ENPH 455 Final Report

Finite Difference Time Domain Simulation of Acoustic Metamaterials

by

Matthew Duke

A thesis submitted to the
Department of Physics, Engineering Physics and Astronomy
in conformity with the requirements for
the degree of Bachelor of Applied Science

Queen's University
Kingston, Ontario, Canada
April 2020

Abstract

Since their initial documentation in 1993, multitudes of novel uses for phononic crystals have been discovered and incorporated into products such as micro-electromechanical systems (MEMS) devices. However, construction and testing of different phononic crystal designs and geometries is costly and time consuming, often requiring highly specialized equipment.

This thesis aims to simplify this process using a finite-difference time-domain simulation of acoustic wave transmission through orthographic materials. The result of Marc Cameron's 2018 undergraduate thesis, the application *phonomena*, was extended to incorporate non-uniform dynamic meshing, high performance computing methods, and usability improvements including visualization, analysis tools and persistent storage of simulation results. Performance was tested on a variety of different computing systems with varying CPU core counts, clock speeds and instruction sets, with a best-case improvement up to 3x faster per time step. Strengths of each implementation tested were discussed leading to recommendations for future development.

All source code is available in a public repository, available on GitHub, along with a Software Description document with design details, test scripts and freely available releases of *phonomena*.

Acknowledgments

I'd like to first thank my supervisor, Dr. James Stotz for his guidance and patience, all while dealing with the struggles of remote meetings and time zones changes. Thank you also to Dr. Marc Dignam and Chelsea Carlson for their review and feedback of my mid term report and to Erik Sauer for his excellent explanation of FDTD methods.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The study of phononic crystals was officially born in 1993, with the publication of a paper drawing parallels between periodic elastic composites, and the study of photonic crystals involved in light transmission [1]. These composites, commonly referred to as phononic crystals, can be broadly defined as a periodic structure of scattering regions embedded in a matrix. This definition encapsulates a wide variety of physical materials and devices, some intentionally designed, others found in nature, such as regularly spaced trees in a forest [2]. Like photonic crystals, these devices act as a wave medium, supporting elastic waves rather than electromagnetic. The substrate can therefore influence the properties of the wave by varying the properties and locations of the periodic inclusion regions, specifically the density and stress tensor. These parameters provide control over the frequency response of the substrate, subsequently providing a framework for wave filters and waveguides.

Coupling phononic crystals with piezoelectric substrates and interdigitated transducers transforms acoustic waves to and from a more useful electrical signal. The stability, reliability and small scale make their use in electronic systems increasing common, as well as in more novel applications, such as pulse compression filters,

microfluidic pumps and single photon emitters.

1.1 Motivation and Background

The work outlined in this thesis builds on a body of knowledge within the Queen's Physics department and Dr. Stotz's research team, which has been investigating novel uses of surface acoustic wave (SAW) devices for over 15 years. Some outcomes of this work have included coherent spin transport using quantum dots produced a piezoelectric field generated by acoustic phonons [3]. A 2009 masters thesis by Joseph Petrus formed the basis of this study, in which the author developed a SAW wave simulation to study and validate the band structure of physical phononic crystals, which was later adapted to the Python language in 2018 by Marc Cameron [4][5]. This project aims to continue development of the application produced by Cameron, dubbed *phonomena*.

1.2 Project Definition and Scope

The work by Cameron consists of a functional finite-difference time-domain simulator including first-order absorption and traction free boundary conditions. This thesis aims to address several limitations of the software including:

- Static, uniform mesh generation,
- No persistent storage of results,
- Limited visualization and analytical tools,
- Limited documentation,

- Non-functional periodic boundary conditions.

This thesis aims to add functionality by addressing some of these limitations while adding structure and documentation to simplify future development. The original goals identified in the fall semester included:

- Dynamic non-uniform mesh generation,
- Evaluation of performance optimization libraries,
- Inclusion of Bloch periodic boundary conditions,
- Documentation for development and use,
- GUI frontend (time-permitting).

The significance of visualization techniques was realized during development and testing of the non-uniform grid algorithm. This increased the priority of the GUI frontend, which ultimately reduced the priority of the Bloch periodic boundary conditions. Persistent data storage was also added as a priority to allow the results to be saved and analyzed at a later date.

As an entirely software based project, environmental and economic considerations were not application. Care was taken during the design phase to minimally impact a user's computer and to avoid introducing any system vulnerabilities.

Chapter 2

Phononic Crystal Simulator

2.1 Theory

2.1.1 Elasticity

Internal forces in elastic structures are described by stress, S, or force per unit area, and strain, T, a measure of internal displacement. For small displacements, elastic forces can be modelled using Hooke's Law, where force is linearly proportional to displacement.

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial xi}\right) \tag{2.1}$$

where S is the stress tensor, u is the displacement, and indices i and j are axis components. Again, using a linear approximation, the strain can be related to strain by:

$$T_{ij}(S_{kl}) \approx \left(\frac{\partial T_{ij}}{\partial S_{kl}}\right) S_{kl} \tag{2.2}$$

Force and displacement can therefore be related such that $T_{ij} = C_{ijkl}S_{kl}$ where C_{ijkl} is given by:

$$C_{ijkl} = \frac{\partial T_{ij}}{\partial S_{kl}} \tag{2.3}$$

Starting from Newton's second law, the simplified equation of motion forms a second order ordinary differential equation:

$$\rho \frac{\partial^2 u_i}{\partial t^2} = \frac{\partial T_{ij}}{\partial x_j} \tag{2.4}$$

where ρ is material density and $x_{i,j,k}$ are dimension components. These equations form the foundation required to solve for wave transmission through elastic materials.

2.1.2 Finite-Difference Time Domain Techniques (FDTD)

Also known as Yee's method for it's creator, mathematician Kane Yee, this numerical analysis technique was developed for computation of electromagnetic waves through matter. Because it acts in the time domain, FDTD can be used to simulate a wide range of frequencies.

This simulation method works by discretizing a solid into "Yee cells", shown in Figure 2.1, where different components of the cell are spaced by staggered half grids. The physical locations of T_{ij} and u_i can be combined with a discrectized version of the

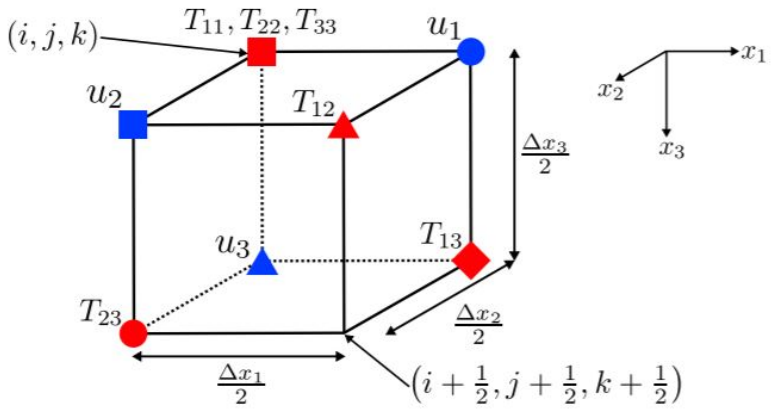


Figure 2.1: Yee cell showing staggered half grid of material properties.

equation of motion (Eq. 2.4) to produce an iterative numerical solution to material deformation and wave propagation. A more complete derivation is given by Petrus [4].

The solver portion of *phonomena* performs a series of steps at each time step. First, a wave impulse is applied to the y-axis face of the simulation. This pulse is typically in the form of a pure sinusoidal or a Ricker wavelet, which is more representative of seismic waves propagating through homogeneous media [6]. The strain tensor components are calculated using the methods described above, followed by the application of boundary conditions relevant to strain. Currently, the only condition is a traction-free boundary at the bottom ($z = 0$) face. This is followed by similar steps for displacement components and its boundary conditions, including a traction free condition at $z = 0$ and absorbing boundaries on all other surfaces. This process is repeated until each time step has been solved.

2.2 Research

The goals of this project were simplified into two main categories: decreasing simulation time and increasing usability. Testing third-part libraries and calculation techniques fell into the first category, along with non-uniform meshing.

Two approaches were identified to improve simulation performance. The first and most obvious was to decrease the calculation time for each time step. Python is simple and highly readable but is not high performing compared to lower level compiled languages such as C++ or Java. Calculation speed can be increased by using more powerful hardware, employing hardware acceleration to maximize the use of the available hardware, or by changing how the code is compiled. A variety of

third party libraries for Python were evaluated for their potential to perform these tasks. The most promising results are shown in Table 2.1. Bohrium was not tested in this project due to the limited operating system requirements, though it would be useful to investigate in the future.

Table 2.1: Non-exhaustive summary of third-party Python performance optimization libraries.

Library	Description	Limitations
Numba	JIT compiler using LLVM, with auto parallelization of certain calculations using GPU or CPU	Additional overhead during compilation.
Bohrium	Auto GPU acceleration of calculations involving NumPy arrays	Not compatible with Windows or Python >3.7.
Intel Distribution for Python	Intel generated Python interpreter with CPU based optimizations included at a low level	Requires specific hardware support (Intel SIMP) [7]

The existing array calculations were evaluated against documented best practices, the most major of which include vectorizing calculations, limiting in-memory data copying and minimizing the total number of operations. [8][9]. The existing implementation of *phonomena* already followed most of these practices, and any potential improvements involved restructuring many of the existing matrices in the program. Due to time constraints, the calculations and array order were not changed.

The second method was to decrease the number of calculations required. This involved reducing the density of points in the mesh without sacrificing accuracy of results. In a standard mesh, details are portrayed at a uniform resolution across the entire simulation space. That level of resolution may be required to capture fine structures or irregular shapes within the mesh, but unnecessary in other regions. These types of meshes are made more complex when using FDTD because of the

staggered half cell layout (see Figure 2.1), though it has been demonstrated to show higher convergence and resource consumption compared to a uniform mesh of the same size [10].

A necessary ability of any scientific simulation tool is the ability to save the results for analysis and inspection without the need for reruns. This decreases importance on simulation run time, as the data from a single run can be reviewed at any time. Several data storage methods were compared for this purpose, including database formats SQLite, or Excel, simpler text-based formats such as CSV and binary serialization ("pickling" in Python). Ultimately, the HDF library was chosen for its prevalence in scientific simulations, common format and ease of implementation in Python. This library is designed for large datasets and offers relatively low level access to file operations, simplifying code profiling and optimization.

2.3 Implementation and Design

2.3.1 Non-Uniform Meshing

Non-uniformity in Yee cell side lengths required changes to the discrete spatial derivative to account for varying Δx_i values. For example, the spatial derivative of some function A, along axis 1 was originally:

$$D_1[A(i,j,k,l)] = \frac{1}{\Delta x_i}\left[A\left(i+\frac{1}{2},j,k,l\right) - A\left(i-\frac{1}{2},j,k,l\right)\right] \tag{2.5}$$

where Δx_i was held constant. In a non-uniform grid, this value becomes a 1-dimensional array. To take the numerical derivative along the half grid, Equation

2.5 becomes:

$$D_1[A(i,j,k,l)] = \frac{2}{x_i(i+\frac{1}{2}) - x_i(i-\frac{1}{2})} \times \left[A\left(i+\frac{1}{2},j,k,l\right) - A\left(i-\frac{1}{2},j,k,l\right) \right] \quad (2.6)$$

This process was repeated for other derivatives by replacing the Δx_i term with the appropriate spacing term to line up with the staggered half grid structure. This process was non-trivial, as the correct spatial indices depended on the Yee cell position of the input variable, A. Implementation in Python using NumPy matrices was performed using the element-wise mean of each element of Δx_i for the staggered grid and the difference for full grid spacing. Array shapes were enforced to match the shapes of the stress and displacement matrices.

```
# full delta (n-1 elements)
self.fdx = (self.x[1:] - self.x[:-1]).reshape((x-1,1,1))
self.fdy = (self.y[1:] - self.y[:-1]).reshape((1,y-1,1))
self.fdz = (self.z[1:] - self.z[:-1]).reshape((1,1,z-1))


# staggered delta (n-2 elements)
self.sdx = np.mean([self.fdx[1:,:,:], self.fdx[:-1,:,:]], axis=0)
self.sdy = np.mean([self.fdy[:,1:,:], self.fdy[:,:-1,:]], axis=0)
self.sdz = np.mean([self.fdz[:,:,1:], self.fdz[:,:,:-1]], axis=0)
```

The algorithm used to dynamically generate a mesh, based on inclusion regions and boundary constraints, was broken into a sequence of 5 steps.

1. X and Y outer boundaries are added to the mesh.

2. The inclusion is filled with evenly spaced grid lines that are spaced as closely as possible to the user-specified minimum grid width.

3. Grid lines are added around each inclusion region based on some mesh ramp function. By default this function is linear, with a user specified slope.

4. Lines that are spaced much more closely than the minimum spacing parameter are removed. This step is used to fix overlap issues between inclusion regions borders.

5. Finally, all remaining spaces are uniformly filled using the user-specified maximum grid spacing value.

This procedure is used along the X and Y axes, while Z is populated using a uniform grid. Mesh density is dramatically reduced using this approach, compared to a uniform mesh of the same minimum density.

2.3.2 Data Storage

Introducing disk operations to the simulation required careful planning because it introduces the possibility of a severe bottleneck, especially on systems with hard disk drives rather than solid state storage. While this step is a necessary component of the simulation, poor implementation would negate any performance gain found elsewhere. A general approach was implemented where the HDF file writer could run as a unique thread or process, using Python's threading or multiprocessing libraries, respectively. Performing file operations in a separate thread allows the main portion of the code to concurrently perform calculations without waiting for file operations to finish. This

approach has the disadvantage of additional overhead, both in starting the second process and in copying data and passing it to the writer.

The fundamental difference between each approach to parallelization revolves around the global interpreter lock (GIL) in Python. This features limits the Python interpreter to a single CPU thread, simplifying resource allocation and removing race conditions. The threading module imitates concurrency by taking advantage of CPU idle time to run other operations. However, if no idle time is available, performance will be the same or worse due to increased overhead involved in spawning and managing threads. By contrast, multiprocessing creates independent instances of the Python interpreter, bypassing the GIL. Because threads no longer share memory, inter-process communication is more complicated and slower than with the threading model.

2.3.3 Performance Optimization

Due to Python's popularity, particularly in machine learning and artificial intelligence fields, there is an overwhelming variety of libraries promising increased performance using unique approaches. To compare the performance of each method, a framework was developed such that the simulation portion of the application exists as a common class, that can be inherited and modified as needed. This maximizes the amount of overlap between simulation methods to help ensure performance comparisons are valid.

Baseline

Five different solving methods were created and are included with the application. The simplest approach runs the simulation in a single thread, with no parallelization or hardware optimization, aside from those included in the standard NumPy library. This method has an advantage in simplicity, as computer resources are allocated to a single purpose, with as little overhead as possible.

CPU Parallelization

The second and third implementations are attempts at parallelization using the threading and multiprocessing techniques discussed earlier. Both spawn a number of worker threads (equal to CPU thread count) which accept function inputs for processing. Calculations that have no cross dependencies are placed in a queue and computed simultaneously by the first free worker. The goal of these methods was to change the structure from a sequence of calculations, shown in Figure 2.2, in order to take advantage of extra CPU cores and/or idle time during each step. With

Figure 2.2: Block diagram demonstrating linear function execution of strain calculations.

parallelization, the calculation sequence follows the structure shown in Figure 2.3. Theoretically, assuming each calculation takes the same amount of time, performance

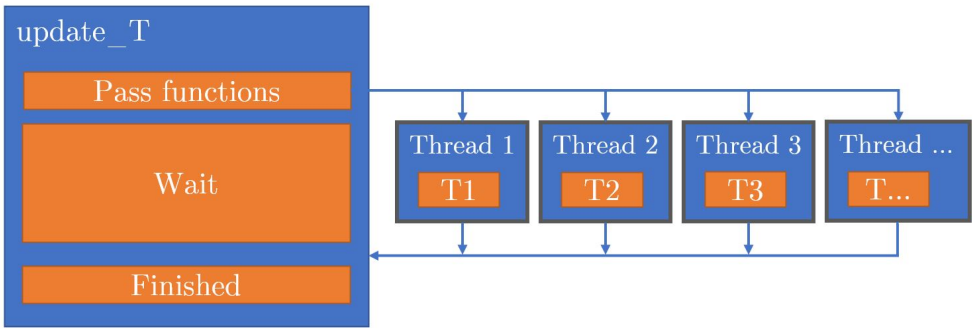


Figure 2.3: Block diagram demonstrating parallel execution of strain calculations.

can be improved by a factor of the number of CPU threads. In reality, this approach adds significant processing time during data transfer.

The advantage of the threading module over multiprocessing is faster data transfer. Due to the GIL, Python threads can share memory, so little processing is needed. The downside of this approach is that it remains limited to a single CPU core. Performance improvements are only possible if there is CPU idle time during calculations. NumPy includes low-level accelerations for vectorized operations, so idle time may be insignificant.

Conversely, multiprocessing has the opposite strengths and weaknesses. Calculations are no longer bound by single-thread performance, and can take place simultaneously. The drawback is in data mapping between the parent and child processes. Sharing data involves memory mapping the 3-dimensional data arrays (u_i and T_{ij}), which requires reshaping them into 1 dimension, then back into the original shape on reception. This process was much more difficult to implement in Python and is far less readable.

Numba

Next, a solver was designed to accommodate the Numba library. Numba has several features, including GPU and CPU acceleration, better utilization of CPU capabilities (using SIMD vectorization) and just-in-time (JIT) compilation of functions. This solver focused on JIT compilation, though future versions could explore other features.

Function level compilation was forced for functions such, such as 'update_T', shown in Figure 2.2. Compiled function do not share memory with the rest of the program; instead, it exists as compiled machine code based on the LLVM compiler infrastructure. All required data must be explicitly passed to and from this function. The initial compilation step adds several seconds, but should only need to be run once, as the result is cached in persistent storage. Like the other methods, extra time is required to pass data between functions, and some efficiency is lost in this process.

Remote Solver

The final simulation method was the simplest in theory, where hardware is optimized rather than the code. It is common in scientific computing to run simulations on dedicated hardware that is known to perform well for specific tasks. The fifth and final solver implementation involved a client/server model in which tasks could be offloaded to a dedicated server which would then return the result on completion. In this model, a large portion of the simulation time is dependent on the network transfer speed to copy data to the server to initialize the parameters, and at the end of the run to transfer the HDF file to the client. The use-case of this implementation is for very long running simulations of several hours or more, or when the program is running on very old or underpowered hardware. Eventually, the server side solver

can be optimized for the given hardware and better utilize multiple cores or optimize the task for GPU acceleration.

Intel Distribution for Python

This Python distribution is compiled and made available by Intel to optimize low level Python bindings to better utilize certain CPU features such as the SIMD instruction set [7]. It is designed to accelerate NumPy and other scientific applications using the Intel Math Kernel Library. Because it is freely available and doesn't require any changes to existing code, each of the methods described above were tested with the standard Python 3.8.2 interpreter and with the 2020 release of the Intel Distribution for Python (Python 3.7.7).

2.3.4 User Interface

A significant portion of work involved the development of a graphical user interface (GUI). The importance of this feature changed drastically over the course of this project due to unexpected demand from the non-uniform meshing portion. Verifying the accuracy of the dynamically generated mesh was very difficult without the ability to view changes in real time. The result was a highly functional user interface that does not require any scripting knowledge from the user. Simulation geometry is calculated and displayed in real-time to properly display the resolution of regions of interest.

Material and simulation properties can then be specified for the substrate and inclusion regions. Finally, the simulation result is displayed as plots displaying mass density, displacement over time, and 1 and 2 dimensional Fourier transforms. Screen

captures of the user interface can be found in Appendix B.

2.3.5 Application Structure & Documentation

Significant effort was made throughout the design of *phonomena* to reduce code duplication and to create a framework to assist with future development. This was partially accomplished by careful organization of the source files. This ensured *phonomena* could be imported into other scripts and used in API form, or run with a GUI. It can also be bundled into a standalone folder with all dependencies included, providing portability and not requiring prior Python installation or knowledge.

A software description document has been developed concurrently with the application itself to describe implementation details at a greater level of detail than what is necessary here.

2.4 Testing

2.4.1 Data Storage

The application was profiled using the Intel vTune application to identify the most significant hotspots. When using the threading module, a 25,000 point simulation of 1000 time steps took 130 seconds to complete. Dynamic profiling revealed a significant portion of CPU time was spent by the HDF module itself, while collecting data and writing to disk. In fact, various HDF modules used over 50% of the CPU time in some cases. File operations did not release Python's GIL, meaning that the main loop was blocked by the writer thread, preventing the simulator from running calculations. Switching to the multiprocessing method showed a significant improvement, with a 88 seconds run-time for the same test case. This result was not unexpected and similar

results should be observed on most systems because very few modern CPUs include less than 2 cores.

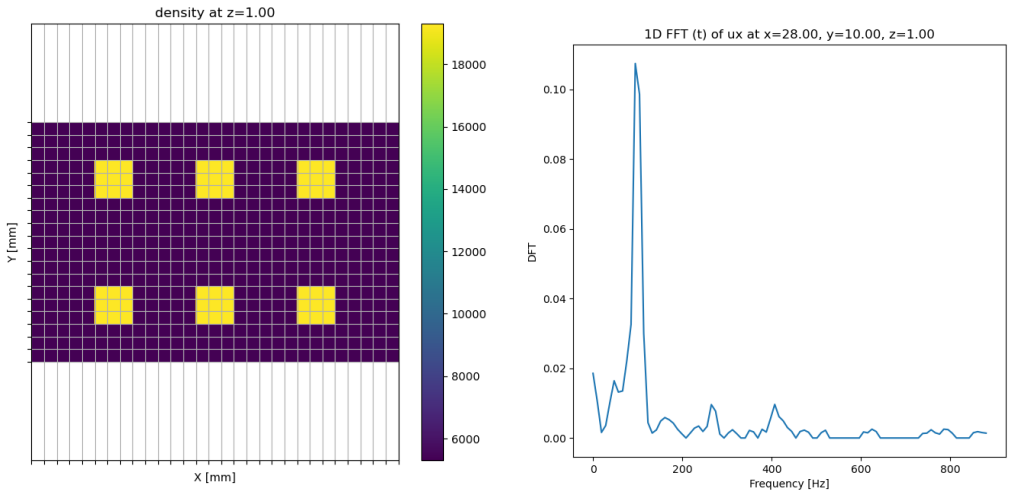
2.4.2 Non-Uniform Meshing

Non-uniform meshing was tested using three simulations with the same geometry, where the maximum and minimum mesh spacing parameters were varied from 0.5 to 1.0 mm. The test consisted of coarse and fine uniform mesh and a non-uniform mesh with minimum spacing equal to the fine mesh and maximum spacing equal to the coarse mesh. A 100Hz sinusoidal input signal was applied to the left face, and the same materials of a gallium arsenide substrate and gold inclusions were used in all three simulations. As expected, the total simulation time correlated to the total points in the mesh.

Table 2.2: Uniform vs. non-uniform mesh generation speed comparison.

Mesh geometry			Simulation time
Max spacing [mm]	Min spacing [mm]	Total points	
1.0	1.0	3780	17.05
0.5	0.5	14760	56.27
1.0	0.5	7308	29.44

The benefit of fine mesh resolutions is in greater approximation to irregular geometry. For example, the shape of the inclusion regions in the coarse mesh is simply a square. A frequency analysis of this geometry shows a clean signal at 100Hz, as expected, with little to no shifts in the frequency spectrum. Because of the much higher elasticity of the gold inclusions, one would expect a portion of the frequency spectrum to be shifted slightly higher than 100 Hz (Figure 2.4). It should be noted that a Hanning window was applied the displacement data prior to the frequency



(a) (b)

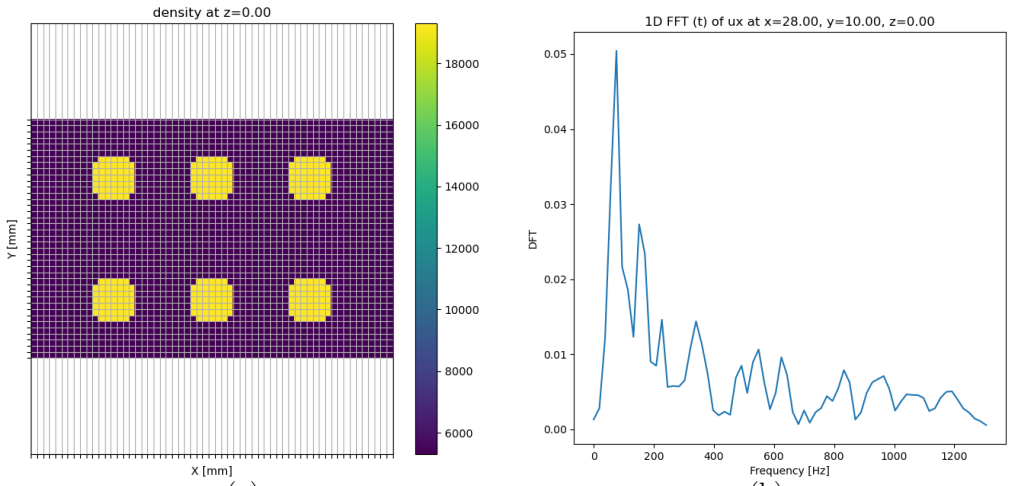
Figure 2.4: a) Coarse density plot overlaid with mesh grid. Substrate is GaAs and inclusions are Au.
b) Frequency analysis following simulation of 100Hz sinusoidal input.

analysis using a fast Fourier transform (FFT). The result was then normalized using a scaling factor of 1

$\sqrt{n}$.

This was observed in the fine resolution mesh, at the cost of over 3x longer runtime (Figure 2.5). This mesh is a closer approximation to a circle, and shows secondary peaks appearing around 180 Hz and 300 Hz.

Some of the same features are present in the non uniform example, where the main peak is wider and encompasses the secondary 180 Hz peak. A second peak can also be seen around 300 Hz. While there is slight decrease in Fourier transform resolution especially in higher frequencies, this simulation required about half the time of the fine mesh simulation. For very large mesh sizes or long-running simulations, non-uniform meshing clearly demands less of a trade-off between run time and result accuracy.



(a) (b)

Figure 2.5: a) Fine density plot overlaid with mesh grid. Substrate is GaAs and inclusions are Au.
b) Frequency analysis following simulation of 100Hz sinusoidal input.

2.4.3 Performance Optimization

A suite of tests were run on three different computer systems with varying computing hardware to help build an accurate picture of the potential and limitations of each performance optimization method. Tests were performed against a 60x60x20 mesh with a total of 78,141 cells for time steps from varying 1000 to 5000 in increments of 1000, with 2 trials per time step. Detailed specifications for each target system are given in Appendix B.

The 2 core system results were interesting, as all solver methods performed roughly the same. Slightly worse performance was observed in the multiprocessing example, which was attributed to the low physical core count. In this case, the overhead involved in memory mapping data between each process outweighed any benefit gained by performing simultaneous calculations. Additionally, this CPU was designed for mobile devices, with lower clock speeds and smaller CPU cache. These limitations

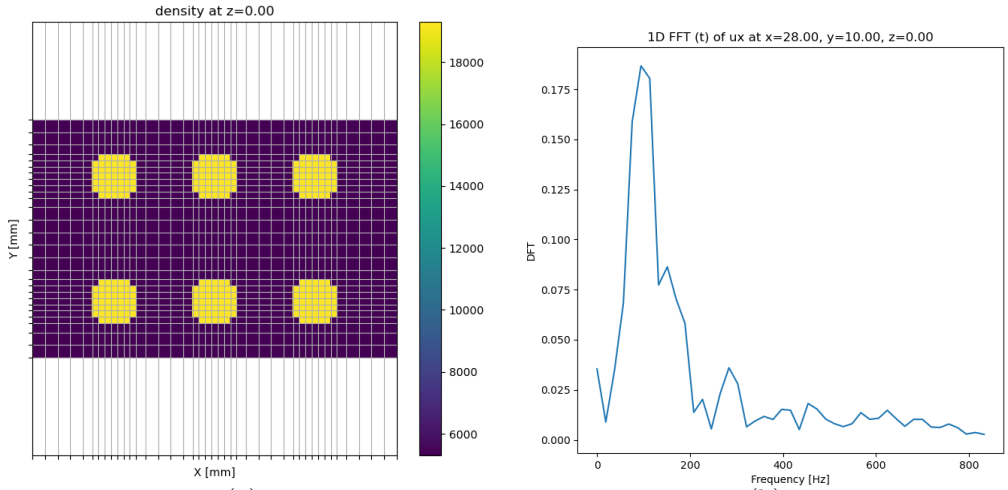


Figure 2.6: a) Non-uniform density plot overlaid with mesh grid. Substrate is GaAs and inclusions are Au.
b) Frequency analysis following simulation of 100Hz sinusoidal input.

may pose more significant bottlenecks than on other systems, lowering the best-case performance ceiling.

Results were more promising on the 4-core desktop CPU. The default/non-optimized solver was clearly slower than the other methods. Interestingly, multiprocessing, threading and Numba implementations performed almost identically. In fact, there was no discernible difference between each of the three high-performance solver methods. This system had the fastest RAM and largest CPU cache, being the newest processor of the three. It may be possible that all three implementations approached the limit of what is possible with the number of calculations and the fixed clock speed.

The result observed on the 6 core CPU is slightly more clear. The limited instructiopn set on this older CPU is a potential cause for the reduced benefit of Numba JIT compilation. As expected, the multiprocessing method has a clearer advantage over threading because of the high core count. However, the advantage is not significant

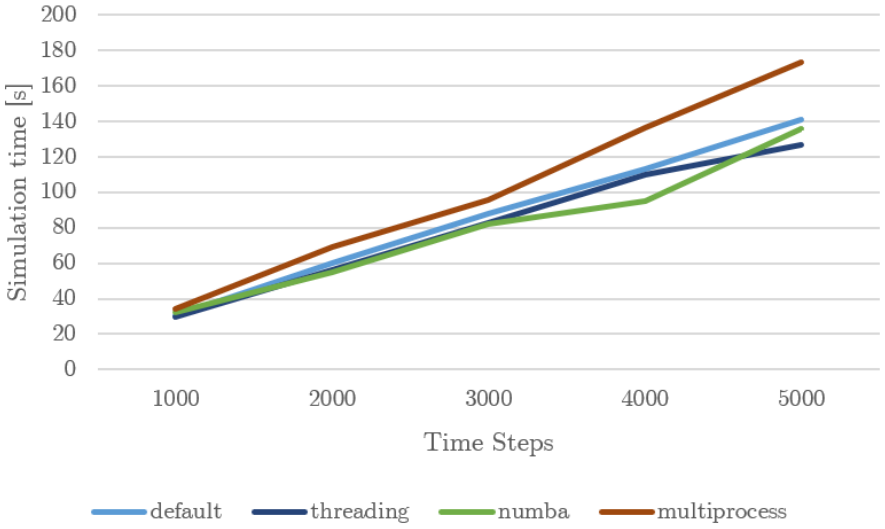


Figure 2.7: Run time comparison of various solver implementations for system 1 B.1

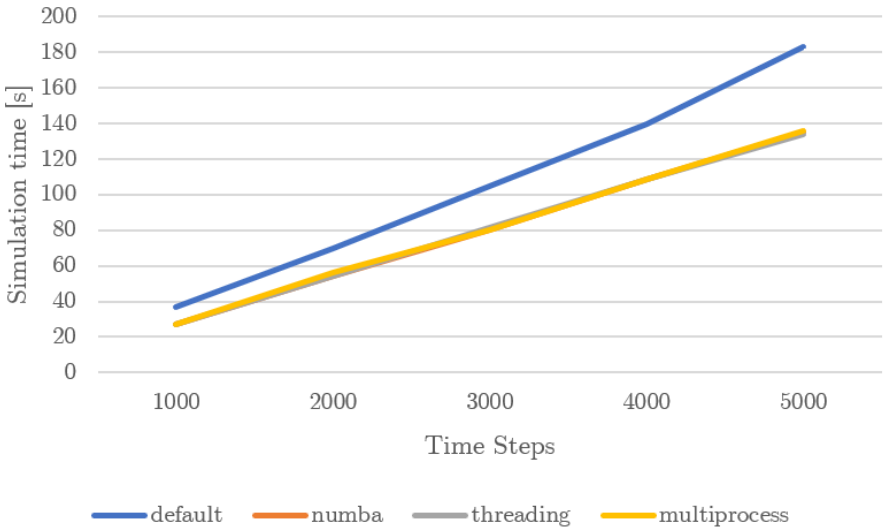


Figure 2.8: Run time comparison of various solver implementations for system 2 B.1

especially considering the extra design challenges.

Finally, the same test was run again using the Intel Distribution for Python. The average computation time per time step for each solver showed improvement on both desktop CPUs. Numba was not compatible with the Intel distribution on the older

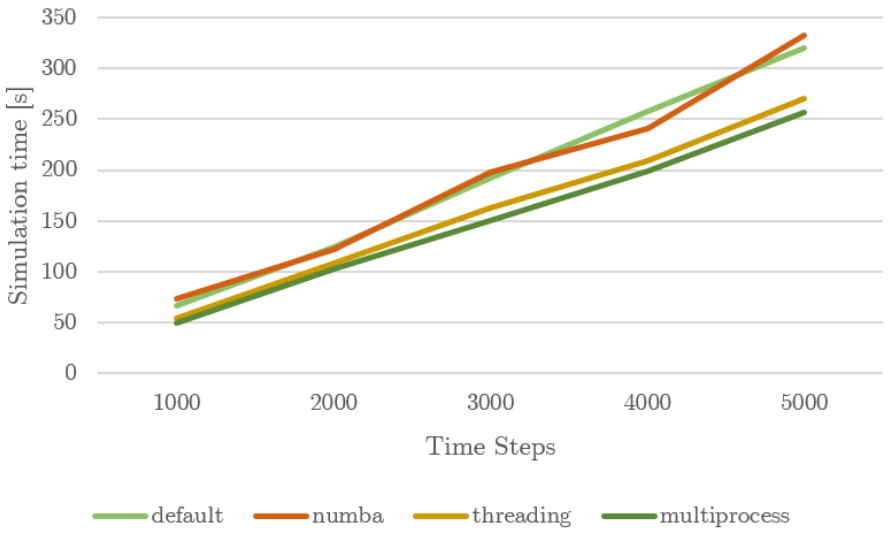


Figure 2.9: Run time comparison of various solver implementations for system 3 B.1

Table 2.3: Average time per loop iteration of each solver implementation on standard Python 3.8.2 and the Intel Distribution for Python (Python 3.7.7)

CPU	Solver	Time per step [s]	
		Python	Intel
Intel i7-6500U	Default	0.0293	0.0292
	Numba	0.0275	0.0290
	Threading	0.0276	0.0272
	Multiprocessing	0.0339	0.0294
AMD Ryzen 2200G	Default	0.0350	0.0162
	Numba	0.0271	0.0192
	Threading	0.0270	0.0138
	Multiprocessing	0.0271	0.0149
AMD Phenom II X5	Default	0.0641	0.0414
	Numba	0.0654	N/A
	Threading	0.0539	0.0374
	Multiprocessing	0.0506	0.0362

AMD CPU so that data was neglected from the results. The difference in performance is far better than other solver methods, with over 2x performance improvement in certain cases. The improvement was more apparent with the newer AMD CPU, which

was partially attributed to its more modern instruction set. SSE4.1 and SSE4.2, which are missing from the Phenom processor, specifically optimize floating point operations [11].

However, as shown in Table 2.3, it was not more effective for all test systems. The Intel distribution gained little to no improvement in performance on the i7-6500U system, despite nearly the same instruction set as the Ryzen CPU. Again, this may be a result of other CPU limitations in mobile hardware such as cache size. Dynamic analysis using the Intel vTune application revealed that the most significant hotspot of the test was in the ntoskrnl.exe application, which is involved in process and memory management and required by Python multiprocessing. For a variety of reasons, this CPU is unable to manage the extra overhead added by process management.

2.5 Conclusions & Future work

The major goals of this project were all addressed in this work to varying degrees of success. The primary objective involving non-uniform meshing was very successful and offers the most substantial improvement in application performance of all the methods tested. This feature provides the user with a more complete toolset to generate efficient mesh grids in a way that was previously not possible. It was built with future upgrades in mind, such as the addition of other mesh spacing functions. Currently only a linear function is used, but other functions, such as parabolic or exponential, could be added.

Several performance optimization techniques were designed and tested, with success in most cases. These implementations should be considered a proof of concept and not a best possible implementation. Future work should focus on inspecting each

method to find bottlenecks and base improvements on the findings. One specific area of potential research is to fundamentally change how simulations are performed. Currently this step is very 'pythonic', with explicit steps and function calls. This creates more readable code, but is inherently slow and forced each of the performance enhancement methods to rely on slow memory mapping or copying operations. It may be possible to eliminate the need for these steps by overhauling the main portion of the simulation. This would allow a JIT compiler such as Numba to optimize the the entire simulation loop rather than squeezing small gains out of individual subsections.

Further improvements include the addition of Bloch periodic boundary conditions to effectively allow the size of large simulation spaces to be reduced to a common unit cell. This would allow for the development of band gap simulations, enabling the results of *phonomena* to be compared against literature. Other boundary condition improvements can also be investigated, such as the inclusion of perfectly matched layers (PML) to better approximate energy absorption.

Significant effort was taken to ensure that this project was left in a state conducive to use and continued development by Dr. Stotz and his team. Low level design decisions are documented in a Software Description Document, available alongside the source code in the GitHub repository. The entire application, including all dependencies, are included in the same location, with Intel Distribution for Python releases available for Windows and standard Python 3.8 releases for Linux and Windows. This project was designed for use, and every effort was taken to ensure it accommodated that purpose.

Bibliography

[1] M. S. Kushwaha, P. Halevi, L. Dobrzynski, and B. Djafari-Rouhani. Acoustic band structure of periodic elastic composites. *Phys. Rev. Lett.*, 71:2022–2025, Sep 1993.

[2] Jiankun Huang, Yifan Liu, and Yaguang Li. Trees as large-scale natural phononic crystals: Simulation and experimental verification. *International Soil and Water Conservation Research*, 7(2):196 – 202, 2019.

[3] James A. H. Stotz, Rudolph Hay, and Paulo V. Santos. Coherent spin transport through dynamic quantum dots. *Nature Materials*, 4:585–588, 2005.

[4] Joseph Petrus. A computational and experimental study of surface acoustic waves in phononic crystals. Master's thesis, Queen's University, 2009.

[5] Marc Cameron. Implementing an fdtd simulation program for phononic crystals. Undergraduate thesis, Queen's University, 2018.

[6] Yanghua Wang. Frequencies of the ricker wavelet. *Geophysics*, 80(2):A31–A3, 3 2015.

[7] Intel Distribution for Python* 2020 Update 1. Technical report, Intel, 3 2020.

[8] Stefan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, March 2011.

[9] Gaël Varoquaux. Optimizing code.

[10] Riccardo Fazio, Alessandra Jannelli, and Santa Agreste. A finite difference method on non-uniform meshes for time-fractional advection–diffusion equations with a source term. *Physical Sciences and Earth Sciences*, 2018.

[11] Intel SSE4 Programming Reference. Technical report, Intel, 2007.

Appendix A

Statement of Work and Contributions

The work outlined in this report was a continuation of the undergraduate thesis done by Marc Cameron in 2018. His work consisted of a FDTD solver using uniform static meshing which served as the basis of this project. Inspiration was drawn from this project for calculation of stress and displacement matrices, as well as absorbing and and traction free boundary conditions. No work on this thesis was completed prior to September 2019.

Total word count: 4637

Appendix B

Technical Details

Figure B.1: Details of each test system used for performance analysis, specifically focusing on CPU differences.

System	Details	
1	CPU	Intel i7-6500U
	Cores	2 cores, 4 threads
	Clock speed	2.50 GHz
	SIMP instruction sets	SSE, SSE2, SSE3, SSE4.1, SSE4.2
	RAM	8 GB DDR3 1600 MHz
	OS	Windows 10 64-bit
2	CPU	AMD Ryzen 2200G
	Cores	4 cores, 4 threads
	Clock speed	3.50 GHz
	SIMP instruction sets	SSE, SSE2, SSE3, SSE4.1, SSE4.2, SSE4A
	RAM	8 GB DDR4 1066 MHz
	OS	Windows 10 64-bit
3	CPU	AMD Phenom II X6 1095T
	Cores	6 cores, 6 threads
	Clock speed	3.2 GHz
	SIMP instruction sets	SSE, SSE2, SSE3, SSE4A
	RAM	8 GB DDR3 1333 MHz
	OS	Windows 10 64-bit

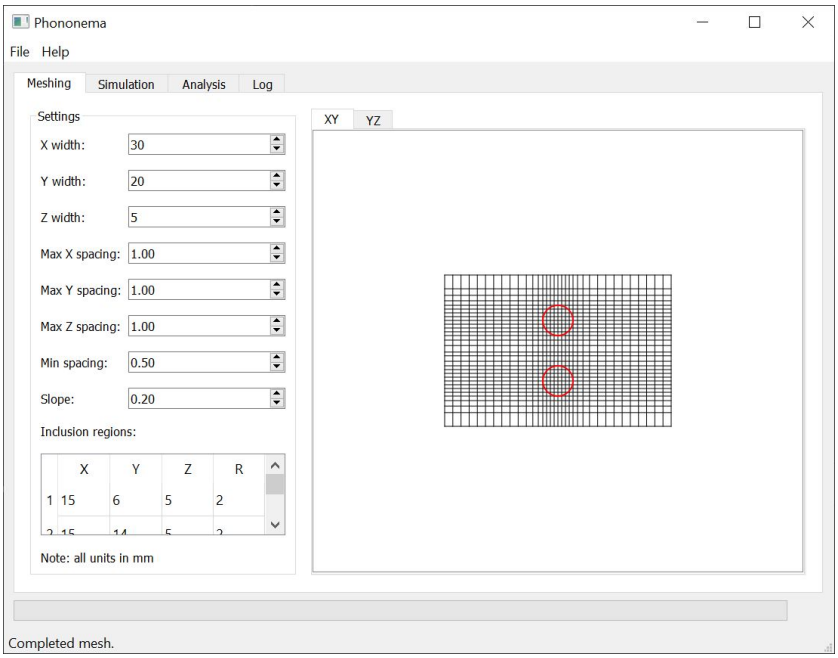


Figure B.2: Mesh generation page of the *phonomena* GUI.

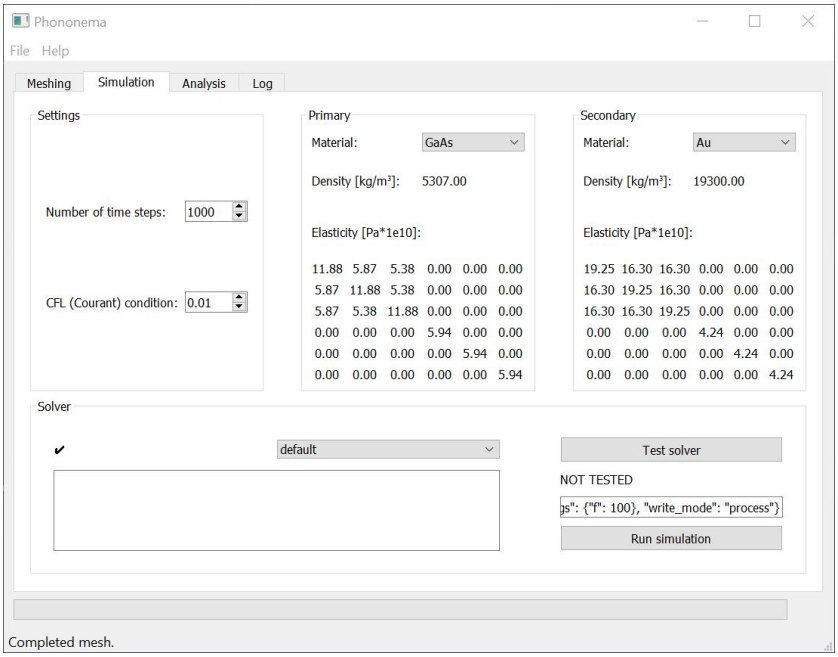


Figure B.3: GUI simulation tab displaying material properties and solver details.

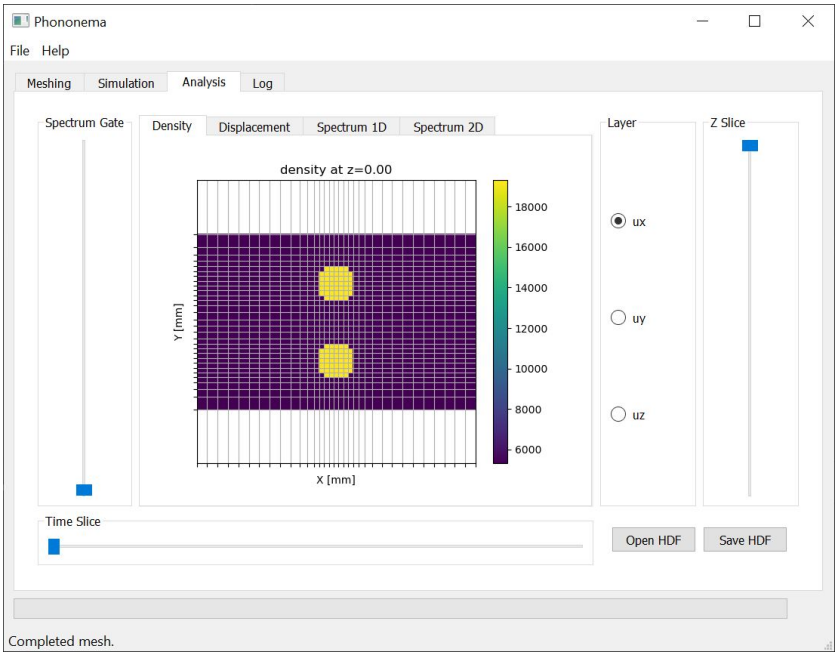


Figure B.4: Analysis portion of the GUI containing density, displacement and frequency decomposition.

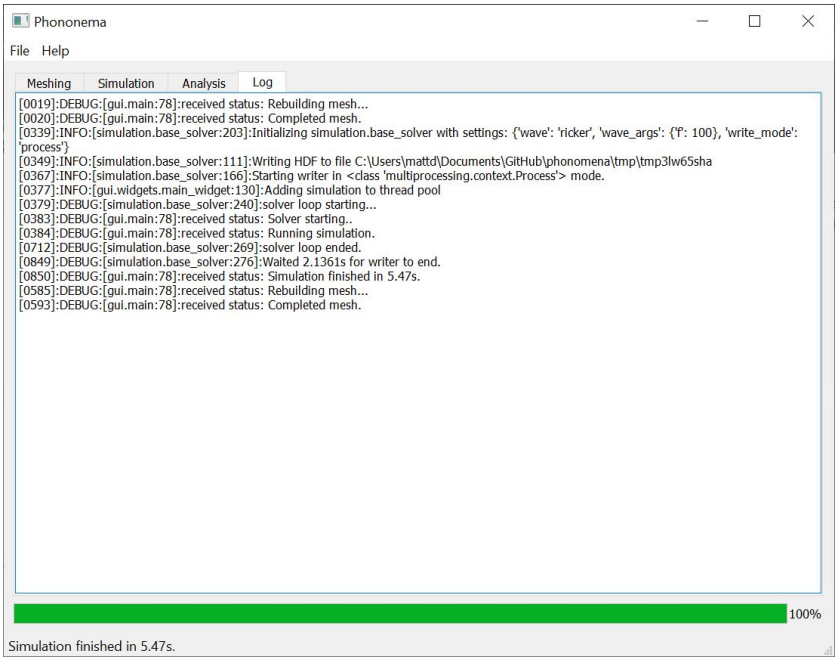


Figure B.5: GUI log tab used to capture and convey information throughout the application to the user.