Summary of Research at Duke General Robotics Lab (Fall 2022 to Summer 2023)

Fall 2022

This semester I have been working with the Duke General Robotics Lab, und-er the supervision and guidance of Dr. Boyuan Chen, in the general field of AI and sound research. What inspired me to pursue this research was a paper I read by Dr. Chen this summer, titled “The Boombox: Visual Reconstruction from Acoustic Vibrations”. In it he used contact microphone data to deconstruct the location and identity of an object collision inside of a plastic box. I was hoping to extend this work to objects of other geometries - in this case I focused on a table - and to train the artificial intelligence on a wave simulation to help it learn the “behavior” of propagation and interference before training on real data. My hypothesis is that by incorporating an accurate enough wave simulation, the AI can A) learn a more general and intelligent “understanding” of wave behavior and B) expand its learning to other shapes. The work I have done for the Fall 2022 school semester is thus summarized below.

Step 1 – Prior Research and Existing Solutions

Before I wrote a single line of code, I knew I needed to catch up on the current state of research in this area. The paper I spent the most time looking into was definitely the one written by Dr Chen - the one that started my curiosity about this project. The paper talked about the specific technicalities of the project, but I did not learn much about the actual math/physics of wave propagation. Hence, I spent some time learning about the wave equation, harmonic modes, superposition, and frequency analysis from different resources on the internet.

Aside from this research, I also committed a couple of weeks to looking into the PyImpact and JWave libraries in python. PyImpact handles collision sounds, some scraping sounds, and general physics. Although this was a great resource, I could not A) get into the code to understand how the mathematics/model calculate sound over time or B) collect pressure/sound data from individual points on a 3D object. This led my focus to creating my own wave propagation simulation. At first, I only saw this to better understand the problem, but it quickly became most of my work this semester.

Step 2 – 2D Wave Simulation

During my research phase I thought a lot about why waves propagate the way they do. I made the insight that if you break it down to a microscopic level, objects are essentially very large connected systems of damped harmonic oscillators. If one atom, molecule, or infinitesimal volume element (referred to now as particles) is disturbed, it applies a pushing force to one neighbor and a pulling force to another. This means the overall behavior of the system to a

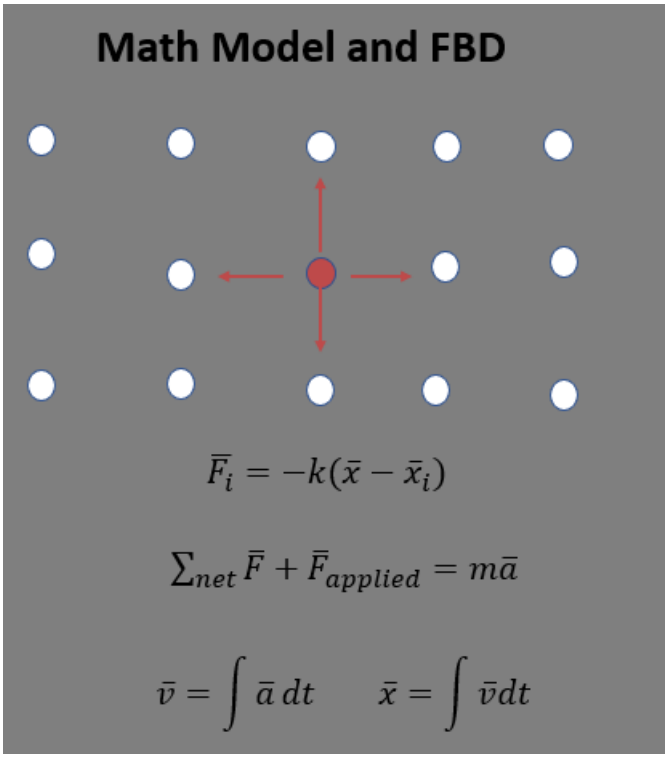
disturbance (that is the sound after a collision) can be modelled by coupling many linear systems. There are numerous simplifying assumptions with this model, including but not limited to…

A) Assuming a homogenous and repeating lattice structure (e.g. rectangular or hexagonal lattice)

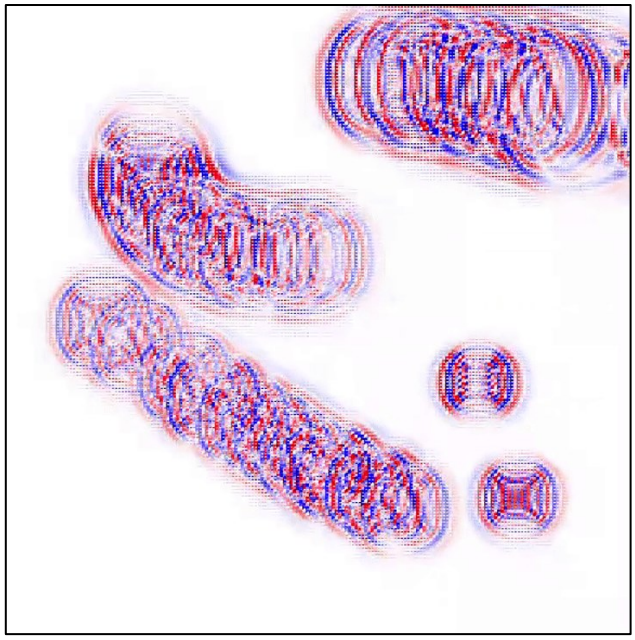
B) Assuming a linear spring-mass model for every particle (double the displacement, the force is also doubled)

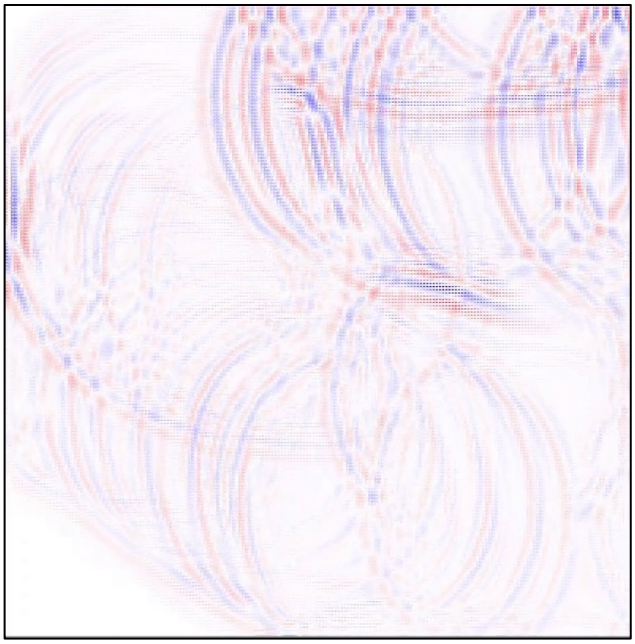
C) Assuming each particle only influences the particles around it (aka locality, experimented with changing the number of neighbors)

D) Assuming there is a degree of damping to each system (added only after position error grew out of hand over time)



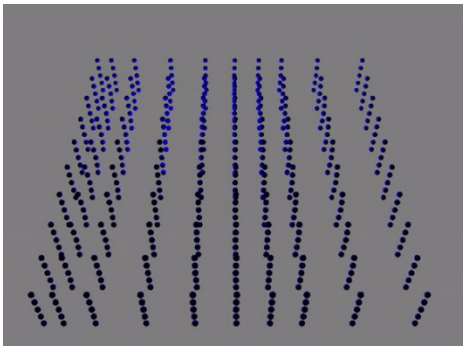
In order to quickly experiment with this model and explore its feasibility, I chose to create a simple 2D simulation in a language I was comfortable with. I chose to work in Java and used the Processing library to handle the graphics. For visualization, I mapped the compression and tension in a link to a color ranging from red to blue respectively. For the boundary cases I created the option to normally fix, tangentially fix, fully fix, not-fix, or drive any of the edges. Snapshots of results are shown below (a link to a YouTube video can be found in the resources section of this document).

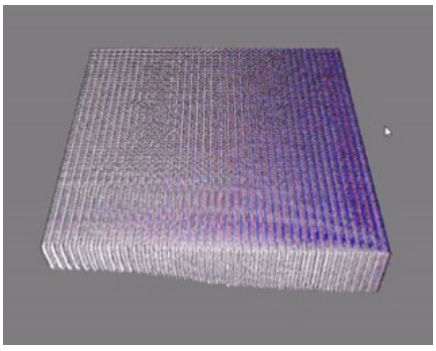


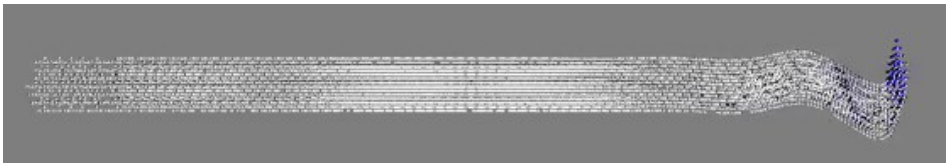


Step 3 – 3D Wave Simulation

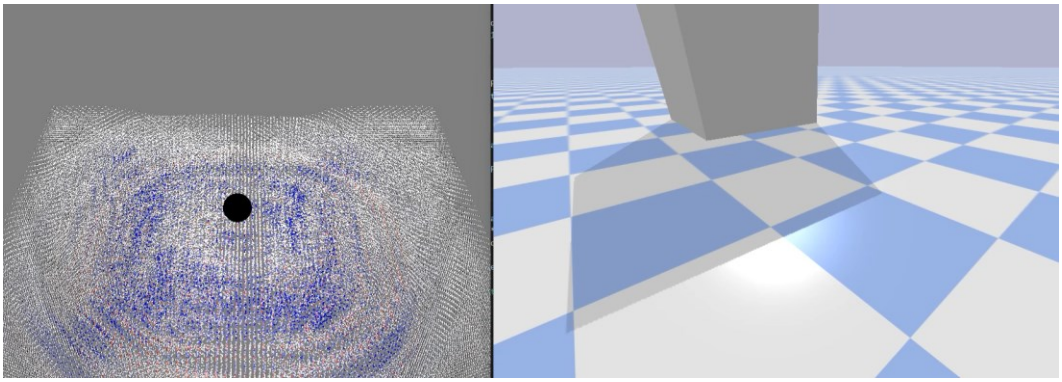
The results of the 2D simulations looked promising, so I chose to explore the idea further and expand. The next goal became to implement the same mathematical model in 3D and on the GPU (previous simulation was running on CPU only). Thanks to Dr. Chen I was introduced to the Taichi Physics Library in Python, which I have used extensively since. Taichi can handle very large parallel calculations on the GPU using tensor fields and kernels, and importantly it is not too difficult to learn. After several weeks of testing and debugging, I had a real-time wave environment with customizable boundary conditions (fully fixed, tangentially fixed, normally fixed, driven) and stable long-term standing waves. Snapshots of the work are shown below, and as before both the videos and codebase is available in the resources section of this document.

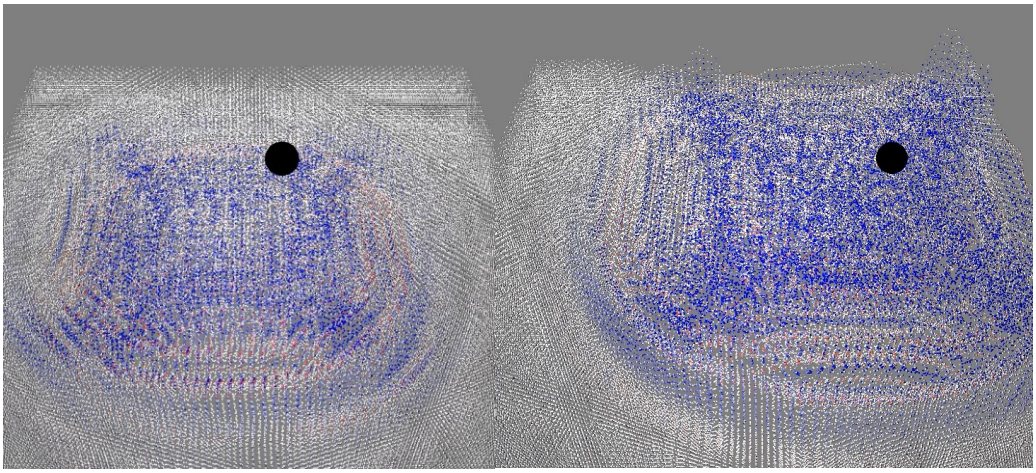




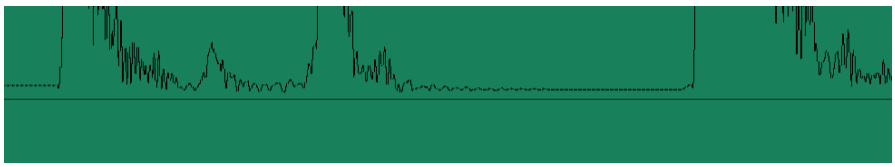


To allow interactions with a separate object however, there must be a way to interface a physics engine with this simulation to allow information such as contact point locations and normal/shear forces to pass into the simulation and make changes to the simulation. The physics library of choice for this is PyBullet (another excellent recommendation of Dr. Chen's). PyBullet only runs on CPU, but can efficiently calculate 3D object trajectories, collision zones and contact points, and all the necessary reaction and impulse forces. For now, a simple cube is picked and the results of this are shown below.



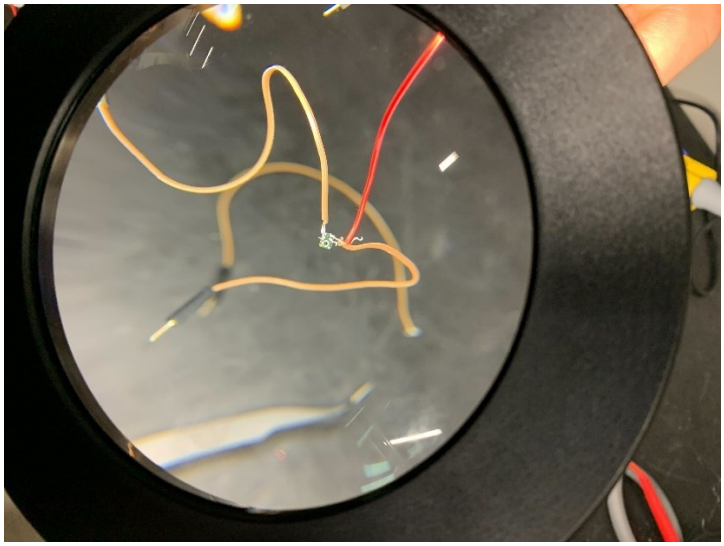


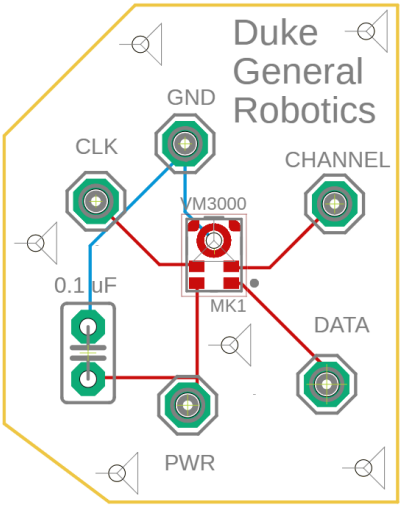
The final addition to this simulation environment for this semester was the addition of generated sound. For this, a point or group of points are selected. The average compression/tension of every connection to every neighbor of this point is calculated. This number is normalized, amplified, and written to a .WAV file over time. Because the finished simulation runs at around 2-10 FPS and sound is usually sampled at 44kHz, interpolation between data is necessary to get a clean sound. This sound and video can then be played simultaneously to qualitatively examine the simulation. Indeed, the sounds produced are reasonable and realistic.



Step 4 – VM3000 Microphone

While working on the FEA Simulation, I simultaneously investigated reading data from the VM3000 digital microphone for my own project and for the lab in general. After purchasing ten and spending hours trying, I realized that it was too small to solder with conventional techniques. There are 4-5 necessary connections all within a square millimeter or two of space. For this reason, I designed a few iterations of a simple PCB that the VM3000 can be attached to. I made one version for a single audio channel, and one version for two connected audio channels. These PCB's will most likely never be used, as it would make the microphone too large for it to have great spatial resolution.





Spring 2023

This semester I have been working with the Duke General Robotics Lab, under the supervision and guidance of Dr. Boyuan Chen, in the field of AI and sound research. What inspired me to pursue this research was a paper I read by Dr. Chen this summer, titled "The Boombox: Visual Reconstruction from Acoustic Vibrations". In it he used contact microphone data to deconstruct the location and identity of an object collision inside of a plastic box. I was hoping to extend this work to objects of other geometries - in this case I focused on a table - and to train the artificial intelligence on a wave simulation to help it learn the "behavior" of propagation and interference before training on real data. My hypothesis is that by incorporating an accurate enough wave simulation, the AI can A) learn a more general and intelligent "understanding" of wave behavior and B) expand its learning to other shapes. The work I have done this semester is a continuation of research from the Fall of 2022. In the research summary for that semester, I

review the 2D CPU simulation, the 3D GPU simulation, and the sound wave file generation. Below is a review of the continuation of this.

Step 5 – Redesign 3D GPU Simulation – logic flow structure

The Taichi-based multi-threaded physics simulation at the end of last semester had several major flaws:

- First was that the PyBullet collision simulation and the Taichi propagation simulation ran concurrently and codependently. This meant the speed of the overall process was limited to the speed of the slowest link in the chain. After incorporating live sound processing and live frame-to-video, the entire program ran from 0.5 FPS to 4 FPS. This is too slow.
- Second was that the implementation was incredibly limited. The object being tested could only be a flat table of any dimension. The structure of the FEA point elements was constrained to an even cartesian lattice. The number of neighbors was built into the math as six, one for every neighbor.

To address the first (and most pressing) limitation, I restarted the entire simulation project from scratch. Because of this, I was able to address the second set of limitations simultaneously - more detail on that in the next segment.

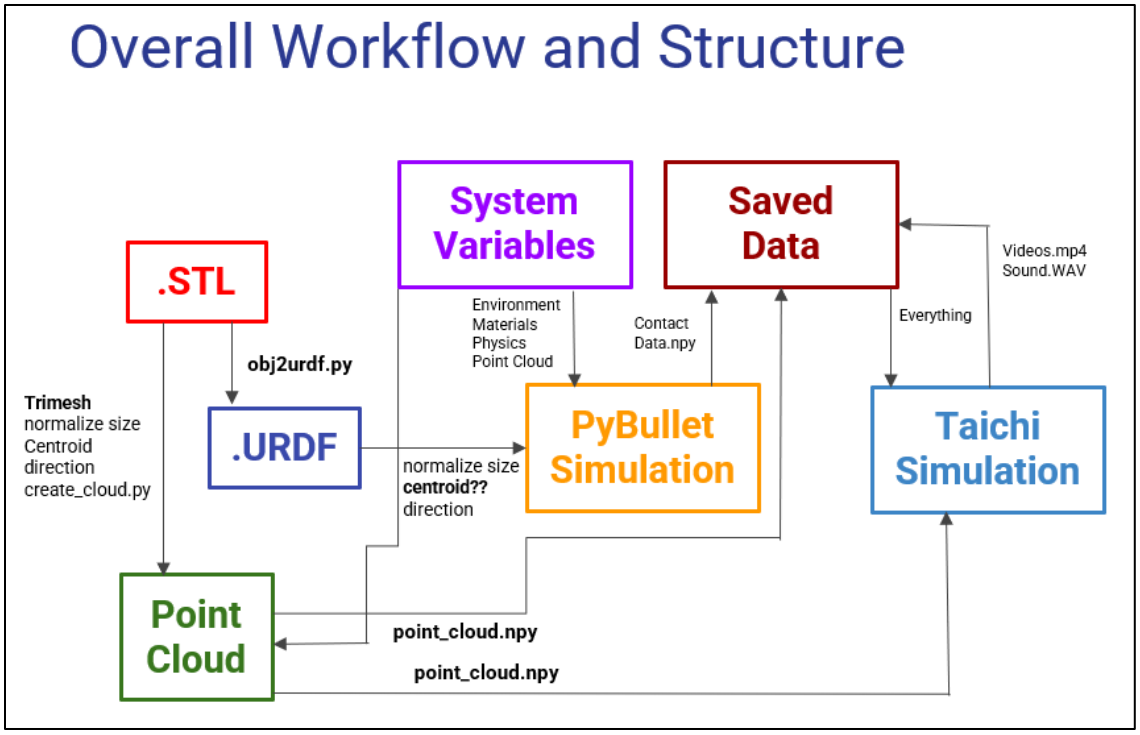
The core principle of speeding up the pipeline is separating as many tasks as possible. Instead of one colossal main file, each subprocess is separated into its own CPU or GPU only file. These subprocess, as of the most up-to-date model and in no set order, are as follows:

A) Convert complex and concave STL/OBJ files into simplified and convex-chunked vhcad.urdf (CPU only, done one time per object)

B) Simulate the collision of this with another object in PyBullet. Use numpy to arrange and save this data (CPU only, done thousands of times per object)

C) Create a point cloud and neighbor distribution from the STL/OBJ file given. Use numpy to arrange and save this data. Also a helper file used to view and interpret these point clouds (GPU only, done couple of times per object)

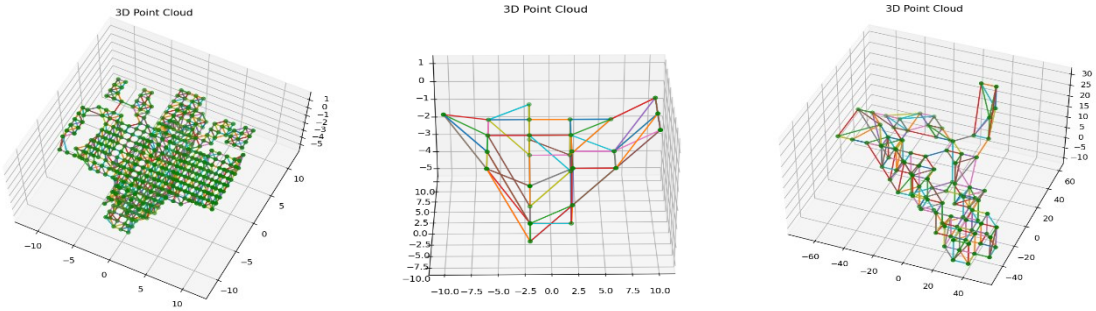
D) Use a given point cloud and collision data to simulate the wave propagation (GPU only, done thousands of times per object)



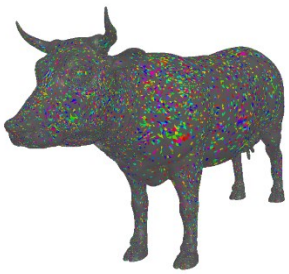
Between these subprocesses, data is saved in the .npy format. This was chosen because of its superior saving-time, loading-time, and file size compared to other data storage methods. All parameters for the subprocesses are defined in a .YAML file. These include physics parameters for both simulation and collision detection, point cloud creation parameters, and object orientation in space.

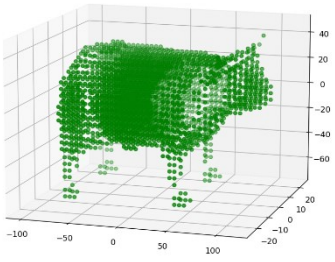
Step 6 – Redesign 3D GPU Simulation – any shape and distribution

To improve the adaptability of this codebase to other projects, I wanted to ensure that any 3D object could be used – both for the object striking and object being struck. Furthermore, I wanted to create everything such that any lattice imaginable could be used. Examples include cartesian lattice, random distribution lattice, face-centered cubic lattice, body-centered cubic lattice, hexagonal packing, etc. Only one line in the main simulation would be modified to switch between these point clouds. Only one line in the main simulation would be modified to switch between 3D objects.



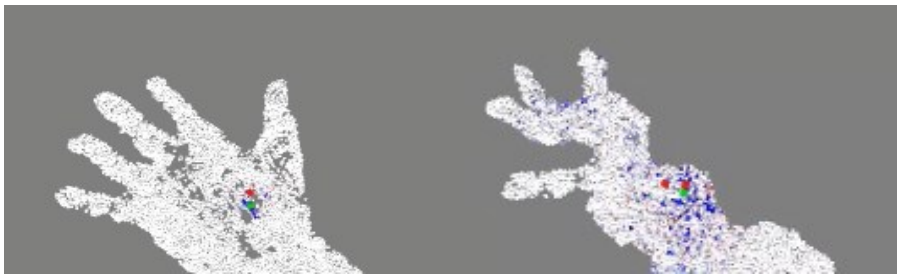
To ensure consistency in the point cloud creation and collision detection, all objects are uniformly scaled and centered to be contained within a unit sphere using the Trimesh 3D library. Trimesh has a function that finds the Euclidean distance of a point to the closest point on the mesh, negative if the object is within the object. Using this, points are assigned only inside. A distance matrix is created, saving the distance from every point to every other (done is SciPy). This matrix is sorted, and the first column is removed (distance from A to A will always be zero). The first “num_neighbors” defined columns are kept. If every point was forced to have exactly “num_neighbor” connections, some edge cases would have abnormally long bonds. We instead find the average bond length and remove all bonds that fall out of range.





Step 7 – Redesign 3D GPU Simulation – major bug

After implementing everything above, one issue remained. At some point in time, every single simulation would become unstable. Either it would blow up, morph into a web, morph into a disfigured version of the original, or create bounded but chaotic orbital behavior. I first looked for a simple (or even fundamental) bug in my vector math or physics. After proving everything was correct, I experimented with different simulation parameters. I varied point cloud density and distribution, the strength of the springs and the velocity decay, and the number of neighbors. I tried constricting the magnitude of acceleration. None of this fixed the problem.



Reaching the conclusion that something may be fundamentally wrong with my model, I created several minimalist setups with just a handful of connected points. It became clear that everything was working as intended, but that simply controlling the distance between neighboring points did not constrain the system enough. A group of points can remain equally spaced but occupy many different configurations. To control this, I added a spring effect for both

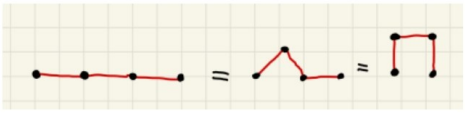
orientation and distance. This way, even if you are the correct spring-distance from your neighbor but wrong angle, you feel a correcting force. This solved the bug. The simulation now maintains stability for a wide range of system parameters. Some small bugs regarding the mapping of collision points from PyBullet to Taichi were also found and solved in this process.

Summer 2023 (as a condensed set of slides, see next page)

Unstable Simulation – Potential Causes

1) Error in position float is compounded. Because of the nature of spring systems, this may grow unbounded (BIBO unstable). At high damping, the instability is gone (not a solution) → I think this is not the major problem here because even if all the velocities are set to a single value problem persists

2) The current model only "controls" for distance between two points, not orientation between two points. See diagram on next page to see how this may cause problems. I did not experience this issue in last simulation because every single force was purely cartesian (here the points are rand dist)



1

Simulation Updates

- How should a user "select" the **location and number of pressure gauges?** Should the gauge be **fixed in space or fixed to object?**
- Should the pressure be the average pressure of all the points in the region? Should it be the pressure of the closest point to the center of the region
- How should the user select regions that are fixed and regions that are driven?
- Real life limitations → a separate loop is computationally expensive BUT if we check before "while running" we do not know the avg_bond_length

2

Simulation Updates

- **Pre_Processing** Sound → **Post_Processing** Sound
- Math necessary for **interpolation** and **scaling** → simulation sampling rate to audio file sampling rate
- Show file structure and process of going from .npy to array to new array to .wav

```
simulation_sampling_rate = 1000.0
audio_num_final_samples = 132300.0
samples per datapoint = 66

max value from zero = 5.5143905
maximum value of audio sample = 2147483648.0
scaling_factor = 389432641.7813887
```

3

Small Details and Updates

- ❑ Ability to toggle sound and graphics → show_GUI + record_Video + save_Sound
  - Without GUI it can run at 6.287 seconds per 1000 frame test at 50x50x50 points
- ❑ Ability to run multiple tests with varying parameters → K,M,D,dt
  - Still need a programmatic way to detect instability and failure
- ❑ Created hundreds of point cloud distributions for the rod with different parameters
  - Number of points
  - Number of neighbors
  - Bond Cutoff
  - Geometric Layout
- ❑ Add feature to track "critical" frames
  - For example, save frame when point X moves

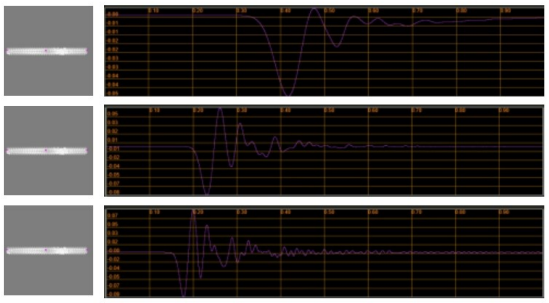
4

Speed of Sound Tests – Measure

Time Delay = (Frame_Move - Frame_Impulse) * (Length_Sound_File) / (Max_Frames)

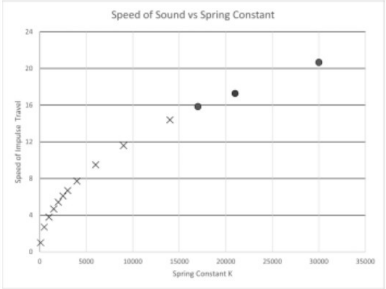
5

Speed of Sound Tests – Test I



6

Speed of Sound Tests – Test I



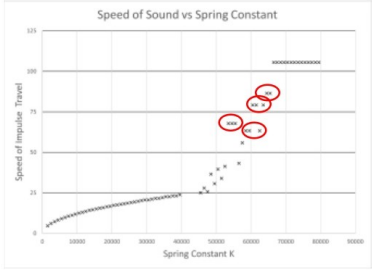
Tests above K=15000 were unstable

MODEL FITS

$-9*10^{-9}\ x^2 + 9*10^{-9}\ x + 3.3702$
($R^2 = 0.9831$)
OR
$4.2354 \ln(x) - 24.785$
($R^2 = 0.8646$)
OR
$0.1061 * x^{0.5139}$
($R^2 = 0.9992$)

7

Speed of Sound Tests – Test I



Tests above K=15000 were unstable

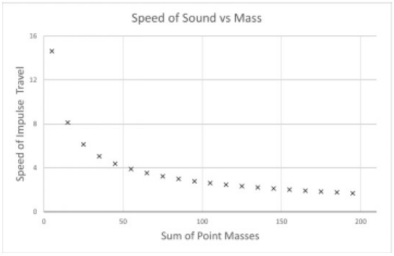
From 0 to 40000 there is a clear pattern (even if unstable)

From 40000 to 67000 it is confusing but there is a correlation

Limit in speed is 105.5 [Length/sec]

8

Speed of Sound Tests – Test II

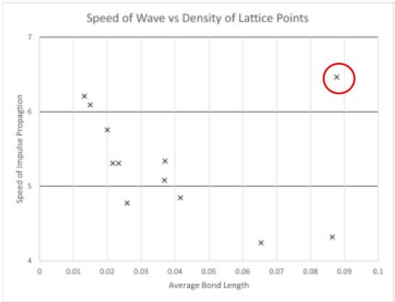


MODEL FITS

$-3.065 * \ln(x) + 17.016$
($R^2 = 0.9194$)
OR
$7.5795 * e^{-0.009x}$
($R^2 = 0.728$)
OR
$40.472 * x^{-0.592}$
($R^2 = 0.9975$)

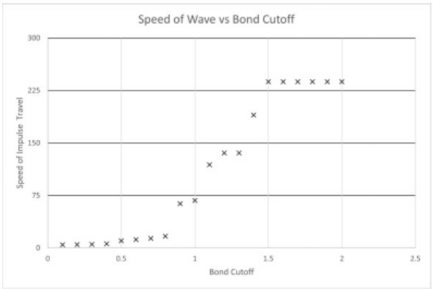
9

Speed of Sound Tests – Test III



10

Speed of Sound Tests – Test IV



Num_Neighbors → 27

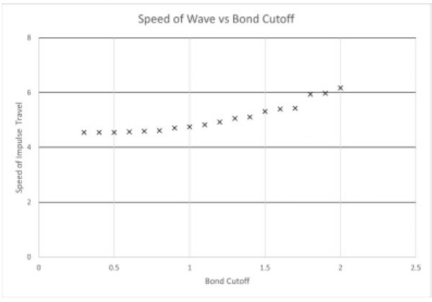
Resembles a sigmoid (smooth transition from one state to another)

State One → 6 neighbor system

State Two → All 27 possible neighbors

11

Speed of Sound Tests – Test IV



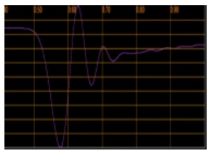
Num_Neighbors → 6

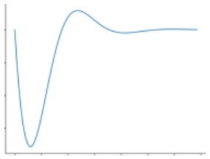
Smooth transition up to State One

12

Transfer Function

$$M_p = \frac{y_{peak} - y_{steady\text{-}state}}{y_{steady\text{-}state}} \rightarrow \zeta = \frac{\ln^2 M_p}{\sqrt{\ln^2 M_p + \pi^2}} \rightarrow \omega_n = \frac{\omega_d}{\sqrt{1-\zeta^2}} \rightarrow H(s) = G_{DC} \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

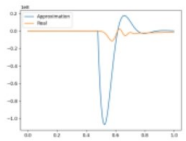




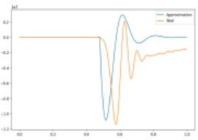
- Prediction → can be accurately modelled by 2nd order system
- Did not find a python library that does this
- Peak is easy, steady state is easy, damped freq is medium
- Damping ratio is hard (and critical)
- Gain never worked

13

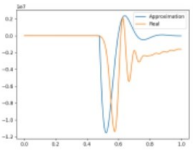
Transfer Function → K=-400



damping ratio = 0.49934
natural frequency = 19.493 Hz
frequency (damped) = 13.793 Hz
Gain = -1599867.4



damping ratio = 0.49935
natural frequency = 19.49383 Hz
frequency (damped) = 13.79310 Hz
Gain = -159986.7

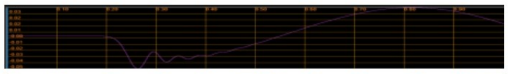


damping ratio = 0.4
natural frequency = 30 Hz
Gain = -106658

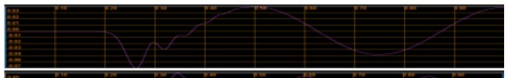
14

Frequency Response → K=-4000

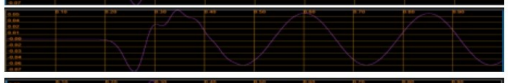
1 Hz



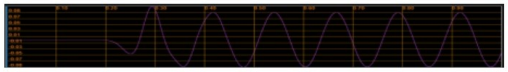
2 Hz



4 Hz



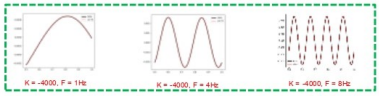
8 Hz



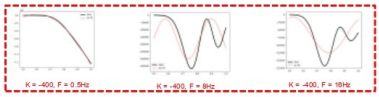
15

Frequency Response

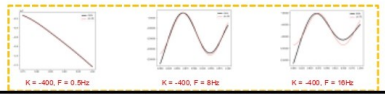
Sin fitting Issues at low K – Depending on Response, Select Bounds



→ T = [0.5, 1.0]



→ T = [0.5, 1.0]

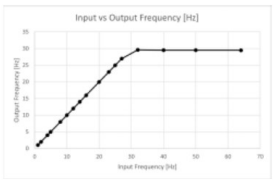


→ T = [0.8, 1.0]

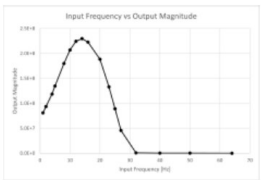
16

Frequency Response

Frequency Response → K = -4000, M = 40, D = 0.98, dt = 0.001, input_dist = 0.02



We expect the output frequency to match the input frequency. It appears that at a threshold frequency (here around 30Hz), the output is capped.

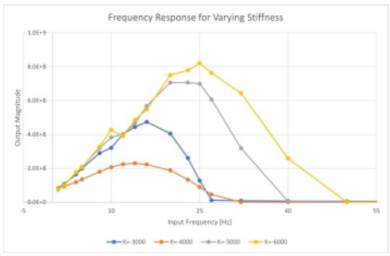


Graphing the magnitude (units of sound/pressure) of the output as a function of frequency gives us a peak at 14Hz. Output is "dead" after 30Hz

17

Frequency Response

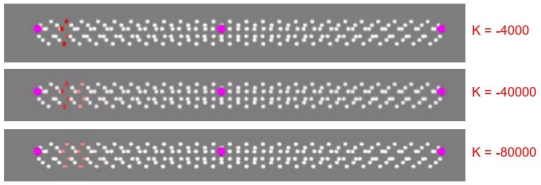
Frequency Response → K = -5000, M = 40, D = 0.98, dt = 0.001, input_dist = 0.02



18

Frequency Response

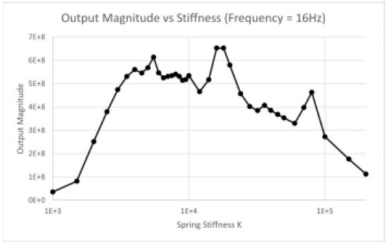
Video and Graphics → driving frequency = 8Hz and disp. = 0.02



19

Frequency Response

Instead of looking at magnitude vs input frequency for several stiffness, look at magnitude vs stiffness for one or more input frequency's (16Hz here)

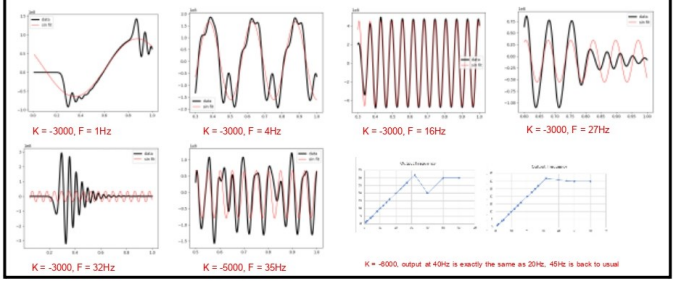


Logarithmic scale for clarity

20

Frequency Response

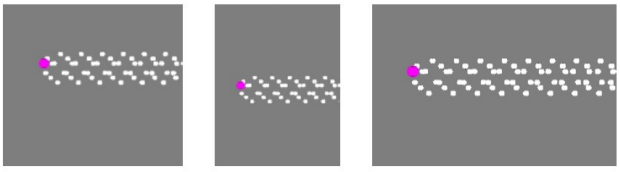
Miscellaneous



21

Minor Bug Fix

Glitch when K is very low → displacement is very high (K = -400 and disp. = 0.04)



Position and velocity discontinuities. Realized I was fixing position, then doing physics on position

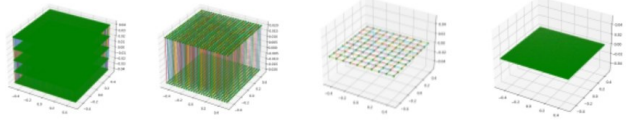
Try fixing position after doing neighbor physics. Still velocity discontinuity

Fix position after doing math AND fix velocity to zero at boundaries

22

Chladni Plates

Creating some neighbor corrected point clouds for the plate



50x50x50 in 1.1x1.1x1.1 volume

25x25x25 in 1.1x1.1x1.1 volume

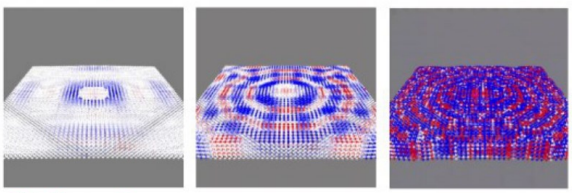
10x10x10 in 1.1x1.1x1.1 volume

50x50x10 in 1.1x1.1x1.1 volume

23

Chladni Plates

Video and Graphics → 50x50x50 in a 1x1x1 volume



8Hz input frequency with displacement = 0.01 coming from the left face. K = - 200

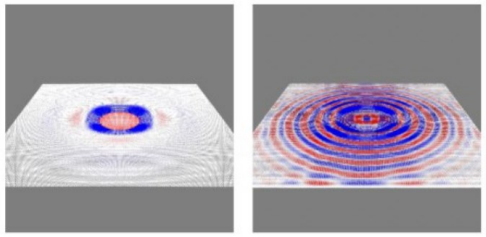
16Hz input frequency with displacement = 0.01 coming from center. K = -200

32Hz input frequency with displacement = 0.02 coming from center. K = -200

24

Chladni Plates

Video and Graphics → 150x150x10 in a 1x1x1 volume



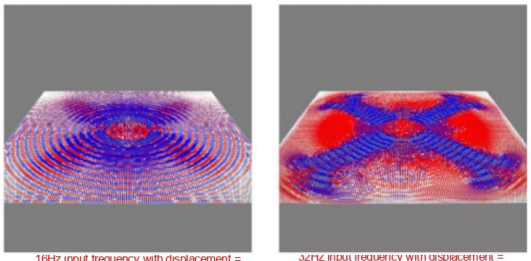
0.03 coming from center. K = - 50

0.01 coming from center. K = - 50

25

Chladni Plates

Video and Graphics → 150x150x10 in a 1x1x1 volume



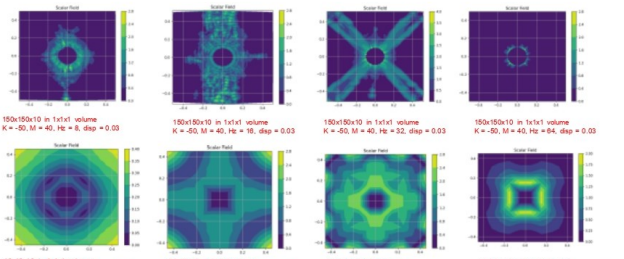
16Hz input frequency with displacement = 0.01 coming from center. K = - 50

32Hz input frequency with displacement = 0.01 coming from center. K = - 50

26

Chladni Plates

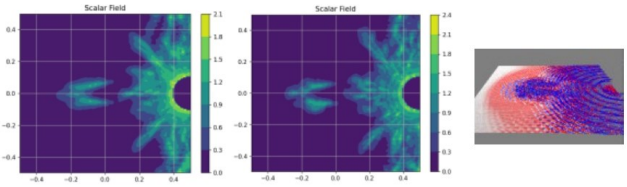
Averaging speed over final 20 frames, plotted as a 2D colormap



27

Chladni Plates

Averaging speed over final 20 frames, plotted as a 2D colormap

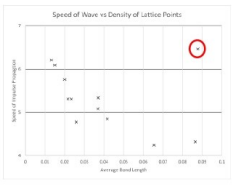


28

Redoing V_{sound} vs Density

Previous Model

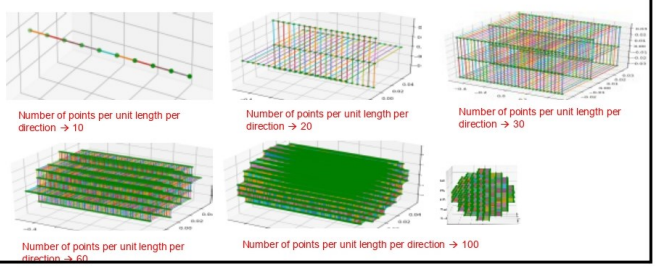
- X→ Average Bond Length *not a good measure, try plotting as X → Num Points / Unit Volume
- Point cloud creation and inconsistent neighbors *now point neighbors are perfect
- Distance Measure Update *before the distance was 0.9 (0.05 of margin on either end). Now the first sensor is driven and the second sensor is read and the distance is the difference between the two at t=0



29

Redoing V_{sound} vs Density

New Model



Number of points per unit length per direction → 10

Number of points per unit length per direction → 20

Number of points per unit length per direction → 30

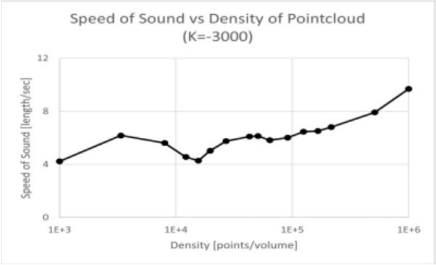
Number of points per unit length per direction → 60

Number of points per unit length per direction → 100

30

Redoing V_{sound} vs Density

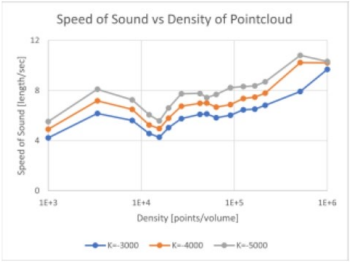
New Model



31

Redoing V_{sound} vs Density

New Model



32

Research on Chladni Plates

https://thelig.ht/chladni/

$$h(x, y, t) = \sin(2\pi f t)s(x, y)$$

h is the height of the plate at (x,y,t) and **s** is the standing wave

$$s(0, 0) = 0$$
$$s(\pm 1, y) = 0$$
$$s(x, \pm 1) = 0$$

Boundary Conditions → standing wave fixed at edges and center ***

$$-\left(2\pi \frac{f}{c}\right)^2 s - \left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right)$$

$$\frac{\partial^2 h}{\partial t^2} = c^2\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2}\right)$$

Wave Equation

33

Research on Chladni Plates

https://thelig.ht/chladni/

$$s(x, y) = \sum_{n=0}^{\infty}\sum_{m=0}^{\infty} a_{nm}\sin(\pi n x)\sin(\pi m y)$$

Fourier Series product version of s(x,y)

$$-\left(2\pi \frac{f}{c}\right)^2 s - \left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right)$$

$$s(x, y) = a\sin(\pi n x)\sin(\pi m y) + b\sin(\pi m x)\sin(\pi n y)$$

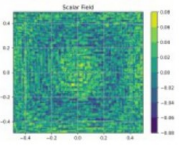
Final set of solutions to the problem

34

Chladni Plates

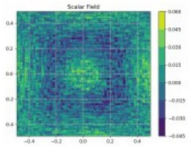
NEW METHOD – Run a sin fitting algorithm to every single particle

F(t) = A sin(w * t)



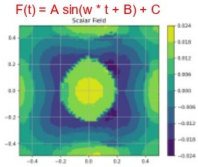
50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 700 → 750

F(t) = A sin(w * t + B)



50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 700 → 720

F(t) = A sin(w * t + B) + C



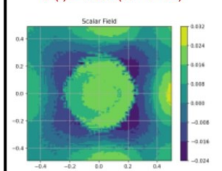
50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 700 → 720

35

Chladni Plates

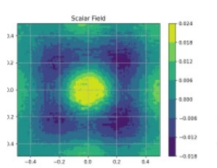
NEW METHOD – Run a sin fitting algorithm to every single particle

F(t) = A sin(w * t + B)



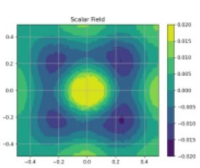
50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 650 → 900

F(t) = A sin(w * t)



50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 650 → 900

F(t) = A sin(w * t) + B



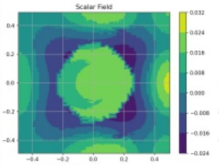
50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 650 → 900

36

Chladni Plates

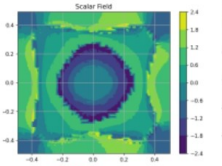
NEW METHOD – Run a sin fitting algorithm to every single particle

F(t) = A sin(w * t + B) + C
AMPLITUDE



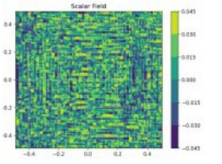
50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 650 → 900

F(t) = A sin(w * t + B) + C
PHASE



50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 650 → 900

F(t) = A sin(w * t + B) + C
OFFSET



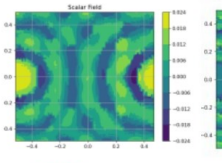
50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 650 → 900

37

Chladni Plates

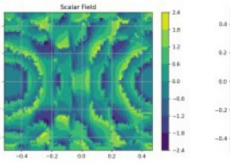
NEW METHOD – Run a sin fitting algorithm to every single particle

F(t) = A sin(w * t + B) + C
AMPLITUDE



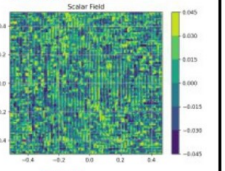
50x50x50
K = -400
M = 40
Hz = 32
disp = 0.02
Frames 800 → 900

F(t) = A sin(w * t + B) + C
PHASE



50x50x50
K = -400
M = 40
Hz = 32
disp = 0.02
Frames 800 → 900

F(t) = A sin(w * t + B) + C
OFFSET



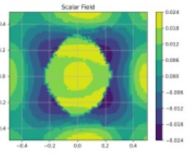
50x50x50
K = -400
M = 40
Hz = 32
disp = 0.02
Frames 800 → 900

38

Chladni Plates

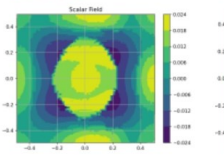
NEW METHOD – Run a sin fitting algorithm to every single particle

F(t) = A sin(w * t + B) + C
AMPLITUDE



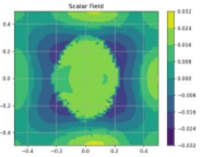
50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 500 → 1000

F(t) = A sin(w * t + B) + C
AMPLITUDE



50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 700 → 800

F(t) = A sin(w * t + B) + C
AMPLITUDE

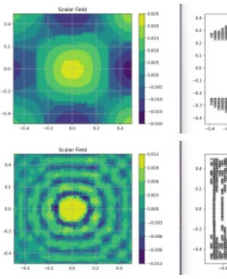


50x50x50
K = -400
M = 40
Hz = 13
disp = 0.02
Frames 800 → 900

39

Chladni Plates

NEW METHOD – Run a sin fitting algorithm to every single particle



50x50x50 CENTER
K = -400
M = 40
Hz = 10
disp = 0.02
Frames 800 → 900

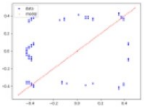
50x50x50 CENTER
K = -400
M = 40
Hz = 55
disp = 0.01
Frames 800 → 900

40

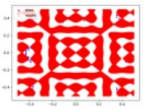
Chladni Plates

NEW METHOD – Run a sin fitting algorithm to every single particle

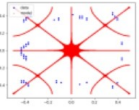
$$0 = a\cos(\pi nx)\cos(\pi my) + b\cos(\pi mx)\cos(\pi ny)$$



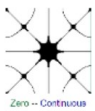
a = 1
b = -1
m = 15
n = 2



a = 1
b = -1
m = 15
n = 2



a = 1
b = -1
m = 5
n = 1

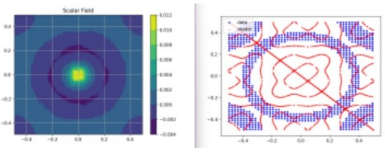


Matches online sources

41

Chladni Plates

NEW METHOD – Run a sin fitting algorithm to every single particle



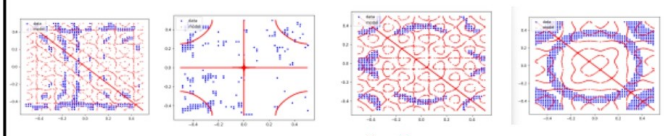
Hz = 4
a = -1
b = -2
m = 2
n = 11

42

Chladni Plates

NEW METHOD – Run a sin fitting algorithm to every single particle

$$0 = a\cos(\pi nx)\cos(\pi my) + b\cos(\pi mx)\cos(\pi ny)$$



Hz = 32
a = 5
b = -6
m = 6
n = 19

Hz = 55
a = -1
b = -1
m = 1
n = 3

Hz = 16
a = -1
b = 1
m = 8
n = 17

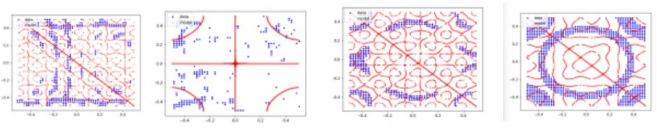
Hz = 4
a = -1
b = -2
m = 2
n = 11

43

Chladni Plates

NEW METHOD – Run a sin fitting algorithm to every single particle

$$0 = a\cos(\pi nx)\cos(\pi my) + b\cos(\pi mx)\cos(\pi ny)$$



Hz = 32
a = 5
b = -6
m = 6
n = 19

Hz = 55
a = -1
b = -1
m = 1
n = 3

Hz = 16
a = -1
b = 1
m = 8
n = 17

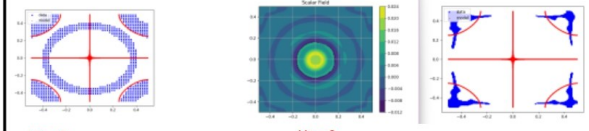
Hz = 4
a = -1
b = -2
m = 2
n = 11

44

Chladni Plates

NEW METHOD – Run a sin fitting algorithm to every single particle

$$0 = a\cos(\pi nx)\cos(\pi my) + b\cos(\pi mx)\cos(\pi ny)$$



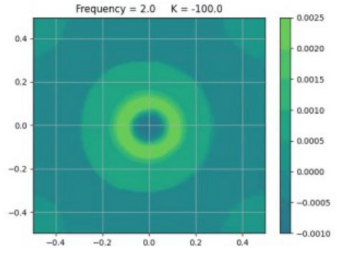
Hz = 4
a = 1
b = -1
m = 1
n = 3

Hz = 8
a = -2
b = -2
m = 1
n = 2

45

Chladni Plates

What If → each frame of a gif was a standing wave caused by a varying frequency



Hz = 2 → Hz = 34
Frame = 1 → Frame = 161

All plots color scaled equally. Hard Reset between frames/rounds

20 FPS

K = -100, M = 40, dt = 0.001, D = 0.98

Input_Disp = 0.003

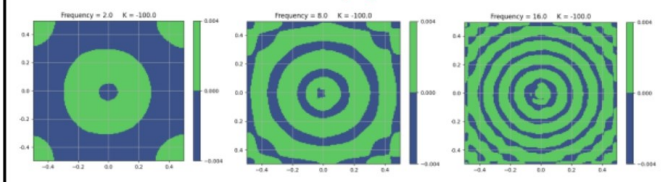
Capture_Frame_Start = 1500 (1.5 sec)
Capture_Duration = 100 (0.1 sec)

46

Chladni Plates

What If → each frame of a gif was a standing wave caused by a varying frequency

Filtered by sign



Hz = 2
Frame = 1

Hz = 8
Frame = 31

Hz = 8
Frame = 71

47