**Phonomena Software Description Document**

**Release 2.0**

April 2020

Queen’s University

Matt Duke

Contents

[Overview 4](#_Toc38019567)

[Introduction 4](#_Toc38019568)

[Goals 4](#_Toc38019569)

[Folder Structure 4](#_Toc38019570)

[Simulation Module 5](#_Toc38019571)

[Grid Class 5](#_Toc38019572)

[Meshing 5](#_Toc38019573)

[Inclusion Regions 6](#_Toc38019574)

[Recording Data 6](#_Toc38019575)

[Material 6](#_Toc38019576)

[Analysis 6](#_Toc38019577)

[Graphical User Interface (GUI) 7](#_Toc38019578)

[Worker 7](#_Toc38019579)

[Logging 8](#_Toc38019580)

[Common 8](#_Toc38019581)

[Data File 8](#_Toc38019582)

[Solvers 9](#_Toc38019583)

[Base Solver 9](#_Toc38019584)

[Writer 9](#_Toc38019585)

[Solver Default 10](#_Toc38019586)

[Solver Threading 11](#_Toc38019587)

[Solver Numba 11](#_Toc38019588)

[Solver Remote 11](#_Toc38019589)

[Development 11](#_Toc38019590)

[Testing 11](#_Toc38019591)

[Creating Solvers 11](#_Toc38019592)

[Building Application 11](#_Toc38019593)

[Windows 11](#_Toc38019594)

[Linux 11](#_Toc38019595)

# Overview

## Introduction

Phonomena is a software tool used to simulate acoustic wave propagation through phononic crystals. This type of simulation is useful in designing solid-state filters and waveguides for acoustic waves.

## Goals

The primary goals of this application are listed, in no particular order, as follows:

1. Simple to use with a robust feature set
2. Basic visualization of simulation and results
3. Ability to save data for future analysis
4. Easily extensible and well documented
5. Accurate results and performance enhancements

The inclusion of a user interface and build system was meant to address goals 1 and 2. The GUI acts as a abstraction layer over the feature set of the main application and is relatively unforgiving in the sense that few constraints are placed on user inputs. To promote ease of use, the program can be bundled as a single portable folder with all necessary dependencies including the entire Python environment. Formal releases for Windows and Linux will be released on the GitHub repository which can be downloaded and run by any user without any additional setup

Data is saved to disk using the hierarchical data format. This allows the output of the simulation to be analyzed by other software programs at a later time. This file is created during every run of the simulation but only saved when requested.

The goal of extensibility and documentation is partially accomplished by the creation of this document. However, this was built into the software by allowing it to be run as a GUI application or as a Python module with can be imported into other applications.

## Folder Structure

The main branch of the phenomena repository is organized in a way to allow for the application to be easily extended and used as a standalone application or as an importable module. The structure is as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| build | | | Temporary folder used by the build script. |
|  | dist | | Location of distribution folder. This folder can be zipped and released as a standalone application |
|  | work | | Directory used during the build process |
| data | | | Contains static files that will be included with the bundled application |
| docs | | | Documents and user guides to assist general users and continued development of the project |
| phonomena | | | Application folder |
|  | gui | | Python module containing user interface application. Acts as a frontend for the simulation application |
|  |  | widgets | Separate file for each tab on the main window of the application |
|  | simulation | | Folder containing files needed for FDTD simulation |
|  |  | solvers | The folder is scanned and will attempt to import each python file |
| tests | | | Scripts used to test performance of the application and different solvers |
|  | results | | Log of test results |
| .gitignore | | | Specify files to ignore on git commits |
| build.py | | | Build script to bundle the application using PyInstaller |
| README.md | | | Repository description file |
| setup.py | | | Installs necessary Python packages using PIP. |

# Simulation Module

The simulation module includes files used to setup, run and analyze the results of the FDTD simulation.

## Grid Class

The Grid class stores and calculates the mesh parameters based on user inputs. The main feature of Grid is the buildMesh function which determines mesh spacing based on the size and density of the mesh and the location of inclusion regions. The algorithm used to generate this mesh is broken down into 5 functions within the buildMesh function that can be used for the X or Y axis.

The grid class contains the stress and displacement matrices as well as staggered and full half grid dimensions. This class is iteratively updated when passed to the

### Meshing

The start and end of the mesh grid is set first as fixed locations. Within those constraints, the grid lines near each inclusion region is added using the specified function, which is linear by default. The inclusion region itself is then filled based on a spacing that is uniformly divisible by the mesh radius and is close to the minimum grid spacing set by the user. Because these two processes may interfere with each other, lines are removed that are closer than the minimum grid spacing. Finally, the remaining area of the grid is filled using a spacing that is close to the maximum grid spacing and evenly divisible by the region that is being filled.

This algorithm is likely not the most efficient and can create unwanted grid geometry for closely spaced or overlapping inclusion regions. However, it is visually intuitive and for all grid geometries tested, the calculation delay was insignificant.

### Inclusion Regions

Inclusion regions are stored using a named set of parameters including the XY coordinates of the center of the target, the depth of the inclusion and the radius of the circle. Helper functions are included to clear or add inclusion regions, as well as to identify each point in the mesh that is within an inclusion.

### Recording Data

In order to reduce complexity, the Grid class is not inherently thread safe. Passing a Grid object to a different thread for writing via a Queue object or similar can result in data corruption or mismatched entries, because the Grid object will be passed by reference. In other words, modifying the Grid object in the main thread will also modify the object in the secondary thread. This is bypassed using the freezeData function, which copies and returns the relevant Grid class data in a way that can not be overwritten. See the BaseSolver class documentation for an example of its usage. Other methods could be used instead, such as copying the entire Grid class, but this method was found to be simple and efficient, and works with multiprocessing of threading style queues.

## Material

The Material class is used to store material properties and perform basic calculations based on these properties such as simulation timestep. It heavily relies on the Grid class, which must be passed to the Material object using the init function. When the grid object is passed to the material class, it is effectively passed by reference, meaning that the Grid object stored in the Material class points to the original Grid object. Any changes made to one will therefore affect the other, as they act as the same object.

## Analysis

In order to reduce code duplication, commonly used functions for analysis were bundled as a single file. The goal was to standardize how analysis was performed throughout the application for consistent and comparable results. This script is used by the GUI widgets or can be used by external scripts.

Features include 1D/2D discrete Fourier transforms (DFT) of displacement data contained in an HDF file using a fast Fourier transform (FFT). This function uses a Hanning window to minimize the effects of leakage and returns vectors for the x-axis grid (for the 2D case) and the frequency bins. An additional function was created to trim the data such that all frequencies above which there is negligible spectral response are removed from the dataset. This is helpful for visualization of the DFT.

# Graphical User Interface (GUI)

The user interface is built using PyQt5, which is a set of Python bindings around the popular Qt library. This library was chosen for its popularity, documentation and the fact that development and support will continue for the foreseeable future.

## Worker

Long running processes, such as running the simulator or building the mesh would cause noticeable delays GUI and cause the program to stop responding if run from the main thread. Instead, these functions are run as workers using a subclass of the QtRunnable object. This allows the function to be run as a separate thread by placing it in a thread pool (QtThreadPool object)

The worker class automatically sends synchronization objects to the funning function as keyword arguments (kwargs). These objects are outlined in the worker signals class.

|  |
| --- |
| **class** **WorkerSignals**(QtCore.QObject):  '''  Defines the signals available from a running worker thread.  Supported signals are:  Finished: No data - function finished  Error: `tuple` (exctype, value, traceback.format\_exc() )  Result: `object` data returned from processing, anything  Progress: `int` indicating % progress  Quit: No data - force close thread  '''  finished = pyqtSignal()  error = pyqtSignal(tuple)  warning = pyqtSignal(tuple)  result = pyqtSignal(object)  progress = pyqtSignal(int)  status = pyqtSignal(str)  quit = pyqtSignal() |

When a worker is created, callback functions can be passed which allow for connections between the main window and worker thread. The main window includes several callback functions for writing status messages, updating the progress bar and dealing with errors. These functions are bundled and should always be passed to the worker class. Custom callbacks can then be connected to the relevant signal object. An example of creating a worker, passing it to the thread pool and running it is:

|  |
| --- |
| self.gui\_update\_pool.clear() *# Optional*  self.gui\_update\_pool.waitForDone(msecs=1000)  worker = Worker(fn=build, callback\_fns=self.window.callback\_fns) *# Create worker*  worker.signals.finished.connect(drawGrids) *# the drawGrids function will be run when the thread sends the finished signal*  self.gui\_update\_pool.start(worker) *# Add worker to thread pool* |

Additional args and kwargs passed to the worker initializer will be forwarded to the target function so the function should have a method to receive the necessary arguments.

## Logging

The log window is used to show debug information from each of the modules. It relies on Python’s built in logging module, which allows any function to use the global logging system.

The log widget extends the logging.Handler class, which allows it to connect to the global logging system. All files that wish to use the logger should import and initialize a logger object at the top of the file.

**from** **logging** **import** getLogger

logger = getLogger(\_\_name\_\_)

This logger will use the same settings as the root logger, which is configured by a function in the common file. The name variable will be unique to each file, allowing for the source of the log string to be identifiable.

# Common

The purpose of the common file is to simplify functions that should be available to any module to avoid duplicate code. It includes functions to save and load data from the JSON file format, configure logging, change the temporary file directory and import solvers. These features are required and used when using the application with the user interface but are also accessible when importing the application to other scripts, such as the test scripts.

The common module also provides a simplistic shared memory approach to sharing variables between different parts of the program. Global variables can be shared between files and accessed through common, by importing the common module in each file.

## Data File

Project settings can be saved to and loaded from a JSON configuration file. This file contains mesh details, geometry, material properties and simulation values. It is not strictly necessary when using the application as an API and can be replaced by hardcoded data (as shown in the base\_solver.TestDefaults class), but it is required when using the GUI, which loads initial settings from the defaults.json file in the data folder.

Data is saved from and loaded to the global Grid and Material objects contained in common using the loadSettings and saveSettings functions. These functions can be modified to add parameters but must be modified together such that they share the same data layout.

An additional feature of the JSON data file is to store material properties. New materials can be added to the application by simply modifying the default data file or loading a new one.

# Solvers

The solver is the most significant result of this work and has the largest impact of program performance and accuracy.

## Base Solver

The BaseSolver class is designed to be inherited as a parent class, then modified to add functionality. Several examples have been included in this version of the code. It aims to simplify development using a common framework that all solvers use. This means that they will all share the same variables and functions with the option to overwrite that and extend them left to each solver.

For example, by inheriting the BaseSolver object, each solver will have a test function, which runs a short simulation on a standardized and common grid. The child solver may wish to add functionality and perform additional tests but should not entirely replace this test. The BaseSolver also includes default configuration parameters that are required by all solvers.

Solvers are designed to be run in the following sequence. On import, the constructor is called for each solver, instantiating the name, description, logger configuration parameters and a unique temporary file for the class. To run a simulation, the solver is setup using the init function, where the grid and material objects are passed with the number of time steps to the solver. These are intentionally copied and stored within the solver class so that are frozen and will not be linked the original grid and material objects. Finally, the simulation can be started using the run function, which is compatible with the GUI WorkerSignals class and will publish notifications to the GUI if running in this way.

Each solver creates a unique file identifier for the HDF file that is persistent until the application closes. This file is created in a folder specified by the environment variable TEMPDIR, which is simply the tmp folder in the application root directory by default. This file should only be used temporarily, as it will be overwritten on every simulation run. It can be saved by simply copying the file.

### Writer

One of the significant inclusions of this version is a method of saving data to a file for later analysis. This is performed by the Writer object, which, as the name suggests, writes data to the HDF file. This is done in two steps, similar to the BaseSolver class. First, the datasets are created, and persistent data is written in the init function. When the Writer is started, it reads data from an internal queue (populated by the BaseSolver run function) and adds it to the HDF datasets. Placing a None object in the queue notifies the writer that the simulation has finished and it can exit.

Writing data to a file is an inherently slow operation. This class is designed to act independently of the main loop to avoid major IO delays. This can be performed in two modes, multiprocessing or threading, specified by the solver parameter ‘writer\_mode’. Threading runs the writer as a Python thread which runs semi-concurrently with the main loop by taking advantage of idle time. However, due to Python’s Global Interpreter Lock (GIL), it can only use a single CPU core, and the simulation loop is delayed by file operations and drive speed. Multiprocessing mode runs the writer as an independent Python process, using a separate CPU core and placing no limitations on the main loop speed. Multiprocessing can be about 10x faster and should always be used on hardware with at least 2 CPU cores.

## Solver Default

The simplest is the default solver which acts only as a wrapper for the base solver and adds no functionality other than a name and description line. It is included as an example of BaseSolver inheritance.

|  |
| --- |
| class Solver(base\_solver.BaseSolver):  def \_\_init\_\_(self):  super().\_\_init\_\_(logger)  self.name = "default"  self.description = "<p>Default solver with no HW acceleration.</p>"  # Combine global and default class cfg object  self.cfg = {\*\*cfg, \*\*self.cfg}  def init(self, grid, material, steps):  *# add features here*  super().init(grid, material, steps)  def run(self, \*args, \*\*kwargs):  *# add features here*  super().run(\*args, \*\*kwargs)  def test(self, \*args, \*\*kwargs):  *# add features here*  super().test(\*args, \*\*kwargs) |

As shown in the example above, some functions are overwritten by the default solver as a demonstration of how to add features while reusing the BaseClass functions. More complex examples are shown in the other solvers.

## Solver Threading

The nature of this simulation is not conducive to parallelization because each time step depends on the one before. However, the order of certain calculations, such each stress tensor component, are independent of each other and it is possible to run these calculations in parallel.

This solver starts a pool of 6 worker threads (one for each stress tensor component) which wait to be given a function to process. Each calculation is then passed to the pool of workers, computed in the thread and returned.

However, due to the GIL, these processes are not concurrent. Numpy operations are highly optimized with little idle time to take advantage, so this method will likely not improve performance by a significant amount and may worsen due to larger overhead.

## Solver Numba

This solver uses the Numba Python library to easily translate Python functions into machine code. It uses the LLVM compiler library to act as a just-in-time (JIT) compiler to accelerate repeated function calls. It has many additional features and is commonly used for scientific computing in Python. The implementation used here is the simplest, but likely not the fastest and future improvements are highly possible.

However, performance was seen to be improved using the Numba solver over the default, particularly for XXX. This comes with the caveat that the initial function compilation adds several seconds of runtime, but once this step is completed the code will perform better. The cache option is enabled by default for this solver, making compilation persistent across application runs.

## Solver Remote

Very large mesh sizes and long simulation runs could potentially take hours to complete, particularly on low-powered devices. This remote solver allows for computation to be performed on a remote server.

# Development

## Testing

Test scripts have been included in the tests folder that were used during the thesis accompanying this work. These can be used as starting points for other test scripts, or for API tools that use all the functionality while bypassing the GUI.

## Creating Solvers

New solvers can be created with additional functionality by inheriting the BaseSolver class. Extra functionality can be added to the init and run functions if needed, but should not replace the functionality provided by

Dependencies used by solver modules are hidden from the build script because they are importable at runtime. If new solvers are developed with added dependencies, they must be manually added to a list of hidden imports in the build script.

## Building Application

The application is bundled with the PyInstaller Python module which creates a single folder including the entire Python runtime environment, all necessary dependencies and the application files. This folder is portable and does not require any extra setup on the user’s computer. This process is performed using the build.py script. It requires the PyInstaller package which is not included as a required library in the setup.py file so it must be downloaded separately. This script also creates a unique build number and build version which are bundled in the application to allow for easier debugging and comparison between different versions.

## Windows

Python 3.8.2 was used to build the Windows release. New releases can be built by simply running the build script with a compatible version of Python (>=3.7).

## Linux

Testing of this application in a Linux environment was exclusively performed on a Debian 10 virtual machine using the Windows Subsystem for Linux component. Running against real Linux operating systems or OS’s other than Debian may reveal bugs that were not found during testing of this release.

To replicate the Debian image, the following steps are required:

1. Install Debian for Windows Subsystem for Linux from the Windows store
2. Update the OS using sudo apt-get update && sudo apt-get upgrade
3. As of the time of writing, Python 3.7.3 was the only release that supported all dependencies on Debian 10. It was installed using: sudo apt-get install python3.7 python3-pip python3-pyqt5 followed by python3.7 -m pip install --upgrade pip
4. The windows C: drive is automatically mounted to /mnt/c. If you are accessing the repository from this drive, the mounted folder should be owned by the Debian user account as this is required by the build script. It can be achieved by creating the file /etc/wsl.conf and adding the following text:

[automount]

enabled = true

options = "metadata"

mountFsTab = false

1. Dependencies were by running setup.py with Python 3.7 as the standard user.
2. The Xming server was installed on the Windows host from [sourceforge.net/projects/xming](https://sourceforge.net/projects/xming/). This acts as a GUI interpreter between Windows and the WSL environment.
3. To allow GUI applications to run from the WSL container, an environment variable was added using the following command: echo 'export DISPLAY=:0' | sudo tee -a ~/.bashrc
4. If the GUI fails to start when running \_\_main\_\_.py due to an error relating to the libQT5Core.so module, the following command can be used as a workaround: sudo strip --remove-section=.note.ABI-tag /usr/lib/x86\_64-linux-gnu/libQt5Core.so.5. More details are available here: <https://askubuntu.com/questions/1034313/ubuntu-18-4-libqt5core-so-5-cannot-open-shared-object-file-no-such-file-or-dir>.