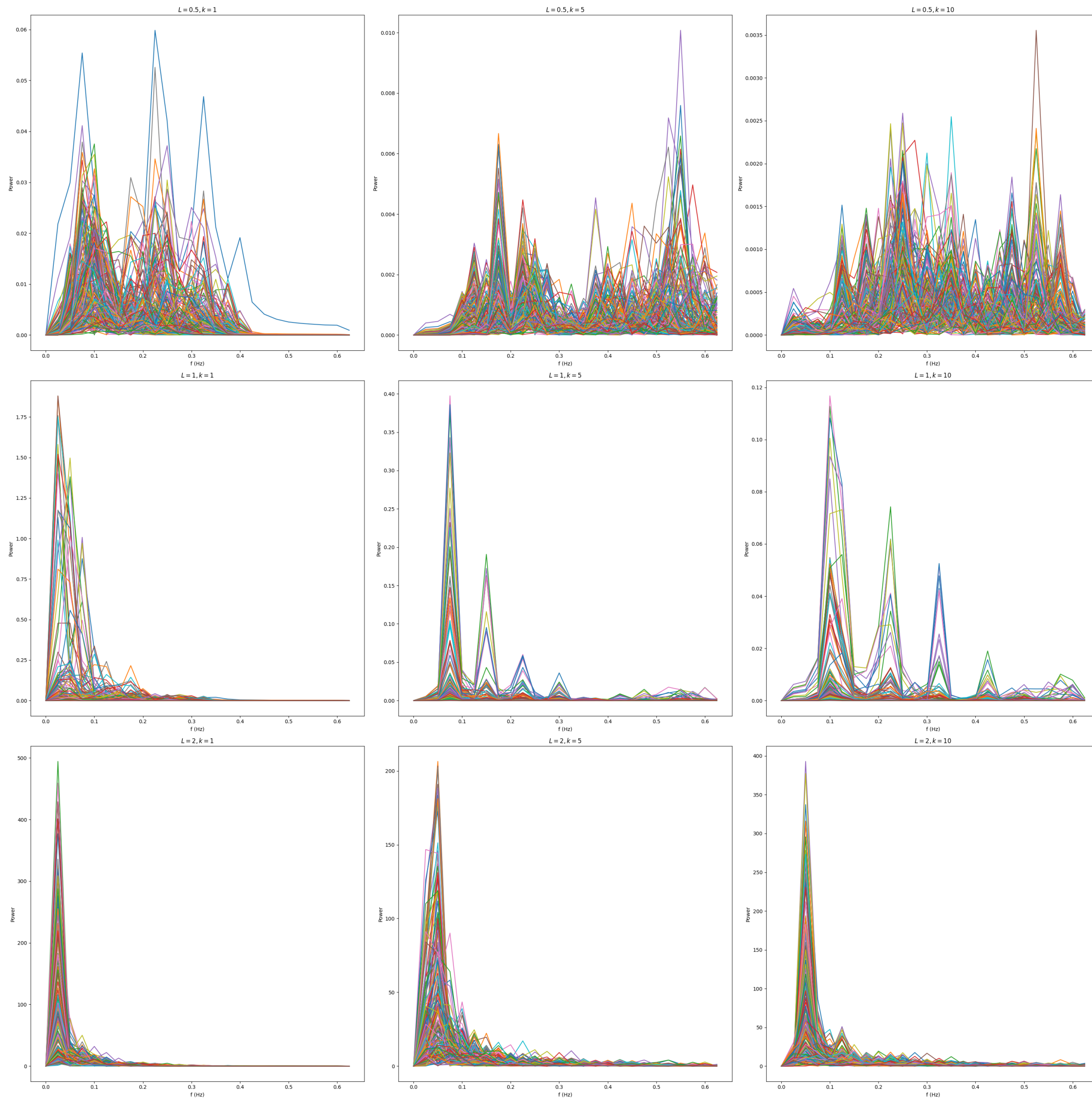
```
```

[14]:

```
# defining the set of possible k and L values that we want to generate spectra
↪for:
L_array = [.5, 1, 2]
k_array = [1, 5, 10]

number_of_columns = 16
number_of_rows = 16

num_timesteps = 50
```

```
t_len = 40

val = 2*number_of_columns * number_of_rows

fig, axes = plt.subplots(len(L_array), len(k_array), figsize=(30,30))

for ix, x in enumerate(L_array): # for every L value in the array... also keep␣
↪its index in the array for plotting purposes
    for iy, y in enumerate(k_array):

        # ------ This code is exactly what we've done before but is now being␣
↪looped over --------
        initial_lattice_points = np.array([[[i+1, j] for j in␣
↪range(number_of_rows)] for i in range(number_of_columns)])

        initial_lattice_velocities = np.zeros((number_of_columns,␣
↪number_of_rows, 2))
        initial_lattice_velocities[3, 3] = [1,-1]
        s0 = np.stack((initial_lattice_points, initial_lattice_velocities),␣
↪axis=0)

        ts = linspace(0,t_len,num_timesteps)
        L = x
        k = y
        ans = odeint(f_loop,s0.reshape(-1),ts)

        positions = ans[:, :val].reshape(-1, number_of_columns, number_of_rows,␣
↪2)  # Reshape to (num_timesteps, number_of_columns, number_of_rows, 2 is the␣
↪x AND y coordinates)

        # again, just as before, we generate the spectra for every point in the␣
↪lattice
        for j in range(1,number_of_rows-1):
            for i in range(1, number_of_columns-1):
                point = positions[:,i,j]
                test = point[:,1]
                f, Pden = signal.periodogram(test, num_timesteps/t_len)

         # ----- end of direct copying and pasting of previous code␣
↪-------------
                axes[ix,iy].plot(f, Pden) # For every point's spectra at a␣
↪given k and L, add the resulting plots to a specific location on the plot␣
↪grid we defined previously

        axes[ix,iy].set_xlabel("f (Hz)")
        axes[ix,iy].set_ylabel("Power")
```

```
        axes[ix,iy].set_title(f"$L={x}, k={y}$")


plt.tight_layout()
plt.show()
```



0.1 Conclusion:

As is expected from the relationship $k \propto f^2$, we see for every L value that as we increase k, the frequency values that the lattice prefers to vibrate at (determined by the peaks in the power spectra) shifts to higher values.

Also, as L increases, we see that the frequency values that correspond to peaks in power decrease. The spread in the spectra also seems to decrease as we increase L. This makes physical sense

considering the fact that when we change L, we are not actually changing the spacing between points; this only changes the effective spring constant, determined when we calculate spring forces by `F = k*(r - L)*rhat`. Because of the way we define the direction of these spring forces and due to the sign L is given in these expressions, an increase in L results in an effective decrease in k, which we have already established results in lower preferred frequency values.

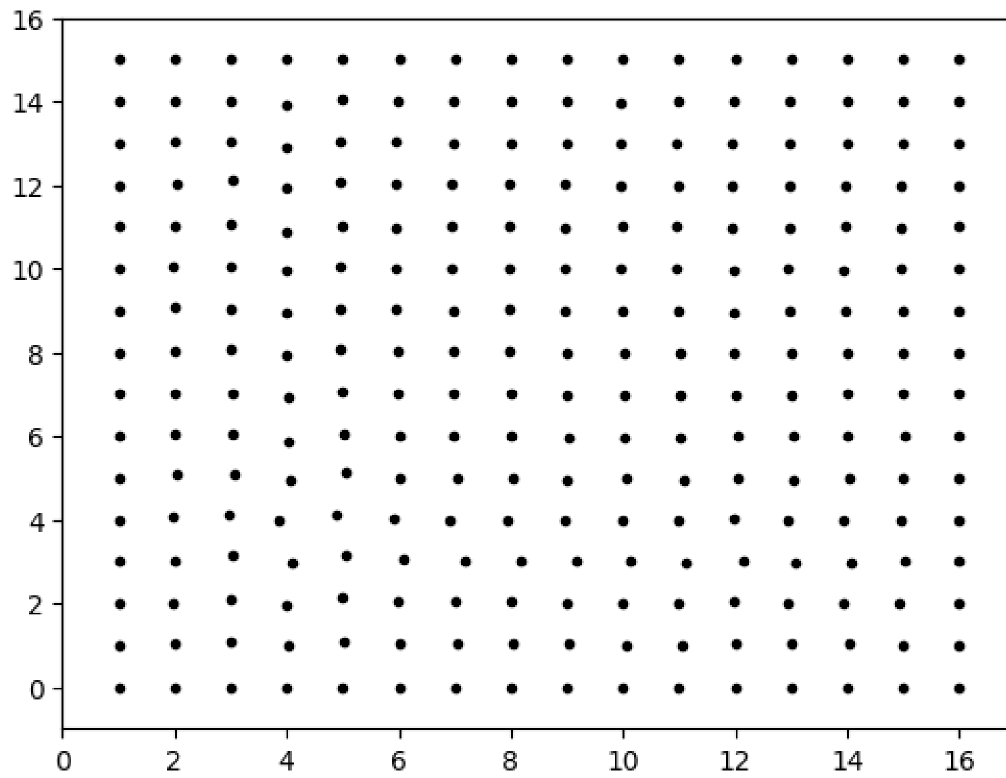
0.1.1 A little fun:

Just for fun, we can generate an animation of the lattice point's positions over time.

[23]:
```
# initializing all of the plots we'll fill with data in our loop over multiple␣
↪k and L values
fig, ax = plt.subplots()
scatters = [ax.plot([], [],'.', color="k")[0] for _ in range(number_of_columns␣
↪* number_of_rows)]

def init():
    ax.set_xlim(0, number_of_columns+1)
    ax.set_ylim(-1, number_of_rows)
    return scatters

def update(frame):
    for i in range(number_of_rows):
        for j in range(number_of_columns):
            # Take the data for point [i,j] at the given time (frame), and take␣
↪the point's x AND y values so that it can be plotted in 2D
            pos = positions[frame, i, j, :]
            # Access the correct scatter object for each (i, j) point
            scatter = scatters[i * number_of_columns + j]
            scatter.set_data(pos[0], pos[1])
    return scatters

ani = FuncAnimation(fig, update, frames=num_timesteps, init_func=init,␣
↪blit=False)

#ani.save("16by16.gif", writer='pillow')

HTML(ani.to_jshtml())
```

```
/var/folders/jp/q2d924_j50z44bjzywq203800000gn/T/ipykernel_23770/1462732790.py:1
6: MatplotlibDeprecationWarning: Setting data with a non sequence type is
deprecated since 3.7 and will be remove two minor releases later
  scatter.set_data(pos[0], pos[1])
```

[23]:
```
<IPython.core.display.HTML object>
```



16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16


```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```