BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

ULTRASOUND SIMULATION IN PYTHON
ULTRAZVUKOVÁ SIMULACE V PYTHONU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
DAVID ČERNÝ

SUPERVISOR
VEDOUCÍ PRÁCE
doc. Ing. JIŘÍ JAROŠ, Ph.D.

BRNO 2022

Department of Computer Systems (DCSY) Academic year 2021/2022

Bachelor's Thesis Specification

24884

Student:	**Černý David**
Programme:	Information Technology
Title:	**Ultrasound Simulation in Python**
Category:	Software Engineering

Assignment:

1. Familiarize yourself with the k-Wave software toolbox for simulation of ultrasound wave propagation.
2. Study the features of the Python language related to the high performance computing.
3. Design a method for transforming simulation codes written in Matlab to Python considering the performance as a primary objective.
4. Implement the designed solution.
5. Evaluate the performance of the developed solution on a standard set of test tasks.
6. Discuss the impact and contribution of your work to the future development of k-Wave.

Recommended literature:

- According to supervisor's advice.

Requirements for the first semester:

- Items 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Jaroš Jiří, doc. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	October 29, 2021

Abstract

k-Wave is a MATLAB toolbox for the simulation of sound wave propagation. The aim of this thesis is to re-implement a subset of k-Wave in Python while focusing on computational performance. The second goal is to develop a set of guidelines for transforming MATLAB source code to Python that could aid in further development. The thesis first summarises core features of the k-Wave toolbox, explores available technologies for high performance computing in Python, and highlights the most important aspects of transforming MATLAB source codes to Python. The second part of the thesis discusses architecture, testing and benchmarking of the Python implementation. The result of this thesis is a Python implementation of the three-dimensional sound propagation simulation compatible with k-Wave. The new implementation improves the structure of the original toolbox while providing performance comparable to the original k-Wave. In some instances, the performance of the new implementation surpasses the original implementation.

Abstrakt

k-Wave je MATLAB nástroj pro simulaci šíření zvukových vln. Cílem této práce je reimplementovat část nástroje k-Wave v jazyce Python se zaměřením na výpočetní výkon. Druhým cílem je formulace sady doporučení pro transformaci zdrojových kódu z jazyka MATLAB do jazyka Python, které by mohly přispět při dalším vývoji. Tato práce nejprve shrnuje klíčové funkce nástroje k-Wave, zkoumá technologie pro vysoce výkonné výpočty dostupné v jazyce Python a zdůrazňuje nejzásadnější aspekty transformace zdrojových kódů z jazyka MATLAB do jazyka Python. Druhá část práce se zabývá architekturou, testováním a měřením výkonu výsledné Python implementace. Výsledkem této práce je implementace trojrozměrné simulace šíření zvuku, která je kompatibilní s k-Wave. Nová implementace vylepšuje strukturu původního nástroje a poskytuje výkon srovnatelný s původním nástrojem, v určitých případech výkon původního balíku převyšuje.

Keywords

k-Wave, simulation, optimization, OOP, NumPy, Python, MATLAB

Klíčová slova

k-Wave, simulace, optimalizace, OOP, NumPy, Python, MATLAB

Reference

ČERNÝ, David. *Ultrasound Simulation in Python.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Jiří Jaroš, Ph.D.

Rozšířený abstrakt

k-Wave je simulační nástroj pro prostředí MATLAB, který umožňuje simulaci šíření ultrazvuku v jednorozměrném, dvojrozměrném i trojrozměrném prostoru. Ultrazvuková simulace je často využívána na poli medicíny. Protože ultrazvukové simulace většího rozsahu jsou výpočetně náročné, využívá k-Wave externí akcelerátory jako např. k-Wave-Fluid-OMP, které umožňují urychlit proces simulace jak na procesoru, tak pomocí grafických karet. K vývoji těchto akcelerátorů se povětšinou používají poměrně nízkoúrovňové jazyky jako C++, které ale nejsou pro všechny uživatele k-Wave dostatečně přístupné. Další nevýhodou nízkoúrovňových jazyků je dlouhá doba vývoje nových funkcí simulátoru.

Kvůli popularitě jazyka Python ve vědecké sféře je i v komunitě uživatelů nástroje k-Wave poptávána verze tohoto nástroje v jazyce Python. Díky vysoké úrovni abstrakce jazyka Python by také bylo umožněno jeho nasazení při rychlém prototypování nových funkcí simulačního nástroje k-Wave. Dalším přínosem verze k-Wave simulátoru pro jazyk Python by byla možnost propojení s velkou škálou jiných populárních nástrojů a knihoven, které byly pro jazyk Python vytvořeny.

Cílem této práce je implementovat verzi simulačního nástroje k-Wave v jazyce Python a vytvořit sadu doporučení pro konverzi zdrojových kódů z jazyka MATLAB do jazyka Python, to vše s důrazem na výpočetní výkon.

Práce nejprve shrnuje základní pojmy a koncepty nástroje k-Wave a popisuje princip jeho fungování, zejména s ohledem na strukturu simulačních dat, komunikaci s externími akcelerátory a funkcemi nástroje, které jsou předmětem implementace. Následně jsou popsány nejzásadnější rozdíly mezi jazyky MATLAB a Python z hlediska konverze zdrojových kódů. Mezi tyto problematiky patří indexování, datové struktury, aritmetické operace, fourierovy transformace a vizualizace dat. V souvislosti s těmito otázkami jsou představeny výpočetní a vizualizační knihovny dostupné v jazyce Python. Zejména je implementace v jazyce Python založena na knihovnách NumPy, numexpr a pyFFTW, které umožňují efektivní vědecké výpočty. Dále jsou vyjmenovány některé zásadní návrhové vzory, které byly použity při implementaci pro zlepšení struktury a modularity výsledného řešení.

Praktická část práce popisuje strukturu výsledné implementace, její návrhová rozhodnutí a odlišnosti od původní implementace v jazyce MATLAB. Nová implementace byla otestována pomocí nástroje kWaveTester, který je součástí referenčního balíku k-Wave.

Nástroj kWaveTester slouží k automatickému generování testovacích dat a srovnávání výsledků simulace akcelerátorů s referenčními výsledky simulace. Práce představuje několik vzorových testovacích příkladů, na kterých byla implementace testována, a zhodnocuje jejich výsledky. Celá implementace je nakonec zhodnocena z hlediska výkonnosti pomocí optimalizačních nástrojů, výkonnost je také porovnána s původní implementací v jazyce MATLAB. Během optimalizace je zhodnocena i práce s pamětí a jsou představeny problémy, které neefektivní přístup k paměti způsobuje.

Po zhodnocení stávajícího stavu jsou navrženy další potenciální způsoby optimalizace včetně příkladů typicky řešených optimalizačních problémů.

Výsledkem práce je shrnutí důležitých aspektů převodu zdrojových kódů z jazyka MATLAB do jazyka Python se zameřením na výpočení výkon. Hlavním výstupem je verze balíku k-Wave v jazyce Python, která obsahuje všechny nezbytné nástroje pro trojrozměrnou ultrazvukovou simulaci. Nová implementace vylepšuje strukturu původního nástroje a také poskytuje výkon srovnatelný s původním balíkem k-Wave, v určitých případech výkon původního balíku převyšuje.

Ultrasound Simulation in Python

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Ing. Jiří Jaroš, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
David Černý
May 11, 2022

Acknowledgements

I want to thank my supervisor, doc. Ing. Jiří Jaroš, Ph.D., for his guidance and willingness to help during the writing of this thesis. This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90140).

Contents

Chapter 1

Introduction

Ultrasound simulation, or more generally the simulation of sound wave propagation, is a type of continuous physics simulation. It models changes in pressure and velocity inside a media over a given span of time using a set of physics equations, the media having predefined density and other characteristics. Common applications include the study of acoustics, modeling of human tissue in biomedicine, and other applications, where propagation of sound in materials is important. One of the many simulation programs available is k-Wave [21], a toolbox written for the environment. It includes configurable tools for sound propagation simulations in up to 3 dimensions. Since both the time and memory complexity of such simulations sharply increase with the size of the simulation medium, optimization is essential to make large-scale simulations feasible. Due to this, k-Wave relies on optimized C++ and CUDA accelerators to provide the necessary performance for larger tasks.

C++ and CUDA, while very fast, require a higher level of programming expertise than would generally be expected of an average user of k-Wave. The time and effort needed to develop new features in said languages is also significantly increased by their verbosity.

Ideally, the selected implementation language should be abstract enough to speed up development time and provide good performance at the same time. While these two requirements go against each other to some degree, it is possible to find a compromise. The Python programming language presents a good candidate for such a compromise. It is sufficiently abstract and can also benefit from a large number of performant computation libraries. Because Python is also increasingly more popular in the scientific field, a Python version of k-Wave is in demand from the k-Wave community.

This thesis aims to implement a subset of the k-Wave toolbox in Python while focusing on the computational performance of the designed solution. The second objective is to create a set of guidelines for converting MATLAB source codes to Python based on the experience gained during the implementation and to discuss techniques for writing performant Python programs.

Chapter 2 introduces the structure and algorithms of k-Wave, which will be the focus of the final implementation. Chapter 3 discusses the conversion of MATLAB source codes to Python, emphasizing the most important differences and performance. Libraries and design patterns used during the implementation are also summarized. Chapter 4 describes the structure, features, and implementation k-Wave-Python. Chapter 5 describes the testing the new implementation, explores the effects of various optimizations on the performance of the simulator, avenues for further development are also explored. The thesis concludes in chapter 6 with a summary.

Chapter 2

A brief overview of k-Wave

k-Wave, as previously described, is a simulation toolbox for MATLAB[1]. Created by Bradley E. Treeby and Benjamin T. Cox in 2010 [21], the project has since been continuously under development. Extensions were later written in C++ and CUDA to accelerate performance-critical parts of the toolbox on both the CPU and the GPU. This chapter will outline the general structure of the MATLAB implementation of the toolbox, its most important features, and architectural decisions.

2.1 Structure of the toolbox

k-Wave consists of a set of core modules that contain the implementation of the k-space first-order simulation algorithms [20, p. 26]. All simulations can be performed in 1, 2, and 3 dimensions, the 3-dimensional variant being the most common in real-world applications such as low-intensity ultrasound neurostimulation [11] or high-intensity focused ultrasound tumor ablation [15]. Variants of the core algorithms are executed using the functions `kspaceFirstOrder1D`, `kspaceFirstOrder2D`, and `kspaceFirstOrder3D` for each number of dimensions respectively. Apart from the core modules, the toolbox also includes many auxiliary modules that handle the initialization and preparation of simulation data, visualization, data recording, and testing.

2.2 Simulation data

Multiple data structures, global variables, and flags serve as input data for initializing and running the simulation. Simulation input data is divided into four groups: the `kGrid`, the `Medium`, the `Source` and the `Sensor`. These four groups are passed to the simulation function as data structures together with additional optional parameters that are described in the k-Wave manual [20, p. 68]. This section summarizes contents of all input data groups and the PML (Perfectly Matched Layer).

kGrid

The kGrid contains variables that define the simulation time and dimensions of the simulation medium. The variables `Nx`, `Ny`, `Nz` define the discrete number of grid points in each

[1]https://www.mathworks.com/products/matlab.html

cardinal direction and dx, dy, dz define the physical spacing of grid points in the respective direction. The variables Nt and dt set the number of simulation time steps and their length.

Medium

The Medium describes the physical medium in which sound waves propagate, the most important value being the sound_speed, which defines the speed of sound propagation in the medium. The field density describes the density of the medium. In case the simulation is non-linear and/or absorbing, additional coefficients are also included.

The medium can be either homogenous, or heterogenous. A homogenous medium only contains one scalar value per field that is used for the entire domain, a heterogenous medium defines each field as an array, that contains a value for every grid point.

```
% ... Preceding initialization steps
Nx = 64;
Ny = 64;

medium.sound_speed = 1700 * ones(Nx, Ny);
medium.density = 800 * ones(Nx, Ny);

medium.sound_speed(:, 1:Ny) = 2000;
medium.density(Nx/2:Nx) = 500;

% ... Following initialization steps
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```

Listing 2.1: Initialization of a heterogenous medium in the k-Wave toolbox

Listing 2.1 shows the initialization of a simulation with a heterogenous medium [2]. Both the medium.sound_speed and medium.density are defined as arrays with values for pressure and density for each grid point. Following the allocation of arrays, slices of values in both the sound speed array and the density array are changed. This allows the simulation of sound propagation through a medium composed of different materials.

Source

The Source defines the sources which dictate where the sound wave originates from in the simulation medium. There are two types of sources: *initial pressure sources* and *time-varying sources*. Initial pressure sources only inject pressure into the simulation in the beginning (as defined by the variable p0), whereas time-varying sources continuously add pressure (or velocity) at grid points specified by a mask.

[2]Similar to an example from k-Wave: examples/example_ipv_heterogenous_medium.m

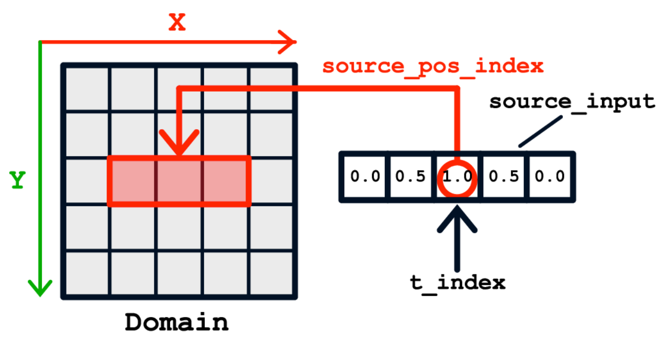


Figure 2.1: Time-varying source

Figure 2.1 represents the function of a time-varying source. The `source_pos_index` points to the grid points of a domain (in this case, a two-dimensional domain), that contain the time-varying source. After each step of the simulation, the source fetches the current source value pointed to by `t_index` (index of the current simulation step) from the `source_input` array. This value is then either set or added to the target grid cells, depending on the simulation settings. The `source_input` and `source_pos_index` have the prefix `p_` and `u_` for pressure and velocity respectively. The target domain depends on the source type. The `p` array (containing current pressure) is used for pressure sources and the `ux_sgx`, `uy_sgy` and `uz_sgz` arrays are used for injection of particle velocity for each cardinal direction.

```
% ... Preceding initialization steps

Nx = 32;
Ny = 32;
source.p0 = zeros(Nx, Ny);
source.p0(Nx/2-2:Nx/2+2, Ny/2-2:Ny/2+2) = 5;

% ... Following initialization steps
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```

Listing 2.2: Creation of an initial pressure source in the k-Wave toolbox

Listing 2.2 shows the initialization of an initial pressure source for a two-dimensional simulation. The `p0` array containing initial pressure is first initialized zeroed. Afterwards, a cube of initial pressure measuring 2x2 grid points with the value of 5 pascals is placed in the center of the medium. The initial pressure is copied to the `p` pressure array during the first time step of the simulation, then the initial pressure then propagates.

Sensor

Changes in pressure and particle velocity during simulation are recorded using Sensors. The grid points where measurements are taken can be set using either a `binary mask` or `cuboid corners`. With boolean masks, the presence of a sensor at a given grid point is defined by the corresponding binary value (`0` or `1`) in the sensor mask. This allows for more granular control over the shape of the sensors but requires more memory to store values for each coordinate. Cuboid corners, on the other hand, only require the coordinates of two opposing points of a geometric shape to describe the mask. In 3D, the two opposing coordinates form a cuboid, a rectangle in 2D, and a line in 1D [20, p. 36]. The `record` field contains the list of all measurements to be recorded – this includes pressure, particle velocity, and their various aggregations like the maximum, minimum, or final recorded value.

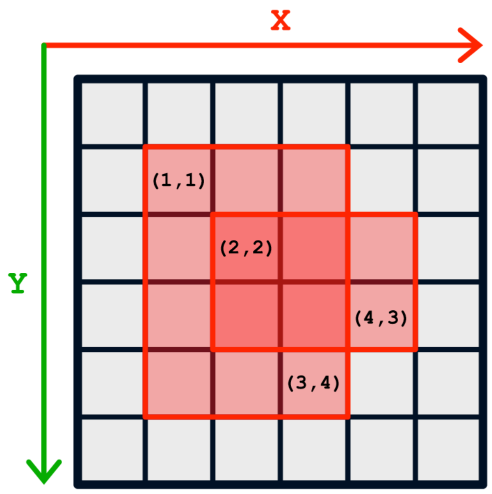


Figure 2.2: Cuboid corners mask

Figure 2.2 shows a two-dimensional grid with two overlapping cuboid corner masks. The first mask originates at coordinate `(1,1)` and ends at `(3,4)`, the second mask originates at `(2,2)` and end at `(3,4)`. As can be seen, this way of representing masks is very compact, a binary sensor mask would require the storage of a value for each individual point contained in the masks, whereas cuboid masks only define spans of coordinates for each dimension.

```
% ... Preceding initialization steps

% Defining a cuboid mask
sensor.mask = [2 2 4 5; 3 3 4 5].';

% Running the simulation
sensor_data = kspaceFirstOrder2D(kgrid, medium, source, sensor);
```

Listing 2.3: Creation of a cuboid corners mask in the k-Wave toolbox

Listing 2.3 contains the definition of a cuboid corners mask in the MATLAB version of k-Wave, similar to one of the mask definitions in the k-Wave example folder [3]. Each cuboid is defined as a sequence of numbers in the format `[x1_start y1_start x1_end y1_end]`, forming the span of columns that will be the part of the cuboid mask. If multiple masks are defined, each mask is stored in a separate column of the array. Because figure 2.2 shows NumPy coordinates, all indices in the MATLAB example are incremented by 1.

[3]Similar to an example from k-Wave: `examples/example_ipv_opposing_corners_sensor_mask.m`

PML

The PML (Perfectly Matched Layer) is a layer at the boundary of the simulation medium. It emulates the effect of pressure waves leaving the simulation medium. Without the PML, the waves would cause 'echoes' in the simulation medium, creating noise in the sensors.

2.3 Data exchange with k-Wave accelerators

For simulations with smaller grid sizes, the default MATLAB implementation of k-Wave provides sufficient performance. However, larger simulations require the usage of external *accelerators* that implement the core algorithms in more efficient compiled languages such as C++. The accelerators function as self-sufficient command-line utilities that accept arguments in order to configure the simulation, the simulation data discussed in the previous section 2.2, is passed to the accelerator using an HDF[4] file. k-Wave that is being run with an accelerator does not call the `kspaceFirstOrder...()` function but instead saves all simulation input data to an input file (sec. 2.3), calls the accelerator specified by the `options.cpp_binary_name`[5] variable. The accelerator then executes the simulation, saving the results back to an output file (sec. 2.3). The k-Wave then reads the externally computed result and continues normal operation.

Exchange of data using HDF files is well suited for this purpose given, the number of variables needed for the initialization of the simulation. The format is also open source and platform independent, enabling the implementation of accelerators in any language which supports the HDF format. Another important feature is the ability to stream data to/from on-disk files, saving memory when working with large domains. This section describes three kinds of HDF files that are used in k-Wave: *input files*, *output files*, and *checkpoint files*.

Input files

The input file is created by k-Wave-MATLAB and consumed by the accelerator. It contains data required to initialize and execute the simulation. The data fields are grouped according to the structure of the simulation input data (sec. 2.2). The location and type of used sensors are stored in the input file, fields being recorded are toggled using command line flags (sec. 4.6). The full list of input fields is available in the k-Wave manual[20, p. 71].

Output files

Output files are created by the accelerator upon the completion of the simulation. Apart from basic information about the simulation, which is similar to fields in the input file, results of simulation and values recorded by the sensors are also stored. The recordings can either be scalar (maximum pressure, minimum velocity, etc.), contain the entire state of the grid (for example, final state of pressure), or contain a value for each sensor grid point and time step, forming a timeline of values for further analysis. The last type of recorded data is the data captured using a cuboid sensor. Cuboid sensors capture a slice of the array for each time step, the shape of which is defined by the cuboid mask. All available output fields are described in the k-Wave manual [20, p. 75].

[4]`https://www.hdfgroup.org/solutions/hdf5/`

[5]Even though the name contains `cpp`, any executable program or script can be specified, as long as it is compliant with the k-Wave command line interface

Checkpoint files

Checkpoint files only contain the grid size and the current simulation state: pressure, velocity, density, and the current time step index. This file type is used during long-running simulations to avoid the need to re-run the entire simulation in case of an outage or an error were to occur. Unlike input and output files, checkpoint files are only created if enabled using a command line argument.

2.4 k-Wave tester

The k-Wave tester tool was created for standardized testing of both the MATLAB toolbox and the accelerators. The test suite is located in the `kwave/testing/kWaveTester` folder, the core implementation of the test script is in the `kWaveTester.m` file. The target executable to be tested can be set using the `options.cpp_binary_name`, as discussed in section 2.3.

The `options.custom_test_case` list contains various flags for setting up the simulation. The utility `kwt_save_input_data.m` can be used for saving the customized simulation files to disk, making it a useful tool when testing various types of simulation implementations. The script `kwt_run_omp_comparison_tests_<DIM>.m` can be used for automatic testing of many different simulation settings combinations. The tester first runs the k-Wave-MATLAB version of the simulation, storing recorded results for later comparison. After the k-Wave-MATLAB simulation ends, the tester creates an input file, calls the accelerator specified by `options.cpp_binary_name` and reads the results from an output file. The values measured by the accelerator stored in the output file are then compared with the reference results, creating the graphs and metrics described below.

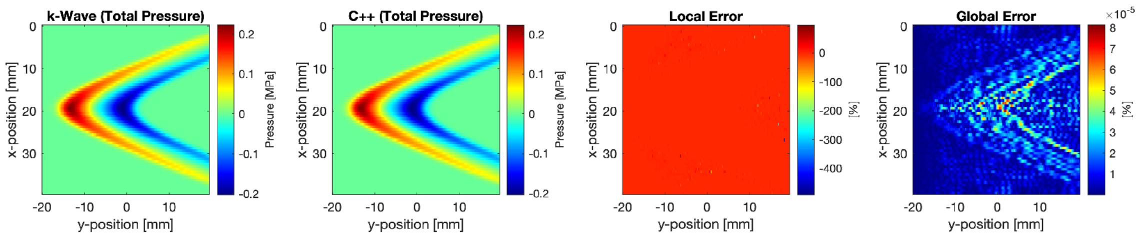


Figure 2.3: kWaveTester example output

Figure 2.3 shows an example output of the kWaveTester tool. The leftmost plot shows the reference output generated by the k-Wave-MATLAB, the picture next to it the output of the accelerator being tested. The images show the values captured as the pressure wave passes through the sensor. The two images on the right show the local and global error measured in percent. The local error displays the difference between reference and measured values for each grid point individually. The global error shows error as percentage of the maximum recorded value.

A text summary of the test performed by the kWaveTester is always listed at the end of the simulation log file. The output for the simulation displayed in 2.3 can be seen below.

```
C++ ACCURACY COMPARED TO MATLAB:
--------------------------------

Error in sensor_data(1).p
    MAX VALS = 223497.7471 (MATLAB) 223497.8125 (CPP)
    L2 = 0.00088376
    LINF = 0.20907 (9.3546e-07 normalised to max value)
```

Listing 2.4: kWaveTester test log summary

The output log summary in listing 2.4 contains three metrics. The `MAX VALS` shows the maximum values recorded in both MATLAB and the accelerator being tested. If the maximum values are similar but the recorded results are not, it could mean that the calculation is performed correctly, but the sensor is recording at incorrect indices (indexing is a common source of issues, as discussed in section 3.3). The `L2` shows the RMSE (Root Mean Square Error) [1] of the compared results. The `LINF` measures the maximum absolute difference between values in the output, this value is then normalized by dividing it by the absolute maximum measured value. For testing CPU implementations of k-Wave, the tester sets the error tolerance at 1×10^{-5}.

Chapter 3

Guidelines for converting MATLAB code to Python

This chapter will summarize the main differences between MATLAB and Python, the important aspects of code conversion, mainly those related to performance. Differences in basic syntax of both languages will not be discussed.

Because Python does not support many scientific computing features present in MATLAB by default, this guide will use the NumPy library for performing such tasks. The NumPy documentation includes a guide created for MATLAB users [8] that summarizes all basic equivalents between MATLAB and NumPy code. The summary of the recommendations in this chapter is available in section 3.9.

3.1 Basic comparison of MATLAB and Python

The author of Python, *Guido van Rossum*, called Python a language that can "*glue together existing components*" [12], speeding up development times while offloading performance-intensive tasks to languages like C, C++ or Java [12]. Although Python has since outgrown this original purpose, the description matches the intended use case – setup of the simulation (loading of arrays, setting up the simulation environment) can be quickly implemented in Python while the performance-critical simulation algorithms are executed in more performant languages without the overhead of calling an external executable.

MATLAB is described as a "*computing environment for engineers and scientists*" [18]. This math-first, programming-second approach gives MATLAB an edge when it comes to scientific computing – it is highly optimized for science-related tasks [17]. Python, on the other hand, was first designed as a programming language and scientific computing is just one of the many applications.

Both approaches have advantages and disadvantages - MATLAB is tried and tested when it comes to scientific computing but is also monolithic, Python is malleable and can be applied to any task, albeit at the cost of being less polished in certain aspects. The two approaches are also reflected in the respective available toolboxes – official MATLAB toolboxes come preinstalled and are tightly integrated with the product, Python requires the installation of third-party libraries to add advanced functionality. Third-party libraries can also be advantageous because of greater flexibility – the user is not tied to one particular library, and a library can be swapped for another one if required. Python users can also benefit from a large ecosystem. As of the time of writing, the PyPI[1] contains more than 350 000 Python packages [10]. Another big difference between MATLAB and Python is that MATLAB is proprietary, whereas Python and the vast majority of third-party libraries are free and open source.

Both MATLAB and Python share a similar level of abstraction, the user does not have to manually manage memory allocation and other low-level tasks, but this level of abstraction comes at the price of performance.

3.2 Python computation libraries

In order to implement k-Wave using Python, it is necessary to find suitable libraries to replace built-in MATLAB functions in two areas – fast computation and visualization. This section discusses selected replacement libraries used in the Python implementation.

NumPy

The NumPy[2] library is used for vector computations in Python. It allows efficient storage of n-dimensional arrays of data and efficient vectorized computations with said arrays. These features are essential for efficiently computing simulation step operations in parallel, as k-Wave often utilizes arithmetic operations on n-dimensional data. NumPy also serves as the foundation for many other scientific libraries. Due to this de facto industry standard status, NumPy objects are natively supported in many other libraries. The performance of FFT-related functions in NumPy falls short of MATLAB, which internally relies on the highly optimized FFTW library [16].

numexpr

One of the disadvantages of NumPy is that it only optimizes individual arithmetic operations between two operands, which can lead to unnecessary allocation of arrays that store intermediate results. The *numexpr*[3] library further optimizes NumPy expressions to reduce unnecessary reallocation.

[1]Python Package Index
[2]`https://numpy.org/`
[3]`https://github.com/pydata/numexpr`

Table 3.1: NumPy vs NumExpr benchmark comparison for the expression `1.5 * x**2 - (x + y + z) * g`

Domain size	NumPy duration [s]	NumExpr duration [s]
32^3	0.28	0.22
64^3	0.26	0.19
128^3	0.20	0.18
256^3	0.21	0.20
512^3	14.40	1.71

A short benchmark 3.1 has been performed to illustrate the impact of numexpr on computation speed. The machine used for the benchmark is described in chapter 5. A random expression `1.5 * x**2 - (x + y + z) * g` has been chosen for the benchmark, the expression represents the kind of calculation that might be performed during simulation. Each of the variables `x`, `y`, `z`, and `x` is a three-dimensional NumPy array in the shape of a cube, the side of which is equivalent to the domain size (`32x32x32`, etc.). With smaller domain sizes, the difference between NumPy and NumExpr is not that significant, but as the domain gets bigger, the reallocations NumPy performs start slowing down the computation significantly.

Despite the significant speed improvement, `numexpr` is relatively easy to implement. The expression is simply wrapped into `numexpr.evaluate(„1.5 * x**2 - (x + y + z) * g", out=output_array)`, the numexpr compiler takes care of the optimization. Although the compilation of expressions also incurs a time cost, the speed gain for larger domains largely outweighs the disadvantages.

The only disadvantage of numexpr is, that it only supports arithmetic operators and a set of predefined functions, it is not possible to use Python functions in the target expression. In k-Wave, this is problematic mainly because of the frequent use of FFTs.

pyFFTW

pyFFTW[4] is the Python wrapper around the FFTW library[5]. It provides access to well-optimized FFT functions that have built-in multithreading support. The interface of the library is similar to NumPy FFT functions, NumPy FFT drop-in replacements are also included with the library for simple replacement of NumPy. The drop-in replacements provide a small speed boost with almost no changes required to the source code [14], although certain modifications are required to fully utilize the available performance of the library. The most significant difference between NumPy and pyFFTW functions is, that NumPy returns a newly initialized array with results, pyFFTW overwrites an internal result array and returns a reference to it. The internal array makes the computation faster by eliminating inefficient copying of data. On the other hand, it breaks interchangeability with NumPy FFT functions.

[4] https://pypi.org/project/pyFFTW/

[5] https://www.fftw.org/

matplotlib

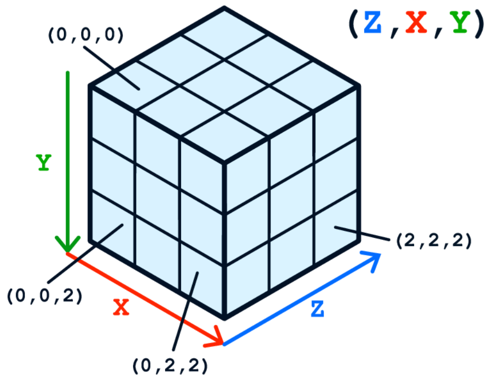
Matplotlib[6] is a Python data visualization library. It offers various kinds of commonly used plot types, all of which are highly customizable. The library is also well integrated into other libraries and tools, for example such as `pandas`[7] and `Seaborn`[8], plots created by Matplotlib are natively supported in data science toolkits such as `Jupyter`[9]. Another advantage of Matplotlib is its native support of NumPy arrays, which are often used during implementation of high-performance computing algorithms.

3.3 Indexing

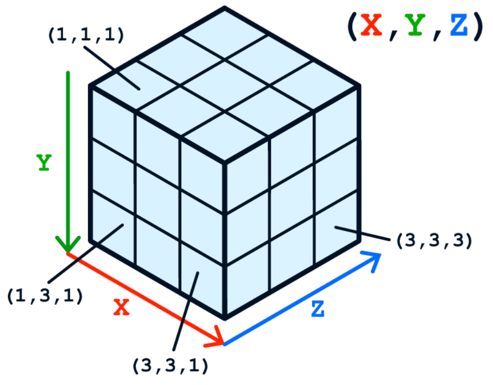
The first major difference between the languages is array indexing. Differences in array indexing might not be noticed at first, but will cause erroneous results and indexing errors during runtime. For this reason, indexing differences must always be remembered when converting code.

The first notable difference is that MATLAB uses column-major (Fortran-style) indexing by default [19] and Python uses row-major (C-style) indexing. Additionally, indexing starts at 1 in MATLAB and at 0 in both Python and NumPy [8], this means all indices must be subtracted by one during conversion.

To illustrate, a 3D array that is indexed `p(1, 2, 3)` in MATLAB would be indexed `p[2, 0, 1]` in Python. When letters are used to denote dimensions in a 3D array: `X` for row, `Y` for column, and `Z` for frame (in other words the z coordinate), MATLAB dimensions are ordered `p(X, Y, Z)`, and Python dimensions are ordered `p[Z, X, Y]`.



(a) NumPy



(b) MATLAB

Figure 3.1: Comparison of indexing

When working with 3D arrays, it is useful to think of them as cubes when slicing. Figure 3.1 shows a comparison of the indexing of 3D arrays in NumPy and MATLAB. Both arrays are of shape `(3, 3, 3)`. Figure 3.1a uses C-style indexing used in NumPy, indexing starts at `0`. The maximum index for each cardinal direction is therefore `2`. Figure 3.1b shows indexing in MATLAB, which uses the Fortran-style indexing. Additionally, indexing in MATLAB starts at `1`, the maximum index for each dimension is therefore `3`.

[6]https://matplotlib.org/
[7]https://pandas.pydata.org/
[8]https://seaborn.pydata.org/
[9]https://jupyter.org/

The different indexing style together with different starting indices can be confusing. Moreover, additional precautions must be taken when dimensions of the array are of different lengths. An array of shape `(128, 64, 32)` in MATLAB would be displayed as an array of shape `(64, 32, 128)` in NumPy.

As of the time of writing, indexing in the k-Wave input files is neither row-major, nor column-major. All arrays are stored from the last dimension to the first – 3D arrays in the order `(Z, Y, X)`, 4D arrays (with the time dimension) as `(T, Z, Y, X)`.

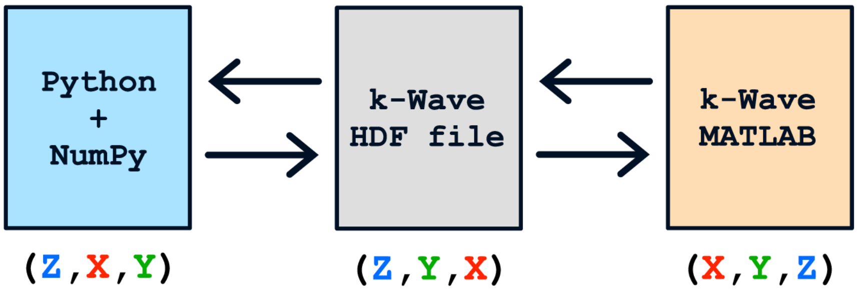


Figure 3.2: Coordinate conversion schema

The conversion between all used coordinates systems can be seen in figure 3.2. Appropriate indexing conversions must be performed any time when reading from and saving to a different format.

Linear indexing

To conserve memory and disk space, k-Wave uses *linear indexing*[10] to store mask indices for sensors and sources. Linear indexing compresses an n-dimensional array coordinate to a single number, which denotes the array position if the array were to be collapsed to a one dimensional array. It is important to correctly convert linear indices stored by MATLAB to their corresponding NumPy counterparts, this can be done using the function `np.unravel_index(flat_indices-1, shape, order='F')`[11], the shape of the original array must also be specified.

A common task is the lookup of all linear indices in an array that contain nonzero values. In MATLAB, this is done using the `find(array)` function, which returns an array of all non-zero linear indices found in the array. The equivalent function in NumPy is `np.flat_nonzero(array)`[12]. MATLAB allows direct indexing using linear indices such as `array(123)`. In NumPy, a special flat indexer `array.flat[123]` must be used.

3.4 Data structures

Structs[13] are used in MATLAB to group data semantically, they can either be created explicitly using `struct()` or implicitly by assigning a field to a struct (even one that does

[10] Alternatively called *flat indexing*
[11] https://numpy.org/doc/stable/reference/generated/numpy.unravel_index.html
[12] https://numpy.org/doc/stable/reference/generated/numpy.flatnonzero.html
[13] https://www.mathworks.com/help/matlab/ref/struct.html

not exist) like `struct1.field1 = value1;`. Python does not have any direct equivalents, but there are multiple possible replacements.

The first option is to use *dictionaries*, a data type containing key-value pairs initialized like `dict1 = {„field1": value1}`. Dictionaries provide a fast and easy way to group related data, but lack the means to enforce data types, declare mandatory fields and add logic like classes do.

The second option is to use classes, which allow for encapsulation [13, p. 14] of the initialization logic. In the case of classes that only contain fields and do not require any substantial related logic, *data classes*[14] can be used. Data classes automatically generate boilerplate code from declared fields. They can be declared by adding the `@dataclass` decorator to a class definition and moving the fields from the constructor to the class body.

3.5 Matrix and scalar arithmetic

Arithmetic operators in MATLAB are either scalar (`+-*/`) – they can be used to perform arithmetic on scalars or between a scalar and a matrix, division and, multiplication operators also have a variant for performing element-wise operations with two matrices, the operators start with a dot (`.*` and `./`). MATLAB also has a dedicated operator `.'` for transposing matrices.

Python only has operators `+-*/` (and a dedicated matrix multiplication operator '`@`'), which perform different operations based on context. New operators can not be defined, but they can be overloaded by a subclass to implement custom behavior. NumPy exploits this property of Python, two NumPy arrays always perform element-wise operations when multiplied/divided. Python does not provide an operator for transposition, a matrix can be transposed using the `array.T` attribute of NumPy arrays, the equivalent in MATLAB would be `array.'`.

Data broadcasting

A special case is the MATLAB function `bsxfun`[15]. It performs an element-wise operation between two matrices with the distinction that they don't need to have the same shape, for example, the matrix `A` of shape `(32, 32, 32)` (3D) can be multiplied efficiently with the array `B` of shape `(32, )` (1D). Normally, array `B` would need to be enlarged to have the same shape as `A`, `bsxfun` performs this without this additional step making the operation faster while also saving memory, NumPy performs this optimization automatically. When two matrices of mismatched shapes are used, NumPy performs automatic *broadcasting*[16]. The only requirement is that the matrices must have at least one length in common – array of shape `(32, 32, 32)` can be multiplied with shape `(32, 1, 1)` or `(32, 32, 1)` but not with `(16, 1, 1)`.

As an example, the expression from k-Wave-MATLAB kspaceFirstOrder3D `rhox = bsxfun(@times, pml_x, bsxfun(@times, pml_x, rhox) - dt .* rho0 .* duxdx);` can be rewritten in NumPy to `rhox = pml_x * (pml_x * rhox - dt * rho0 * duxdx)`, the clarity is greatly improved thanks to implicit broadcasting, matrix and scalar multiplication also use the same `*` operator.

[14] https://docs.python.org/3/library/dataclasses.html
[15] https://www.mathworks.com/help/matlab/ref/bsxfun.html
[16] https://numpy.org/doc/1.22/user/basics.broadcasting.html

3.6 Fourier transforms

Performing FFT related computations can be done efficiently in MATLAB using the built-in functions `fft`, `fftn`, `ifftn`, etc. NumPy has equivalent functions in the package `numpy.fft`, they are not suitable for high performance applications, as is discussed in chapter 5. It is advisable to use the `pyFFTW` library as it is also used for FFT in MATLAB [7].

Caution is required when replacing NumPy FFT functions with pyFFTW functions, although the functions and their arguments are similar, there are some notable distinctions. This section will compare FFT functions in NumPy and pyFFTW, namely `fftn` and `ifftn` which are often used in k-Wave.

Firstly, NumPy FFT functions promote input data types differently from pyFFTW. With an input of type `float32`, the resulting datatype will be `complex128` in NumPy but `complex64` in pyFFTW. With an input datatype of `float64`, datatype of the result will be `complex128` for both libraries. This discrepancy in input and output data types can cause differences in results, it is also not easily detectable during debugging because of Pythons dynamic type system. More details about this and other differences between NumPy and pyFFTW can be found in the pyFFTW manual [3, p. 20].

Another important difference is that results from calls to NumPy FFTs return a newly initialized array each time, pyFFTW (when not used in the drop-in mode, as discussed in section 3.2) returns a reference to the same internal result array[3, p. 11], results must be copied using `numpy.array.copy()` as a subsequent call to the function overwrites the internal result array, causing the previous result to be destroyed. Each pyFFTW function that is initialized by default creates an internal input and output array, to avoid extra memory allocation, the function `func.update_arrays(input_array, output_array)`[17] can be used to set a single shared array for multiple functions, it can also be used to make pyFFTW use the same array for both input and output for a given function.

During initialization, FFTW compares a number of FFT algorithms and chooses the fastest one in a process called *planning*, the generated configuration is called *wisdom*[18]. pyFFTW supports different kinds of FFTW planning with varying speed and optimality: `FFTW_ESTIMATE`, `FFTW_MEASURE`, `FFTW_PATIENT` and `FFTW_EXHAUSTIVE` [2].

The `FFTW_MEASURE` planner (the default option in pyFFTW) provides a good compromise between planning speed and performance, it was therefore chosen for k-Wave-Python. Because planning can have a noticeable startup cost, it is best to cache the generated wisdom to a file for re-use. In the pyFFTW library, exporting can be done using the function `pyfftw.export_wisdom()`[19], importing is likewise done using `pyfftw.import_wisdom(wisdom)`. The Python standard library *pickle*[20] can be used for serializing and deserializing of wisdom for storage on disk.

3.7 Data visualization

Although visualization is not necessary for k-Wave to function, it is useful for checking progress of a simulation. As previously said in section 3.2, Matplotlib is the library of choice for visualization in the Python implementation. In the following paragraphs, the

[17] https://pyfftw.readthedocs.io/en/latest/source/pyfftw/pyfftw.html#pyfftw.FFTW.update_arrays
[18] https://www.fftw.org/fftw3_doc/Wisdom.html
[19] https://hgomersall.github.io/pyFFTW/pyfftw/pyfftw.html#pyfftw.export_wisdom
[20] https://docs.python.org/3/library/pickle.html

symbol `plt` refers to the library imported using the standard `import matplotlib.pyplot as plt` command.

For one dimensional simulations, a simple line graph can be used for visualization, a pressure curve can be plotted using `plt.plot(data)`[21].

For two and three dimensional simulations, the visualization technique is the same because only a two-dimensional slice of the array can be plotted.

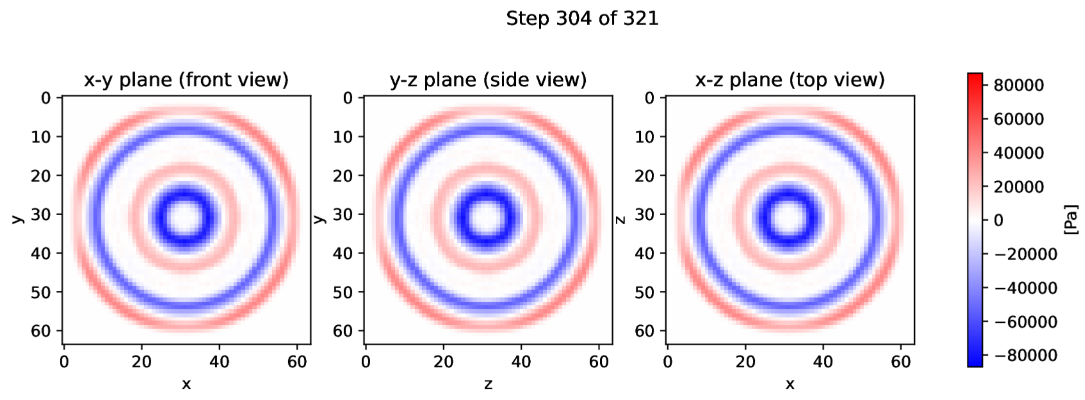


Figure 3.3: Example plot using `plt.imshow()`

The `plt.imshow(data)`[22] can be used for visualizing 2D slices of arrays as images. Figure 3.3 shows a k-Wave-Python simulation progress visualized using Matplotlib.

3.8 Design patterns

During the implementation of the object oriented Python version of the k-Wave toolbox, numerous measures were taken to improve the code structure of the simulation. One of the means of improving code structure is to utilize software *design patterns*, solutions to commonly occuring problems in software engineering. The advantage of design patterns is their abstract nature – they can be applied to any programming language which has at least some of the concepts in OOP: *abstraction*, *encapsulation*, *inheritance*, and *polymorphism* [13, p. 18].

Design patterns are divided into three categories: *creational* (concerned with creation of objects), *structural* (concerned with creating larger structures from objects), and *behavioral* (concerned with communication and interaction between objects) [13, p. 29]. This section lists multiple important design patterns used in the implementation: the *Builder*, the *Observer*, the *Strategy* and the *Adapter*.

Builder

The Builder is a creational design pattern [13, p. 105]. The task of the Builder is to make the initialization process of objects more flexible. With simple classes, initialization using the constructor is sufficient, but with complex objects like the ones used for k-Wave simulations, the class constructor becomes very long, as well as the parameter list. The

[21] https://matplotlib.org/3.5.0/api/_as_gen/matplotlib.pyplot.plot.html
[22] https://matplotlib.org/3.5.0/api/_as_gen/matplotlib.pyplot.imshow.html

Builder solves the issue of complicated initialization by breaking it down into separate steps. The builder instance holds a semi-initialized instance of the target class internally, the instance is initialized step by step using setter methods of the builder.

There are multiple advantages to this approach: the constructor is simplified, the initialization is separated from the usage of the class (this also enables creating different builders for the same class), some parts of the instance can only be initialized optionally, the parts can also be initialized in any order since the order in which the setters are called is not predetermined. After all initialization steps are complete, the fully initialized instance is retrieved from the builder. Builders are used in the k-Wave-Python implementation for initializing the `kspaceFirstOrder` simulation classes (sec. 4.3).

Observer

The Observer, a behavioral design pattern [13, p. 336], allows the state of an object (or changes thereof) to be observed by any number of *Observers*. The *Observable* (the object being observed) curates a list of observers, allowing them to be added and removed at any time. After an event in the observable occurs, it notifies each observer in its list of the change, the observers can then perform any logic defined by their callback functions. The advantage of using observers is the independence of the logic responsible for recording values from the core logic of the simulation. This design pattern is closely related to the *publish/subscribe model* [5, p. 158]. The `kSensor` objects (sec. 4.5) are observers observing the `kspaceFirstOrderBase` subclasses.

Strategy

The Strategy, similarly to the observer, is a behavioral design pattern [13, p. 368]. It is used when a group of algorithms share a common structure but differ in certain aspects or settings. Conventionally, algorithm variants and options are handled by branch statements, this however becomes more and more difficult as more variants of the algorithm are added, making the algorithm difficult to navigate. The Strategy solves this by keeping the structure of the algorithm but delegating the parts that differ to objects which encapsulate the differing logic. During initialization, the suitable strategy (algorithm variant) is chosen based on configuration. This way different algorithms can be used with the same basic structure without the need for changing the core structure. This pattern is used in the core simulation algorithms of `kspaceFirstOrder` 4.3 for setting up pressure and velocity sources.

Adapter

The Adapter is a structural design pattern [13, p. 150]. It solves the problem of compatibility between two parts of a program that need to communicate, but the format of data they use is different. Conversion to the target format could be handled by the consumer of the data, but this approach does not scale well when the data source is being used in multiple places. Furthermore, if the source format changes in any way, the change needs to be reflected everywhere the source is used. The Adapter solves this by wrapping the original source of data, creating an interface between the data source and the data consumer. Any time the consumer requests data, the adapter can translate the request to the data source. After the data from the source is retrieved, it can be converted to the target format and passed to the consumer.

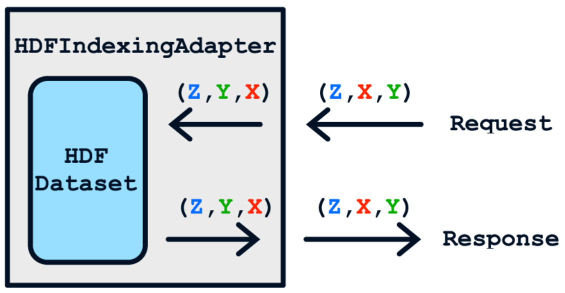


Figure 3.4: Indexing adapter

The class `HDFIndexingAdapter` is an example of an adapter. Because the k-Wave-MATLAB stores arrays using `(Z, Y, X)`, it is incompatible with the rest of the simulator, which uses C-style indexing. The situation can be seen in figure 3.4. Because data from the input files is handled in many places of the program, an Adapter is required to ensure compatibility without making the implementation-dependent on a particular indexing style. If the indexing in the input files changes in the future, the adapter can either be easily modified to accommodate the change or discarded if the indexing in the input file becomes C-style.

3.9 Summary

This section contains the summary of technologies and guidelines discussed in this chapter.

1. Convert basic syntax (if statements, for loops, function definitions, etc.) to Python.
2. Replace MATLAB structs (sec. 3.4) with classes, data classes or other equivalents.
3. In places where multi-dimensional arrays are used, use the NumPy library (sec. 3.2).
4. Convert operators to operators used by NumPy (sec. 3.5), replace the `bsxfun(...)` with a simple `*` multiply operator, NumPy performs automatic broadcasting.
5. When indexing, always remember to subtract 1 (usually not in code, just during re-writing) from indices when re-writing. If indices are being loaded from an input file (sec. 2.3), subtract 1 before doing other operations, as the indices were stored by MATLAB, which starts indexing at 1.
6. When indexing multi-dimensional arrays, remember to convert Fortran-style indices to C-style indices (sec. 3.3). Apply extra caution when loading linear/flat indices from an input file, as they are stored in MATLAB column-major order.
7. Preferably use the pyFFTW library (sec. 3.2) for good FFT performance.
8. Use numexpr (sec. 3.2) where possible, avoid creating temporary copies of arrays by preallocating them. Perform in-place operations where possible.
9. If visualization is needed, use matplotlib (sec. 3.2) but be aware of the performance implications.
10. For details about NumPy equivalents for MATLAB features, refer to the official guide [8].

Chapter 4

k-Wave-Python implementation

The goal of the implementation is to create a Python version of the core k-Wave sound propagation simulation and all key features surrounding it. The simulation should be able to load all necessary data from an input HDF file, initialize all objects required for the simulation, perform the specified simulation and finally save all results and recorded outputs to an output HDF file. The implementation must be able to communicate with k-Wave-MATLAB using the command line, passing input and output data with HDF files.

The command line interface must be compliant with the options specified in [20, p. 54], ideally interchangeable with the k-Wave-Fluid-OMP [22] C++ implementation.

Apart from specified functional requirements, the implementation has multiple goals:

1. **Performance** - performance of the new implementation must be better or at least comparable to the original MATLAB version
2. **Accuracy** - the results must be reasonably accurate and not deviate from the reference implementation
3. **Ease of use** - the simulator should be easy to set up, easy to use and easy to extend

The secondary focus is on modularity and extensibility – k-Wave-MATLAB is tightly coupled, this means it is sometimes difficult to modify or access k-Wave functionality separately, examples of this include duplication of certain snippets of logic and code used for visualization located directly in the simulation loop. k-Wave-Python attempts to mitigate such issues by adhering to the *DRY (Don't Repeat Yourself) principle* [5, p. 26] completely separating logic related to simulation and other logic used for recording values and loading/saving of arrays. The final implementation of k-Wave-Python consists of around **2000 lines of code** (not including empty lines and comments), which is a substantial improvement over versions written in C++.

This chapter describes the design and implementation of k-Wave-Python. Differences from the original MATLAB implementation, architectural choices and improvements over the original implementation are also discussed.

4.1 Project structure

Unlike the monolithic, predominantly procedural MATLAB implementation of k-Wave, the Python implementation puts a much greater emphasis on modularity and abstraction, as was stated in the implementation goals. The MATLAB implementation usually defines variables in the global scope and manipulates them by calling functions or executing entire modules that contain logic in the global scope. This approach is more convenient due to the large number of variables, but it also has many disadvantages:

1. **hard to unit test** - very long functions (or entire modules) that complete many different tasks in sequential order are difficult to unit test, because it is not possible to test individual parts in isolation.
2. **hard to re-use** - for the same reason, code that is not sufficiently subdivided into functions is difficult to re-use. This leads to the MATLAB implementation having many duplicated snippets of code that could otherwise be defined once, making the source code longer and harder to maintain.
3. **hard to extend** - the global scope allows functions to have immediate access to all needed data, but on the other hand also other unrelated data. By minimizing the amount of data a function has access to and hiding implementation details, it is possible to create code that is easier to extend and test – making changes to one part of the program will not affect other parts.
4. **hard to understand** - unclear flow of data within the program and low abstraction make the code more difficult to understand. It is not always clear, where a change of state ocurred because many variables are being modified in the global scope, semantic blocks of code that would otherwise be labelled by their function name are left without context.

4.2 Input and output file handling

k-Wave simulations are initialized using standardized input files (sec. 2.3) generated by k-Wave-MATLAB. These files are also the main means of communication between the Python modules and the MATLAB modules.

Input files

The `h5py` library is used for accessing and manipulating HDF files. It provides a simple interface `h5py.File` to access a specified file similarly to a Python `dict`. The module `kwave.utils.h5_utils` contains classes that provide additional layers of abstraction over `h5py.File`.

The class `H5File` which wraps `h5py.File`, simplifies retrieval and setting of values in HDF files, it also handles retrieval of scalar values from input files. Since scalar values are written by k-Wave-MATLAB to the input files as an array of dimensions `(1, 1, 1)`, it is necessary to unwrap it before retrieval. It is also needed to convert unsigned integer values to the NumPy type of `int64` because data types are not enforced in Python, making the resultant bugs related to unsigned values hard to detect.

The final abstraction layer is the `kWaveH5Dataset`, it wraps the `H5File` and serves for loading specific fields required during initialization of the simulation. The individual fields can either be accessed directly using the indexer like `file[„field"]`, `dataset[„p0_source_flag"]` or using specialized methods and attributes. Some methods require the dimension (`x`, `y`, `z`, `t`, etc.) to be specified - `dataset.get_kgrid_dim(„x")`.

One of the implementation issues is a relatively large number of parameters that are used. For example, the `kGrid` (k-Wave grid) requires the `Nx` (number of grid points) and `dx` (the distance between grid points), this adds up to 6 parameters used in the constructor, not to mention other additional parameters. To mitigate this, multiple classes used expressly for storing related data were used. They are used only for simplifying the initialization and passing of related data. The list includes `PMLDim` and `PMLDimProperties` for initializing the PML; `SoundProperties`, `DensityProperties` and `AbsorptionProperties` for initializing the `Medium`. These objects are returned by the `kWaveH5Dataset` initialized using the data from the wrapped file.

Output files

The output file is created using the `H5File` class. Input and output files share many common fields, some fields are therefore directly copied from the input file to the output file. Other fields are populated using results from the simulation and data from `kSensor` sensor based on selected CLI flags.

4.3 kspaceFirstOrder

The group of kspaceFirstOrder classes represent the core logic of the simulation algorithm. The k-Wave-Python implementation focused only on the three-dimensional `kspaceFirstOrder3D` variant of the simulation. The lower-dimensional variants have a similar structure and can be implemented similarly to the three-dimensional implementation. Because the implementation relies on inheritance, the simulation equations are shared, avoiding duplication of logic.

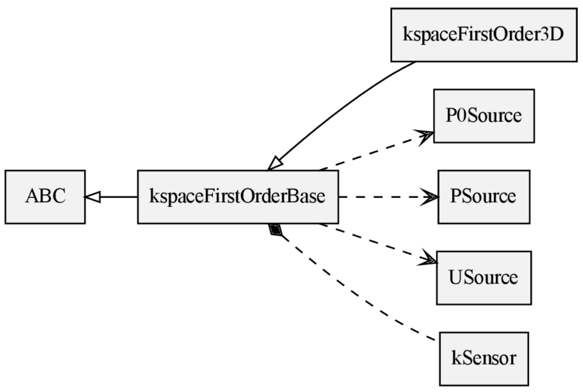


Figure 4.1: kSpaceFirstOrder class hierarchy

The `kspaceFirstOrderBase` class hierarchy can be seen in figure 4.1. All shared logic is contained in the base class `kspaceFirstOrderBase`. Shared logic includes simulation equations, management of attached sensors, control of simulation step iteration, calculating number of remaining steps, etc. The base class contains multiple important abstract methods: the `init_data()` method handles the allocation and the initialization of simulation data, the `_sim_step` is the core of the simulation algorithm. These two methods are overridden by the simulation subclasses and adapted to the number of dimensions. The simulation class is dependent on pressure and velocity sources, which are described in section 4.4. For the recording of data during the simulation, the simulation class also holds a list of sensors, which are described in section 2.2.

The class `kspaceFirstOrder3D` inherits the previously mentioned features and adapts them for three-dimensional simulations, overriding the `init_data()` and `_sim_step()` with three-dimensional simulation logic. The method `_sim_step()` containing the simulation algorithm is significantly shorter than the k-Wave-MATLAB counterpart as most of the equations, recording, and pressure/velocity sources are hidden behind interfaces. Unlike the k-Wave-MATLAB, the k-Wave-Python is not aware of the type of source being used, nor is it aware of the equation of state, linearity, data recording and visualization. The modularity of the algorithm is the adaptation of the Strategy design pattern, which was described in section 3.8.

kspaceFirstOrder initialization

Full initialization of k-Wave is a complex process, it requires multiple related objects to be initialized, some of them in a fixed order because of interdependence. The initialization steps are the following:

1. Load `kGrid`
2. Load `Medium` and setup absorption variables using `kGrid` data
3. Load `PML` using data from `kGrid` and `Medium`
4. Select equation of state
5. Select linear or non linear mode
6. Configure sources (`p0`, `p` and `u` source)
7. Setup and attach a `kSensorRecorder` for data recording (optional)

The `H5InputDataLoader` class executes this sequence and initializes all objects using data from a specified input file. Instead of directly initializing an instance of k-Wave, an auxiliary data structure called `kspaceInputData` is returned by the loader, this allows greater flexibility when initializing the simulation - the loaded data can be inspected or overridden manually. The only component that is not loaded from the input file is the PML (sec. 2.2), it is instead initialized using data from the `Medium` and the `kGrid`.

Some initialized objects can additionally be adjusted using supplied command line arguments. Namely the *computation backend* (the `backend` option) described in section 4.3), the number of simulation time steps (the `benchmark` option overrides `kgrid.Nt`), and the data recorded by the `kSensorRecorder`. These adjustments are however not handled by the input data loader.

kspaceFirstOrderBuilder

One of the innovations of the Python implementation is the `kspaceFirstOrderBuilder` which implements the *Builder* design pattern (sec. 3.8). It allows the final `kspaceFirstOrder` object to be initialized gradually instead of passing a large number of arguments to the constructor. All objects needed for the initialization can be directly retrieved from the `kspaceInputData` loaded from an input file. The computation backend is selected based on command line options.

Computation backends

Computation backends are classes containing references to different implementations of common functions used by k-Wave during simulation, usually related to FFTs (`fft`, `fftn`, `ifftshift`, etc.). The two currently available backends are `NumpyBackend` and `PyFFTWBackend`, each of them using functions from their respective libraries - NumPy and pyFFTW.

Swappable backends simplify the comparison of benchmarks using different computation libraries. It also allows fast replacements of libraries or fallback libraries in case a library is not available for the target platform. The simulation itself does not know, which backend is being used, the pointers to functions are simply replaced during initialization by the builder.

4.4 Sources

As described in section 2.3, there are three kinds of sources, which inject pressure or velocity into the simulation: p0 sources, p sources and u sources. The k-Wave simulation is initialized with instances of all three of the sources, their methods are called at particular points in the simulation algorithm to inject pressure or velocity, the `self` reference of the simulation is passed to allow access to the internal simulation state. Depending on the selected options, the density and velocity variables of the simulation are changed. When a disabled source is used, the called methods do not perform any action.

P0 sources

The initial pressure sources are the simplest of the tree source types. On the first step of the simulation (`t_index == 0`), the pressure at all grid points is set to pre-determined values, this pressure then propagates and no more pressure is injected during the course of the simulation.

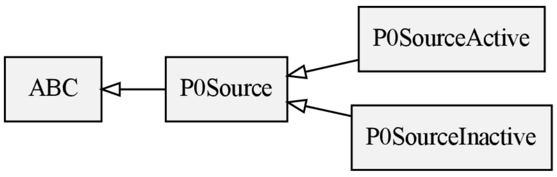


Figure 4.2: P0 source class hierarchy

Figure 4.2 shows the class hierarchy of p0 sources. The class `P0SourceActive` is used, when the p0 source is enabled by the `p0_source_flag` in the input file (sec. 2.3) and is initialized by data from the `p0_source_input` from the input file, the class `P0SourceInactive` is used when the source is disabled. Because the simulation classes (sec. 4.3) only see the `P0Source` interface, the simulation is not dependent on the kind of source used.

P sources

The variable pressure sources add pressure to the simulation continuously. The mechanism of time varying sources is described in section 2.2. The indices, where the pressure is set after each iteration is defined by the `p_source_pos_index` mask indices. The value being set to all points of the mask is selected from a list of pressure values. When `p_source_many` is enabled, the pressure value is set for each point of the mask individually. The series of pressure values can be both shorter or longer than the number of time steps, the source will either not use all the available values or stop emitting pressure if the time index `t_index` exceeds the length of the array.

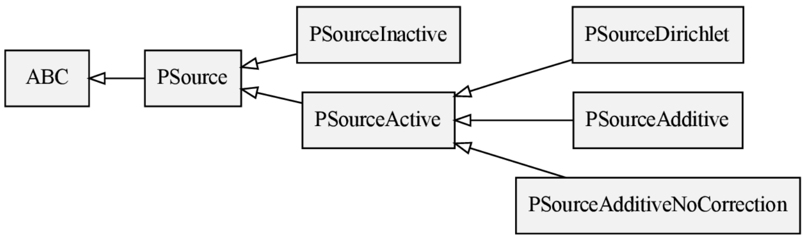


Figure 4.3: P source class hierarchy

Similarly to the p0 source (sec. 4.4), the p source also has its class hierarchy, as seen in figure 4.3. The p source can either be `PSourceInactive` or `PSourceActive`, based on the `p_source_flag` from the input file. `PSourceActive` is further subdivided into three subclasses `PSourceDirichlet`, `PSourceAdditiveNoCorrection` and `PSourceAdditive` based on the configured source mode `p_source_mode`. The simulation only communicates with the `PSource` interface.

U sources

The last source type is the velocity source. Instead of changing the pressure, it injects particle velocity for each cardinal direction. Selection of mask indices is similar to the p source, they are stored in `u_source_pos_index` field in the input file. The velocity source can be independently enabled for each direction by the `ux_source_flag`, `uy_source_flag` and `uy_source_flag` respectively. The values for the velocity sources are also stored separately in `ux_source_input`, etc. The stored values can either be scalar or defined for each source point, depending on the „`u_source_many`" flag. The time varying source mechanism is the same at with p sources (sec. 4.4) and is described in section 2.2.

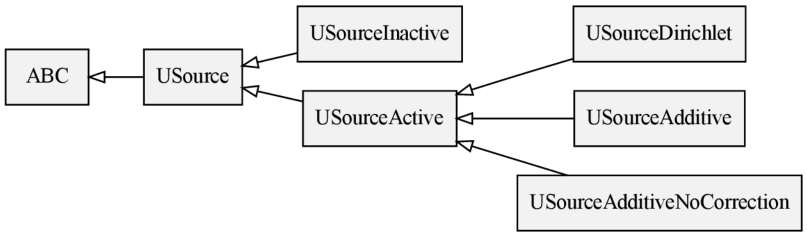


Figure 4.4: U source class hierarchy

The class hierarchy, as shown in figure 4.4, is almost identical to the p source class hierarchy, the difference being the injection of particle velocity instead of pressure. The source can be either enabled with `USourceActive` or disabled with `USourceInactive`, depending on the `u_source_flag` from the input file. The available variants of enabled u sources are `USourceDirichlet`, `USourceAdditiveNoCorrection` and `USourceAdditive`, depending on the `u_source_mode` setting. As with the two previous source types, the simulation only interacts with the `USource` interface.

4.5 Sensors and recorders

k-Wave relies on *sensors* to record measurements of the simulation state, the sensor mask can be defined either as a binary mask or a cuboid corners mask, as described in section 2.2.

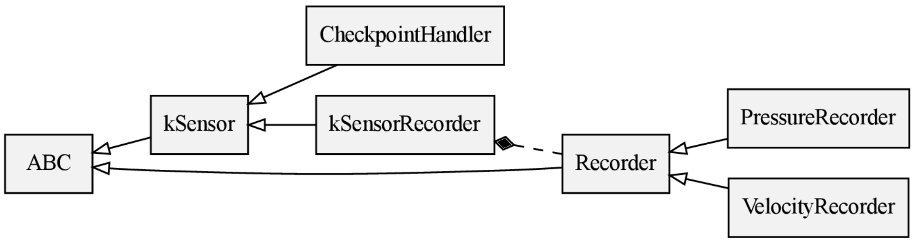


Figure 4.5: Class hierarchy of sensors and recorders

Figure 4.5 shows the class hierarchy of sensors and recorders in k-Wave-Python. The `kSensor` (sec. 4.5) serves as the base class for all sensors attachable to the simulation. The `kSensorRecorder`, which is the subclass of `kSensor` has the same role as the sensors in the original k-Wave-MATLAB. In the original implementation, the values being recorded (for example `p`, `p_max`, `u_rms`) were checked in a large if statement to determine, whether the recording of a particular value is enabled. The `kSensorRecorder` improves this by using a list of `Recorder` (sec. 4.5) instances instead. Each `Recorder` represents a single value being recorded. During the initialization of the simulation, a list of recorders is generated based on the command line recording flags, this list is then passed to the `kSensorRecorder`.

kSensor

k-Wave-Python introduces sensors, that can be optionally attached to the simulator. The simulation itself is not aware of any implementation details of the attached sensors, it simply calls callback functions of each attached sensor after a simulation step, passing itself as an argument in the process. The attached sensors can then extract any needed data directly from the simulation instance which contains current simulation state. The way sensors process data depends solely on the user. Data recorded after each step can be stored in memory and later visualized, saved as a spreadsheet or streamed directly to disk.

This implementation approach is comparable with the *Observer* design pattern (sec. 3.8). The abstract base class `kSensor` contains methods and attributes expected by the simulation objects during callbacks, this base class can then be inherited and easily extended by the user. Sensors are attached to an initialized simulation object using the `.attach_sensor(kSensor)` method at any time during simulation.

Recorder

Recorders handle the recording of individual sensor fields, which can use different slicing and aggregation functions. Examples of fields include `p_raw` (records the entire pressure domain), `p_max` (records the maximum pressure at the sensor grid points for each time step), `u_min_all` (records the minimum particle velocity in the pressure arrays for each direction respectively), etc. The full list of available flags is listed in the k-Wave manual [20, p. 54] under *output flags*.

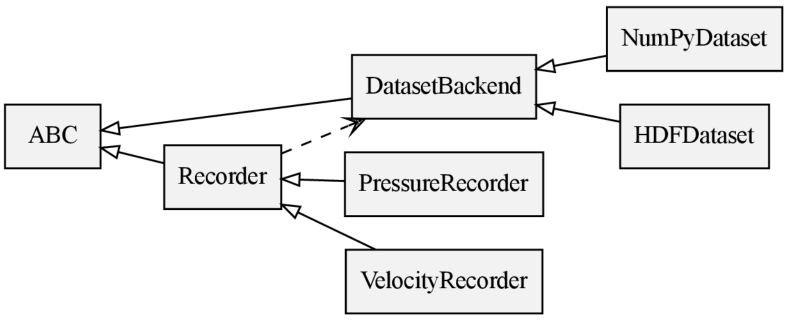


Figure 4.6: Recorder class hierarchy

The above shown figure 4.6 displays the recorder class hierarchy. The `Recorder` class is the abstract base class for all recorder types, it contains abstract methods related to allocation (`_allocate()`), indexing (`_dst_index()`, `_src_data()`) and aggregation functions (`_func()`). To isolate the format to which data is recorded from the recorders, the `DatasetBackend` is used. The `DatasetBackend` provides an abstract interface for the allocation of arrays, accessible using the `allocate(shape, name)` method. Each backend initializes an array differently, the `NumPyBackend` returns a new NumPy array of the specified shape, the `HDFBackend` returns an `h5py.File` HDF file handle.

The pressure and velocity recorder groups both have their subclasses: `PressureRecorder` and `VelocityRecorder`. The fundamental difference between them is, that pressure recorders only allocate one array for pressure recording and velocity recorders allocate up to three arrays (for each velocity direction), depending on the number of dimensions. The individual recorders are further subclassed from these two groups. For instance the previously mentioned `u_min_all` field is represented by the `U_MIN_ALL_Recorder` class.

```
class U_MIN_ALL_Recorder(Binary, Single, VelocityRecorder):
    template = "u{}_min_all"

    def _func(self, current_data, new_data):
        return np.minimum(current_data, new_data)
```

Listing 4.1: Definition of the `U_MIN_ALL_Recorder` class

The definition of the `u_min_all` recorder field can be seen in listing 4.1. The class utilizes multiple inheritance to configure the properties of the recorder. The order of inheritance is generally in the order this order:
`class ExampleRecorder(<indexing_type>, <count_type>, <recorder_type>).`
The `<indexing_type>` configures the type of indexing the recorder field uses:

- `Flat` - record at grid points defined by a list of linear/flat indices (sec. 3.3)
- `Cuboid` - record at grid points defined by a cuboid mask (sec. 2.2)
- `Binary` - record the entire domain

The `<count_type>` defines, whether the values are recorded for each time step (`Multiple`), or if the recorded values overwrite previously recorded ones (`Single`). The `<recorder_type` selects the type of recorder (`PressureRecorder` or `VelocityRecorder`).

Although this approach creates many subclasses and makes certain parts of the recording process less transparent, it greatly simplifies the introduction of changes. A change made in the base class is automatically applied to all subclasses, the entire behavior of a recorder can be changed by simply changing the classes it inherits from, this also allows the creation of new recorders using a very small number of lines of code.

4.6 Command line interface

k-Wave-Python provides a command line interface for simple access to its simulation capabilities. This interface is based on the standardized command line interface for accelerators described by the k-Wave manual [20, p. 54] with the exception of implementation-specific flags (`-g` for GPU accelerators), compression (`-c`) and recording non-staggered grid velocity recording (`-u_non_staggered_raw`). The full list of available arguments and flags can be viewed by using the `-help` flag, the example output can be seen in appendix D.

Apart from the standard arguments, k-Wave-Python introduces non-standard ones. The `-show` flag enables the pressure domain preview visualization which utilizes Matplotlib (sec. 3.2). The refresh rate of the visualization is tied to the `-r <interval_in_percent>` parameter which sets the logging interval in percent. The `show` flag should only be used for debugging and with large refresh intervals, as it will slow down the simulation loop. The other non-standard parameter is the `backend` which sets the default backend used for calculating FFTs as described in section 3.2. The available backends are `numpy` and `pyfftw` (the default), the `numpy` backend being significantly slower. The backend selection can be used for debugging and for comparing various FFT libraries. Non-standard arguments can be disabled and hidden by setting the `compatibility_mode`. For details on usage of the command line, refer to appendix B.1.

Chapter 5

Testing and optimization

This chapter describes the process of testing of k-Wave-Python using the kWaveTester and analyzes performance of the new implementation from the point of both computational performance and memory usage. Performance bottlenecks, their causes and possible avenues for further optimization are also discussed.

If not stated otherwise, all benchmarks were conducted on a laptop with the `Intel i5-8257U` CPU clocked at 1.4 Ghz and 8GB of RAM.

5.1 Testing

The testing of k-Wave-Python was performed using the `kWaveTester` tool from k-Wave-MATLAB. The `kWaveTester` serves as the ground truth because it can both generate input data for any combination of simulation settings and compare the output out the program being tested with the reference results.

The `kwt_run_omp_comparison_tests_3D` test script was used for thorough testing. The script iterates trough more than 200 test configurations, ensuring every aspect of the program is tested. Because of unimplemented features and known issues, some of the test cases generate invalid results. Because k-Wave-Python currently does not support the staggered grid and the kspace, they must be disabled using `options.use_sg = false;` and `options.use_kspace = false` in the k-Wave-MATLAB simulation file that is being run using the kWaveTester (`kspaceFirstOrder3D` in the case of k-Wave-Python). Multiple simulation runs were conducted to test the implementation k-Wave-Python. The simulations used non-linear, heterogenous and absorbing settings (`LIN=1`, `ABS=1`, `HET=2`) and the grid size of 128x64x32 to test as many implementation edge cases as possible at the same time.

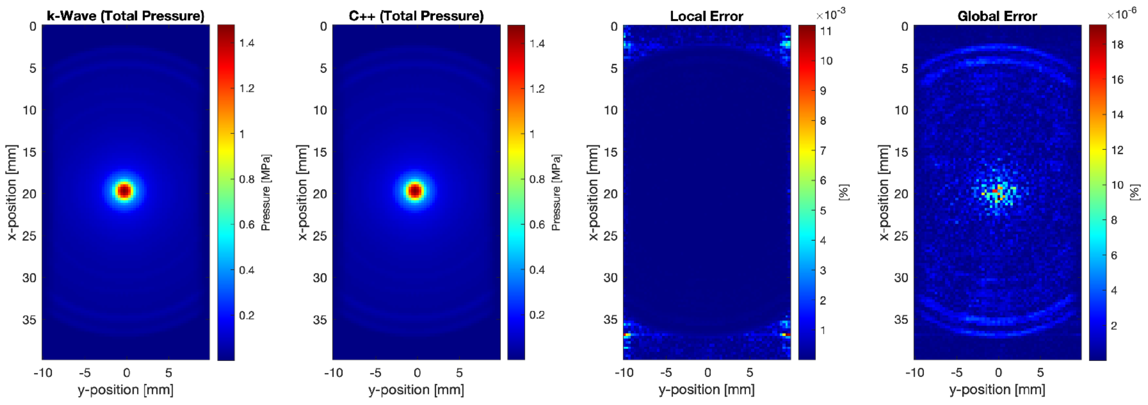


Figure 5.1: kWaveTester output for p0 source

Figure 5.1 shows the output of the kWaveTester for a simulation using a single p0 source. Execution in k-Wave-MATLAB took **30.55s**, execution in k-Wave-Python took **25.84s**. The total normalized error was `LINF=2.619e-07`.

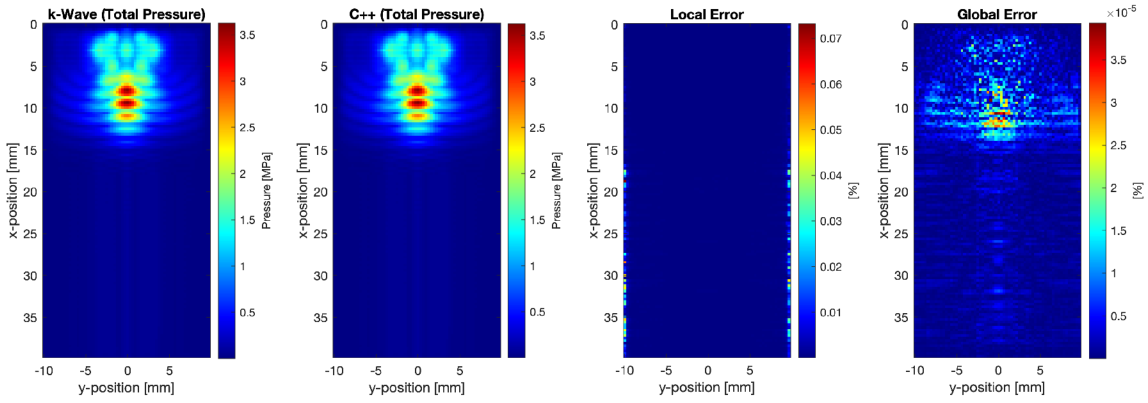


Figure 5.2: kWaveTester output for multiple p sources

Figure 5.2 shows the output of the kWaveTester for a simulation using multiple p sources. Execution in k-Wave-MATLAB took **34.86s**, execution in k-Wave-Python took **26.44s**. The total normalized error was `LINF=8.4321e-07`.

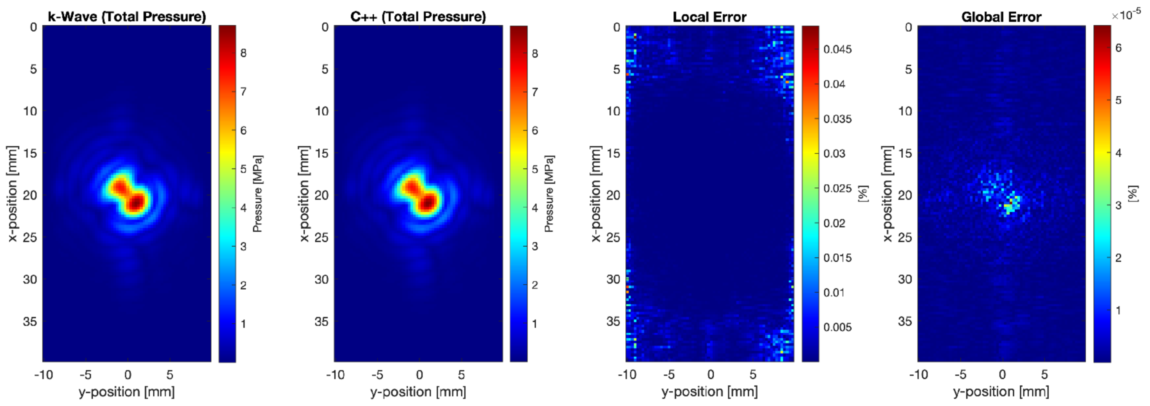


Figure 5.3: kWaveTester output for multiple u sources

Figure 5.3 shows the output of the kWaveTester for a simulation using multiple u sources. Execution in k-Wave-MATLAB took **30.08s**, execution in k-Wave-Python took **25.65s**. The total normalized error was `LINF=5.3346e-07`.

The results of the above described tests show, that the k-Wave-Python implementation is reasonably accurate and around **18%** faster for this configuration. There are some cases where calculated results are currently incorrect or cannot be tested because of unimplemented features.

5.2 Performance benchmarks

A performance comparison between k-Wave-MATLAB and k-Wave-Python was conducted using the kWaveTester for various domain sizes. The used version of MATLAB was R2021a, version of Python 3.10.4, NumPy 1.21.4, numexpr 2.8.1. The simulation settings used were 3D, non-linear, heterogenous, and absorbing. A simulation was executed for each medium shape using the kWaveTester. Duration of data preprocessing is not included in the performance measurements.

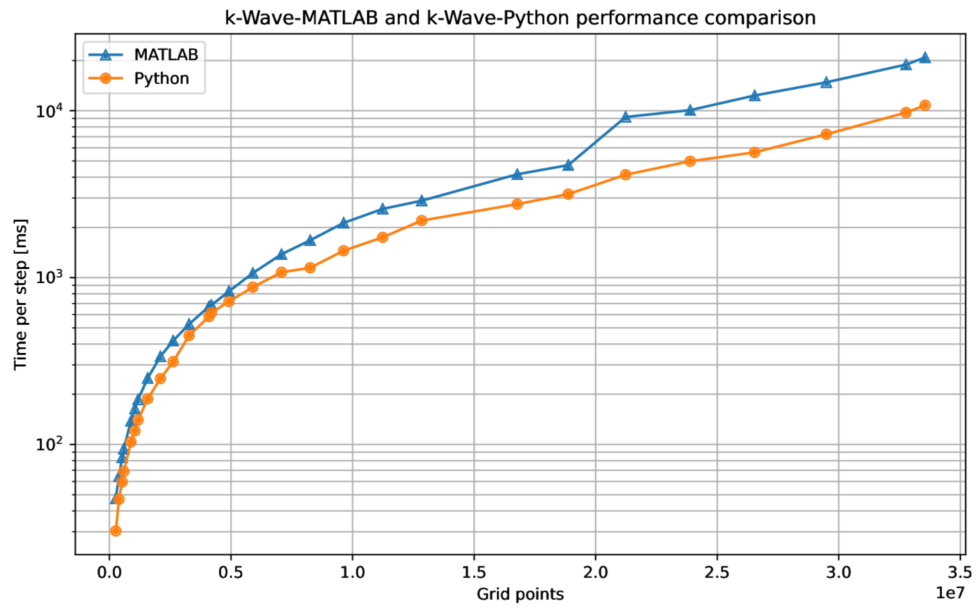


Figure 5.4: Time step duration comparison with domain sizes from 64x64x64 up to 512x256x256

Figure 5.4 shows the comparison of time step speeds based on measured data from appendix C. As can be seen, k-Wave-Python outperforms k-Wave MATLAB in every case. The average speedup over the MATLAB implementation is around **50%**, the biggest speedups were observed with domain sizes.

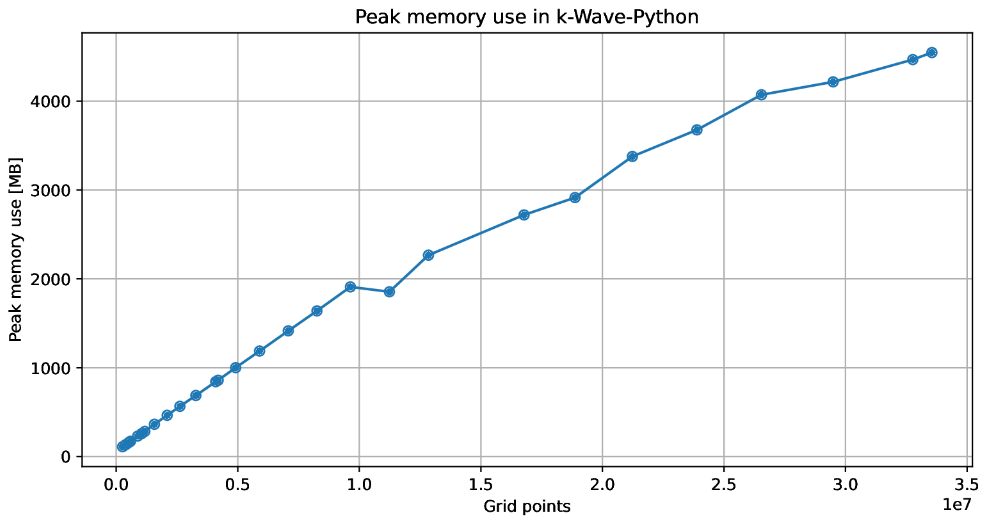


Figure 5.5: Peak memory usage in k-Wave-Python with domain sizes from 64x64x64 up to 512x256x256

Figure 5.5 shows the peak memory usage in k-Wave-Python, as measured during benchmarking. The recorded values are on average higher than reference values from the k-Wave manual [20, p. 80]. One of the reasons is the inherent overhead of the Python, the other reason is the usage of temporary arrays in some parts of k-Wave-Python. After the initial spike in memory use, the memory usage usually decreases, as seen in figures 5.4 and 5.5, meaning the recorded peak memory usage only occurs in a fraction of the total runtime.

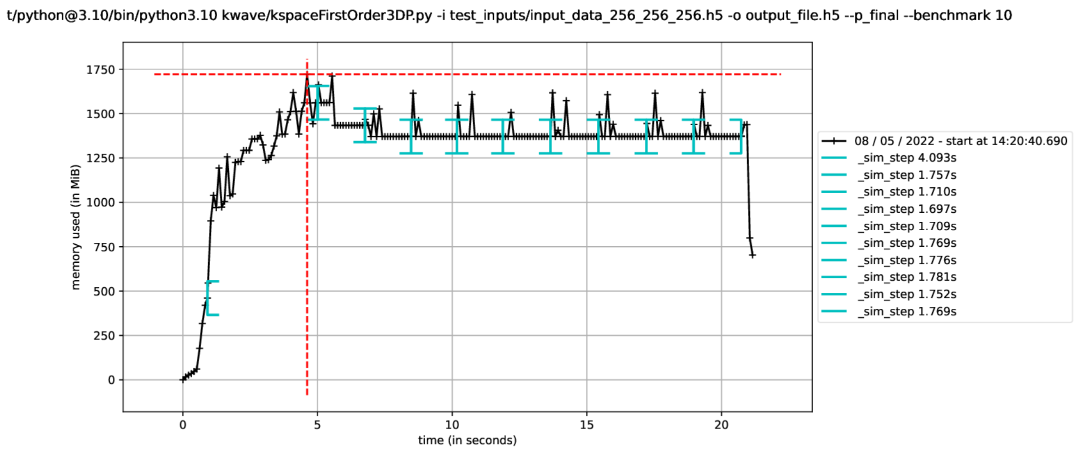
5.3 Memory profiling and optimization

Optimization of memory handling is closely tied to the overall performance of the simulation algorithm. Effects of inefficient memory handling might not be perceivable with smaller arrays, but as the domain size increases, every unnecessary allocation or reallocation decreases performance and might also make the program run out of memory. In low-level languages such as C and C++, the user has tight control over allocations that occur in the program and accidental allocation is less likely. However, with high-level garbage-collected languages such as Python, the situation is much less clear.

Even when using libraries such as NumPy that store arrays efficiently, close attention must be paid to the way a particular expression is written. Seemingly analogous implementations of an expression can often have very different performance characteristics. One of the common causes of unnecessary allocations is the incorrect usage of assignment: whereas the expression `y = x[:]` will create a new object for `y` in memory each time executed, `y[:] = x` will copy values of `x` without reallocating `y`. This style of writing code requires some changes in the way functions are written. For instance, functions should not return the computed values but instead copy them to an already existing array, which is passed to the function as an argument.

Memory profiling example

In this example, the effects of temporary variables on memory allocation are analyzed. A simulation with the dimensions of 256x256x256 grid points is used for demonstration of the effects of temporary variables.



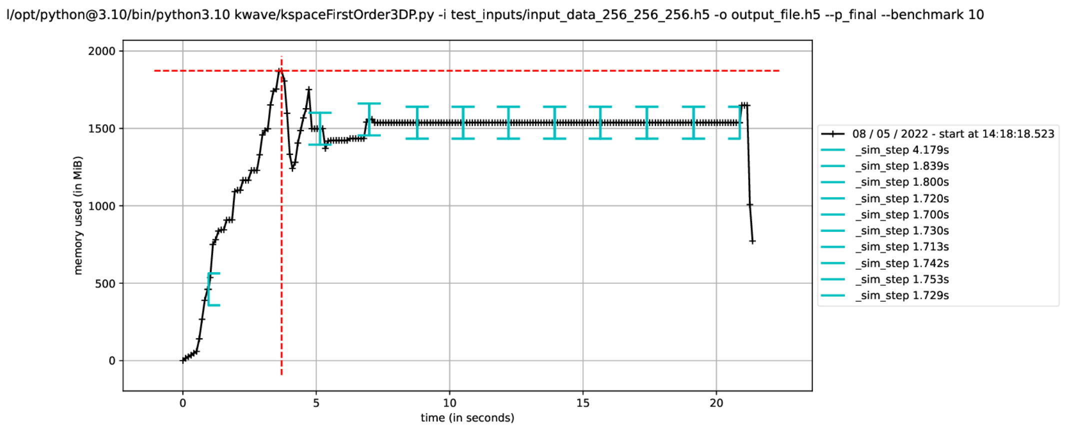
```
t/python@3.10/bin/python3.10 kwave/kspaceFirstOrder3DP.py -i test_inputs/input_data_256_256_256.h5 -o output_file.h5 --p_final --benchmark 10
```

Figure 5.6: Memory allocation with temporary array captured using `mprof`

Figure 5.6 displays total allocated memory as a function of time captured using the `mprof`[1] memory profiler. Functions decorated with the `@profile` decorator (such as `_sim_step` in this example) are highlighted in blue. The red dashed cross highlights the time of peak memory consumption.

In k-Wave-Python, the first simulation step usually creates a large spike in memory allocation, because all temporary arrays and libraries are being initialized. Peak memory consumption cannot be easily compared with languages such as C++ because the user does not always have full control over allocation and de-allocation.

As can be seen in figure 5.6, a pattern of periodic spikes in memory consumption can be observed during each time step after the initial memory allocations. This pattern indicates, that temporary arrays are being created during calls to certain functions. This is not desirable because the allocation of memory becomes less predictable.



```
l/opt/python@3.10/bin/python3.10 kwave/kspaceFirstOrder3DP.py -i test_inputs/input_data_256_256_256.h5 -o output_file.h5 --p_final --benchmark 10
```

Figure 5.7: Memory allocation with a pre-allocated array captured using `mprof`

Figure 5.7 displays memory allocation after a pre-allocated array for temporary results was introduced. Because temporary data is written to the pre-allocated array instead of creating new arrays, the spikes in memory consumption from the previous example disappear. This memory consumption of the program becomes more predictable and less time is consumed by allocation.

[1] https://pypi.org/project/memory-profiler/

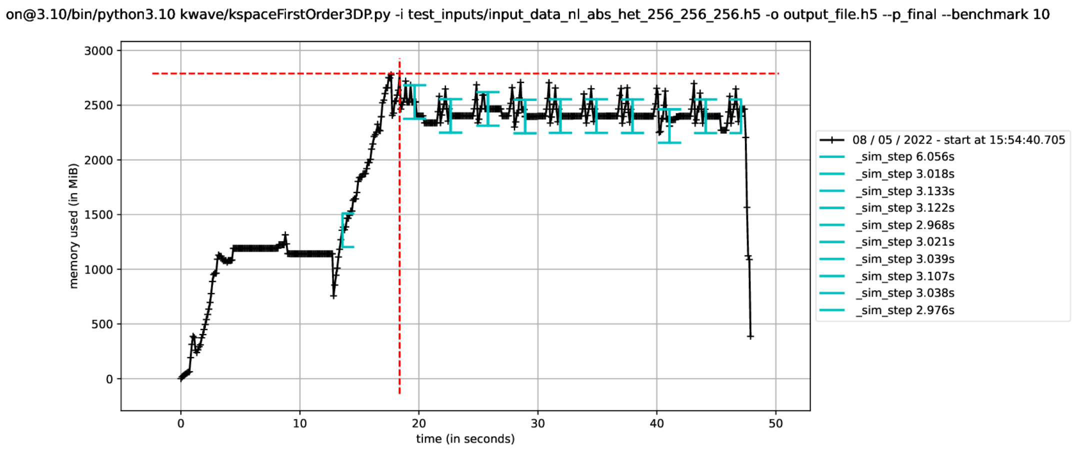


Figure 5.8: Memory allocation with non-linear, heterogenous, and absorbing settings captured using `mprof`

Figure 5.8 shows an example of a non-linear, heterogenous, and absorbing simulation. Even though the numexpr library is being used, it sometimes is not possible to avoid temporary allocations. Even if temporary arrays can be avoided, it does not always guarantee an increase in speed. Forcefully rewriting expressions to avoid reallocation may make the calculations less efficient, thus slowing the program down.

Analysis of bottlenecks

Multiple profiling tools were used during optimization of the implementation. The performance profiling tools `cProfile`[2] and `py-spy`[3] provide an aggregate overview of the most performance intensive function calls, they are mainly used to identify the general location of a bottleneck. Once an approximate location of the bottleneck is found, a line profiler can be used to locate the exact lines of code causing the slowdown. The `kernprof/line_profiler`[4] profiler was used for measuring the percentage of time spent in individual functions. In this section, a 128x128x128 simulation was run for 1000 time steps with `py-spy` attached, generating figures described below.

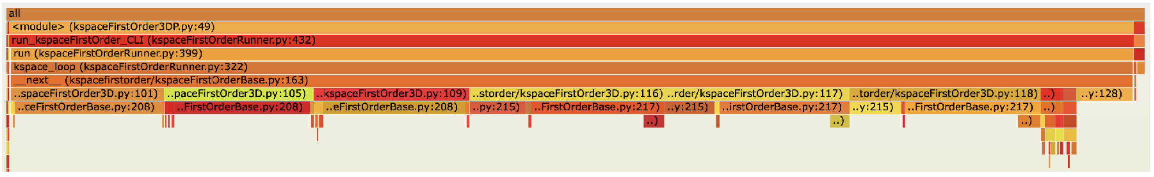


Figure 5.9: Performance profile of a linear, lossless, and homogenous simulation captured using `py-spy`

Figure 5.9 displays a *flame chart* that breaks down percent of runtime spent in each function of the call stack. More than 90% of the runtime is spent in the `kspace_runner`, which

[2]https://docs.python.org/3/library/profile.html
[3]https://github.com/benfred/py-spy
[4]https://github.com/pyutils/line_profiler

executes the main simulation loop and all related behavior such as data recording and visualization. The runtime is evenly spread between the individual simulation functions, which is to be expected. The simulation function `kspaceFirstOrder3D._sim_step()` was profiled using the `line_profiler` tool for 1000 time steps and the domain size of 128x128x128. Simulation settings used were linear, homogenous, and lossless. Around **45%** of computation time was spent in the function `calc_duxdx()`, **43%** in `calc_ux_sgx()`, **6%** of time was spent in `recalculate_rho()`. With the selected settings, the computation time is roughly divided by the number of FFTs computed.

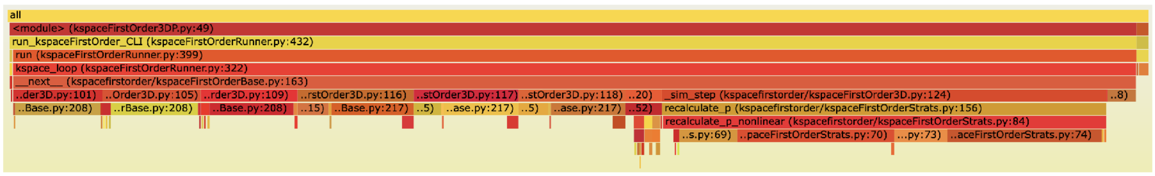


Figure 5.10: Performance profile of a non-linear, absorbing, and heterogenous simulation captured using `py-spy`

With non-linear, absorbing, and heterogenous simulations, the profile changes considerably. A large portion of runtime, as shown in figure 5.10, is spent in the function `recalculate_p()`, that handles the updates of the pressure array. The function causing the slowdown in this case is the `EosAbsorbing.recalculate_p_nonlinear()`. It contains four FFT calculations (`fftn()` and `ifftn()`) and is also difficult to optimize using numexpr. Similarly to the first example, the `kspaceFirstOrder3D._sim_step()` function was profiled using the `line_profiler` with the same domain size and number of time steps. The settings used were non-linear, heterogenous and absorbing. This time, the function `recalculate_p()` consumed almost **36%** of the computation time, **29%** of time was spent in `calc_duxdx()`, and around **27.5%** in `calc_ux_sgx()`. It can thus be concluded, that the calculation of FFTs has the most significant impact on the overall performance of the simulation.

Process of optimization

This section outlines the approximate steps taken during the optimization of k-Wave-Python. In the first optimization phase, unnecessary reallocations of NumPy arrays were eliminated where possible. Reallocations are usually unintentionally caused by assigning a result of a calculation to an variable pointing to an already existing array, this makes the garbage collector destroy the object and a reallocation occurs. Reallocations can be eliminated by assigning a new value to an already existing array like `x[:]  = result` or by using `np.copyto(x, result)`. Any such optimizations should first be verified since copying might not always be faster than creating a new array.

In the second phase, all equations, that can benefit from it, were wrapped with numexpr. This mainly applies to equations for recalculating pressure and density, which contain a large number of arithmetic operations. Equations that contain FFTs which cannot be included into numexpr can be optimized this way at least partially.

Since FFTs form a large majority of the computation time, as described in section 5.3, the performance is ultimately tied to the performance of the underlying FFT library (in this case pyFFTW).

Further avenues for optimization

One of the ways better performance could be achieved is by using various Python libraries, that enable *ahead-of-time*, *just-in-time* compilation, or improve parallelism of the computations.

Ahead-of-time compiled functions provide speeds comparable to C, this can be done by writing functions in *Cython*[5] as separate modules and compiling them. Compiling to C might not always equate to a speed boost. Since k-Wave-Python uses the NumPy and numexpr libraries, which are already vectorized, the benefit of compiling might be negligible [4, p. 162].

Just-in-time compiled solutions are easier to deploy to the target program. The `Numba`[6] library allows compilation of individual functions using a decorator. Compilation occurs during the first call of the functions, the JIT compiled code is then cached for further use. Limitations are however similar to the previously described pre-compiled Cython. Further deployment of numexpr, which could also be classified as a JIT library, is also an option.

While pyFFTW automatically works in multiple threads, some parts of the simulation algorithm could still benefit from added parallelization. This mainly applies to equations that are calculated for each dimension of the array separately. This includes velocity and density calculations for each cardinal direction `X`, `Y`, and `Z`. Since the Python *GIL* (Global Interpreter Lock) prohibits parallel computation using threads [9], separate processes must be used instead of threads. This can be achieved using the `multiprocessing` standard library, which allows the simple creation of worker pools. There are multiple caveats to this approach. For smaller domains, the overhead of forking and joining threads might outweigh the possible speed benefits. For larger domains, the cost of synchronizing results between threads might also be an issue [4, p. 280]. Another issue could be the duplication of allocated arrays during forking, a possible solution is to allocate shared NumPy arrays using `multiprocessing.Array` to avoid duplication of the memory from the main thread [4, p. 298].

One of the components which is currently not optimized is the class `HDFIndexingAdapter` (sec. 3.8) used for converting indices from HDF to NumPy. The slowdown is caused by the indexing conversion during runtime, which often needs to re-shape the target data.

While the k-Wave-Python performance was also benchmarked on the *Barbora* cluster from IT4Innovations [6] with Python 3.9.6 (and other versions) and MATLAB R2021a, the performance measured indicates, that locally measured speed improvements over the k-Wave-MATLAB do not translate to computational clusters. One of the further avenues of development could therefore be the adaptation of k-Wave-Python take advantage of cluster computing.

[5]https://cython.org/
[6]https://numba.pydata.org/

Chapter 6

Conclusion

The goal of this thesis was to analyze conversion of MATLAB programs to Python and to summarize the main issues arising from such conversions. The topics discussed mainly relate to the issues encountered during the re-implementation of the k-Wave toolbox, emphasizing the aspect of performance. The second goal was to create a Python implementation of a subset of the k-Wave toolbox, thus laying the groundwork for further development.

As a part of this thesis, a working Python k-Wave implementation was developed. The performance of the new implementation is comparable to the original MATLAB implementation, in some cases, the new implementation surpasses the performance of the original implementation by around **10-50%**. The total number of lines of code is approximately **2000 lines** (not including empty lines and comments), which is a substantial improvement over C++ implementations.

Based on the insights gained during implementation and conversion from MATLAB, a set of guidelines and tips was proposed. A summary of high performance computing technologies used in Python was also compiled.

The takeaway from this thesis is, that optimization is a non-trivial multi-faceted issue. Proper optimization requires due diligence and careful balancing of counteracting effects.

Currently, the performance of the implementation is limited by a small portion of the total lines of code, mainly due to FFT computations. In terms of optimization, acceleration of the most performance-critical simulation functions could provide substantial performance improvements. Function calls to FFTs currently prevent the application of numexpr to entire equations, bridging this gap could lead to improved memory handling. Performance on computer clusters could also be the focus of further development.

In the future, the implementation could be expanded to include 1D and 2D simulations. Due to the popularity of Python, k-Wave could also be brought to a wider audience of potential users while also enabling integration with other powerful Python libraries.

Bibliography

[1] Eurostat. *Root mean square error (RMSE)* [online], 9. may 2019 [cit. 2022-05-09]. Available at: https://ec.europa.eu/eurostat/cros/content/root-mean-square-error-rmse_en.

[2] fftw.org. *Planner Flags* [online]. [cit. 2022-05-06]. Available at: https://www.fftw.org/fftw3_doc/Planner-Flags.html.

[3] Gomersall, H. *pyFFTW Documentation rev.86df872* [online]. 2021-12-27 [cit. 2022-01-16]. Available at: https://pyfftw.readthedocs.io.

[4] Gorelick, M. and Ozsvald, I. *High Performance Python*. 2nd ed. Sebastopol: O'Reilly Media, Inc., may 2020. ISBN 9781492055020.

[5] Hunt, A. and Thomas, D. *The Pragmatic Programmer*. 1st ed. Addison-Wesley, 2000. ISBN 0-201-61622-X.

[6] It4Innovations. *Barbora* [online]. [cit. 2022-05-10]. Available at: https://www.it4i.cz/en/infrastructure/barbora.

[7] Moler, C., Eddins, S. and MathWorks. *Faster Finite Fourier Transforms MATLAB* [online]. [cit. 2022-01-15]. Available at: https://www.mathworks.com/company/newsletters/articles/faster-finite-fourier-transforms-matlab.html.

[8] NumPy Developers. *NumPy for MATLAB users* [online]. [cit. 2022-01-15]. Available at: https://numpy.org/doc/1.22/user/numpy-for-matlab-users.html.

[9] Python Software Foundation. *GlobalInterpreterLock* [online]. [cit. 2022-05-06]. Available at: https://wiki.python.org/moin/GlobalInterpreterLock.

[10] Python Software Foundation. *Python Package Index* [online]. [cit. 2022-01-15]. Available at: https://pypi.org.

[11] Robertson, J. L., Cox, B. T., Jaros, J. and Treeby, B. E. Accurate simulation of transcranial ultrasound propagation for ultrasonic neuromodulation and stimulation. *J. Acoust. Soc. Am.* 2017, vol. 141, no. 3, p. 1726–1738. DOI: 10.1121/1.4976339. Available at: http://bug.medphys.ucl.ac.uk/papers/2017-Robertson-JASA.pdf.

[12] Rossum, G. van. Glue It All Together With Python. In: Thompson, C., ed. *Workshop on Compositional Software Architecture* [online]. [cit. 2022-01-15]. Available at: http://www.objs.com/workshops/ws9801/papers/paper070.html.

[13] Shvets, A. *Dive Into Design Patterns* [online]. V2021-2.32th ed. 2021 [cit. 2022-05-03]. Available at: https://refactoring.guru/design-patterns/book.

[14] Stenson, I. *Faster Fast Fourier Transforms in Python* [online], 11. june 2021 [cit. 2022-05-09]. Available at: https://blog.hpc.qmul.ac.uk/pyfftw.html.

[15] Suomi, V., Treeby, B. E., Jaros, J., Makela, P., Anttinen, M. et al. Transurethral ultrasound therapy of the prostate in the presence of calcifications: A simulation study. *Med. Phys.* 2018, vol. 45, no. 11, p. 4793–4805. DOI: 10.1002/mp.13183. Available at: http://bug.medphys.ucl.ac.uk/papers/2018-Suomi-MP.pdf.

[16] The MathWorks, Inc.. *Fft* [online]. [cit. 2022-05-07]. Available at: https://www.mathworks.com/help/matlab/ref/fft.html.

[17] The MathWorks, Inc.. *MATLAB Performance* [online]. [cit. 2022-05-09]. Available at: https://www.mathworks.com/products/matlab/performance.html.

[18] The MathWorks, Inc.. *MATLAB vs. Python: Top Reasons to Choose MATLAB* [online]. [cit. 2022-01-15]. Available at: https://www.mathworks.com/products/matlab/matlab-vs-python.html.

[19] The MathWorks, Inc.. *Row-Major and Column-Major Array Layouts* [online]. [cit. 2022-01-15]. Available at: https://www.mathworks.com/help/coder/ug/what-are-column-major-and-row-major-representation-1.html.

[20] Treeby, B., Cox, B. and Jaros, J. *k-Wave - User Manual* [online]. August 2016, 2016-08-27 [cit. 2022-01-17]. Available at: http://www.k-wave.org/manual/k-wave_user_manual_1.1.pdf.

[21] Treeby, B. E. and Cox, B. T. k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields. [online]. SPIE. vol. 15, no. 2, p. 1 – 12, [cit. 2022-01-15]. DOI: 10.1117/1.3360308.

[22] Treeby, E. B., Jaroš, J., Rendell, P. A. and Cox, T. B. Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using a k-space pseudospectral method. *Journal of the Acoustical Society of America*. 2012, vol. 131, no. 6, p. 4324–4336. DOI: 10.1121/1.4712021. ISSN 1520-8524. Available at: https://www.fit.vut.cz/research/publication/10069.

Appendix A

Contents of the included storage media

- `k-Wave-MATLAB/` - folder containing source code of the k-Wave-MATLAB toolbox which served as the reference during implementation of k-Wave-Python
- `Sources/` - folder containing source files of the k-Wave-Python implementation
- `Thesis/` - folder containing source files of the thesis text
- `Data/` - folder containing example data, measurements and simulation logs
- `README.md` - file describing contents of the storage media
- `xcerny74.pdf` - PDF version of the thesis text

Appendix B

k-Wave Python user manual

This appendix contains the guide for the installation and usage of k-Wave-Python. The basic requirements include:

- A UNIX-like operating system such as Linux or MacOS.
- Python[1] version 3.10, lower versions such as Python 3.9 may run, but full compatibility is not guaranteed.
- The Python libraries numpy[2], numexpr[3], matplotlib[4]. Although the simulator can run without the library pyFFTW[5], it is highly recommended because of performance benefits.
- If the pyFFTW library is used, the underlying FFTW3[6] library must also be installed on the target system.

Python libraries can be installed using the Python `pip` package management tool using the command `pip install -r requirements.txt`, the `requirements.txt` file contains above listed required libraries. The pip tool may also be called `pip3` depending on the configuration of the target system.

To simplify the installation process, k-Wave-Python can be directly installed using pip. This can be done by navigating to the root folder of the project and using the `pip install .` command. A new `kwave` python package will be installed system-wide. A new utility `kspaceFirstOrder3DP` will also be installed and can be used in any terminal. The default install only installs the NumPy FFT backend, to install the additional pyFFTW backend, use the command `pip install „.[fftw]"`. For more information, refer to `README.md`.

[1] https://www.python.org/
[2] https://numpy.org/
[3] https://github.com/pydata/numexpr
[4] https://matplotlib.org/
[5] https://github.com/pyFFTW/pyFFTW
[6] https://www.fftw.org/

B.1 Using the simulator

The command line interface of the simulator can be accessed using the file `kspaceFirstOrder3DP.py`. It can be executed using the commands: `python3 kspaceFirstOrder3DP.py` or `./kspaceFirstOrder3DP.py` on Linux/MacOS (this may require granting execution permissions using `chmod`).

To use the simulator with k-Wave-MATLAB during development, create a hard link of the CLI script using `ln kspaceFirstOrder3DP.py kspaceFirstOrder3DP`, then move the `kspaceFirstOrder3DP` hard link to the `k-Wave-MATLAB/k-Wave/binaries` folder in the k-Wave-MATLAB. To ensure individual modules are correctly found, and add the root folder of k-Wave-Python to the `PATH` and `PYTHONPATH` environment variables.

To simplify the process of using k-Wave-Python within the k-Wave-MATLAB toolbox, the `link_kwave.py` utility was developed. After executing `python3 link_kwave.py path/to/kwave/binaries`, the utility will create a new hard link for the `kspaceFirstOrder3DP` in the specified binaries folder, permissions to execute the script will also be added. Additionally, the k-Wave-Python folder will be exported to PATH and PYTHONPATH. Because the environment variable export is not persistent, the script lists the shell commands that should be added to the configuration file of the terminal. The command `python3 kspaceFirstOrder3DP.py -i input_file.h5 -o output_file.h5` is the simplest example of usage of the command line interface, only the input and output files are defined. By default, the simulator will only record the `p_raw` field and log progress every 5% of time steps. The log output will resemble the example output in appendix E. All available parameters can be seen in appendix D.

Appendix C

Benchmark

Table C.1: Time step duration [ms] and maximum memory usage in k-Wave-Python [MB]

Domain size	MATLAB	Python	Memory [MB]
64x64x64	47.42	30.32	111
96x64x64	64.52	46.90	135
128x64x64	83.52	59.65	160
96x96x64	94.07	68.93	173
96x96x96	138.35	103.76	231
128x128x64	163.90	120.71	259
128x96x96	186.07	140.26	284
128x128x96	249.72	187.80	364
128x128x128	337.02	248.29	465
160x128x128	418.46	313.31	566
160x160x128	526.11	450.23	688
160x160x160	671.74	584.03	843
256x128x128	684.18	615.97	860
192x160x160	827.36	719.57	1001
192x192x160	1064.74	877.12	1188
192x192x192	1381.20	1077.28	1415
224x192x192	1674.80	1145.75	1640
224x224x192	2128.18	1450.44	1910
224x224x224	2582.44	1744.35	1855
256x224x224	2890.54	2197.79	2267
256x256x256	4164.30	2759.15	2720
288x256x256	4723.41	3166.60	2915
288x288x256	9180.95	4138.25	3378
288x288x288	10088.54	4983.87	3677
320x288x288	12347.96	5637.26	4072
320x320x288	14813.73	7234.01	4217
320x320x320	18922.59	9761.35	4468
512x256x256	20854.13	10784.86	4547

Appendix D

k-Wave-Python help menu

```
usage: kspaceFirstOrder3D [-h] -i <file_name> -o <file_name> [--version]
                          [-r <interval_in_%>] [--verbose <level>]
                          [--benchmark <time_steps>] [--show] [-b <backend>]
                          [-t <num_threads>] [-s <time_step>]
                          [--checkpoint_interval <sec>]
                          [--checkpoint_file <file_name>] [-p] [--p_rms]
                          [--p_max] [--p_min] [--p_max_all] [--p_min_all]
                          [--p_final] [-u] [--u_rms] [--u_max] [--u_min]
                          [--u_max_all] [--u_min_all] [--u_final]
                          [--copy_sensor_mask]

kspaceFirstOrder3D launcher script. This script can be used to run
kspaceFirstOrder3D simulations using HDF input files. Simulation dimensions
and settings are automatically loaded from the input dataset. Other available
options are described below.

options:
  -h, --help            show this help message and exit

mandatory parameters:
  -i <file_name>        name of HDF5 input file
  -o <file_name>        name of HDF5 output file

optional parameters:
  --version             print version and build info
  -r <interval_in_%>    progress print interval (default = 5%)
  --verbose <level>     level of verbosity <0, 2> (default = 1)
  --benchmark <time_steps>
                        run only a specified number of time steps
  -t <num_threads>      number of CPU threads for FFT (default = 8)
  -s <time_step>        time step when data collection begins (default = 0)
  --checkpoint_interval <sec>
                        checkpoint after a given number of seconds (default =
                        60)
  --checkpoint_file <file_name>
                        name of HDF5 checkpoint file

implementation specific parameters:
  --show                will display simulation progress preview
  -b <backend>, --backend <backend>
                        backend used for FFT computations (default = pyfftw)

output flags:
  -p, --p_raw           store time varying acoustic pressure
  --p_rms               store rms of P
  --p_max               store max of P
  --p_min               store min of P
  --p_max_all           store max of P (whole domain)
  --p_min_all           store min of P (whole domain)
  --p_final             store final pressure field
  -u, --u_raw           store time varying particle velocity (ux, uy, uz)
  --u_rms               store rms of ux, uy, uz
  --u_max               store max of ux, uy, uz
  --u_min               store min of ux, uy, uz
  --u_max_all           store max of ux, uy, uz (whole domain)
  --u_min_all           store min of ux, uy, uz (whole domain)
  --u_final             store final particle velocity field
  --copy_sensor_mask    copy sensor mask to output file
```

Appendix E

Example output log

```
+---------------------------------------------------------------+
|                 kSpaceFirstOrder3D-Python v0.1                |
+---------------------------------------------------------------+
| Number of CPU threads:                                      8 |
| Reading simulation configuration:                        Done |
+---------------------------------------------------------------+
|                      Simulation details                       |
+---------------------------------------------------------------+
| Simulation dimensions:                       128 x 128 x 128 |
| Simulation time steps:                                     50 |
+---------------------------------------------------------------+
|                        Initialization                         |
+---------------------------------------------------------------+
| Memory allocation:                                       Done |
| Data loading:                                            Done |
| Elapsed time:                                           0.08s |
+---------------------------------------------------------------+
| FFT plans creation:                                      Done |
| Pre-processing phase:                                    Done |
| Elapsed time:                                           0.08s |
+---------------------------------------------------------------+
|                    Computational resources                    |
| Current host memory in use:                             240MB |
+---------------------------------------------------------------+
|                          Simulation                           |
+----------+----------------+--------------+--------------------+
| Progress |  Elapsed time  |  Time to go  |  Est. finish time  |
+----------+----------------+--------------+--------------------+
|       4% |         0.464s |       22.29s |  06/05/22 11:58:43 |
|       8% |         1.047s |      16.048s |  06/05/22 11:58:37 |
|      12% |         1.581s |      13.913s |  06/05/22 11:58:35 |
|      16% |         2.121s |      12.725s |  06/05/22 11:58:35 |
|      20% |         2.646s |       11.76s |  06/05/22 11:58:34 |
|      24% |         3.167s |       10.94s |  06/05/22 11:58:34 |
|      28% |          3.692s |       10.225s |  06/05/22 11:58:34 |
|      32% |          4.223s |        9.571s |  06/05/22 11:58:34 |
|      36% |          4.767s |        8.973s |  06/05/22 11:58:33 |
|      40% |          5.303s |        8.374s |  06/05/22 11:58:33 |
|      44% |          5.842s |         7.79s |  06/05/22 11:58:33 |
|      48% |          6.382s |        7.214s |  06/05/22 11:58:33 |
|      52% |          6.913s |        6.637s |  06/05/22 11:58:33 |
|      56% |          7.441s |        6.063s |  06/05/22 11:58:33 |
|      60% |          7.958s |        5.488s |  06/05/22 11:58:33 |
|      64% |          8.475s |        4.921s |  06/05/22 11:58:33 |
|      68% |          8.993s |         4.36s |  06/05/22 11:58:33 |
|      72% |          9.508s |        3.803s |  06/05/22 11:58:33 |
|      76% |         10.034s |        3.254s |  06/05/22 11:58:33 |
|      80% |         10.564s |        2.709s |  06/05/22 11:58:33 |
|      84% |         11.087s |        2.163s |  06/05/22 11:58:33 |
|      88% |         11.615s |        1.621s |  06/05/22 11:58:33 |
|      92% |         12.145s |         1.08s |  06/05/22 11:58:33 |
|      96% |         12.673s |        0.539s |  06/05/22 11:58:33 |
|      98% |         12.936s |         0.27s |  06/05/22 11:58:33 |
+----------+----------------+--------------+--------------------+
| Elapsed time:                                            13.47s |
+----------------------------------------------------------------+
|                          Summary                               |
+----------------------------------------------------------------+
| Peak memory in use:                                       505MB |
+----------------------------------------------------------------+
| Total execution time:                                  13.8536s |
+----------------------------------------------------------------+
|                     End of computation                         |
+----------------------------------------------------------------+
```