The University of Auckland

BTECH 451 Project Report

Resonant Ultrasound Spectroscopy

Author:
En Yung Hoang

Supervisors:
Dr. Kasper Van Wijk
Paul Freeman
Dr. Sathiamoorthy Manoharan

June 2016

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

Abstract

This report contains and describes all the information related to my project up to June 2016. My project is on Resonant Ultrasound Spectroscopy (RUS), this is a physics procedure used to determine the material properties of solid objects. The formulas and calculations used in RUS have been written as computer algorithms that was originally written in C programming Language by Jerome H.L. Le Rousseau. His code was later updated by Leighton Watson, a Physics student here at the University of Auckland, as part of his honours project. The original C code was later translated to Python by Paul Freeman for ease of use.

After the translation into Python, the performance of the RUS algorithm has reduced significantly and my project is to use Cython to wrap C functions and/ or rewrite the Python implementation to increase its performance to speeds that are fairly close to the C implementation.

The first section of the report will briefly outline what RUS is and how it works with illustrative diagrams for better understanding. The latter sections will look at the RUS algorithm and its implementations. These include an explanation of why the C implementation is slow, the tool that I will be using to increase the performance of the Python implementation, alternatives to Cython and why I chose Cython, some examples and common uses of Cython, an example of how Cython works, Python profilers, a comparison between C, Python and Cython, and finally, some reasons why Python is slower than C.

Towards the end of this report, I have included what I have done so far for this project and some plans and goals for the latter half of this year-long project.

1 Acknowledgments

I would like to thank my academic supervisors and course coordinator for the continuous guidance, support and feedback throughout this project. I really appreciate your time and hard work.

Dr. Kasper Van Wijk

Academic Supervisor
The University of Auckland
Physics Department

Paul Freeman

Academic Supervisor
The University of Auckland

Dr. Sathiamoorthy Manoharan

Information Technology Coordinator
The University of Auckland

Contents

2 Introduction

Resonant Ultrasound Spectroscopy (RUS) consists of two problems; these include the forward and inverse problems. For the remainder of this report, I will refer to these problems as algorithms. The forward algorithm is the main focus for this project, it is a technique that measures the elastic properties of solid objects to estimate the corresponding resonant modes. With the use of these resonant modes, it is possible to determine the material properties of solid objects. Computer algorithms have been written to quickly convert between the elastic properties and their corresponding resonant modes. This algorithm was originally written in C programming language but was later rewritten using Python programming language due to the original code relying heavily on outdated versions of software libraries or libraries that were difficult to obtain and install. The only known support for the C version is on a Linux based operating system with the requirement of several other libraries that has to be installed locally. Python helped extensively with this because Python comes with a large collection of libraries which can be easily installed locally and is highly portable. The only disadvantage with using Python is its performance, it's significant slower than the C implementation.

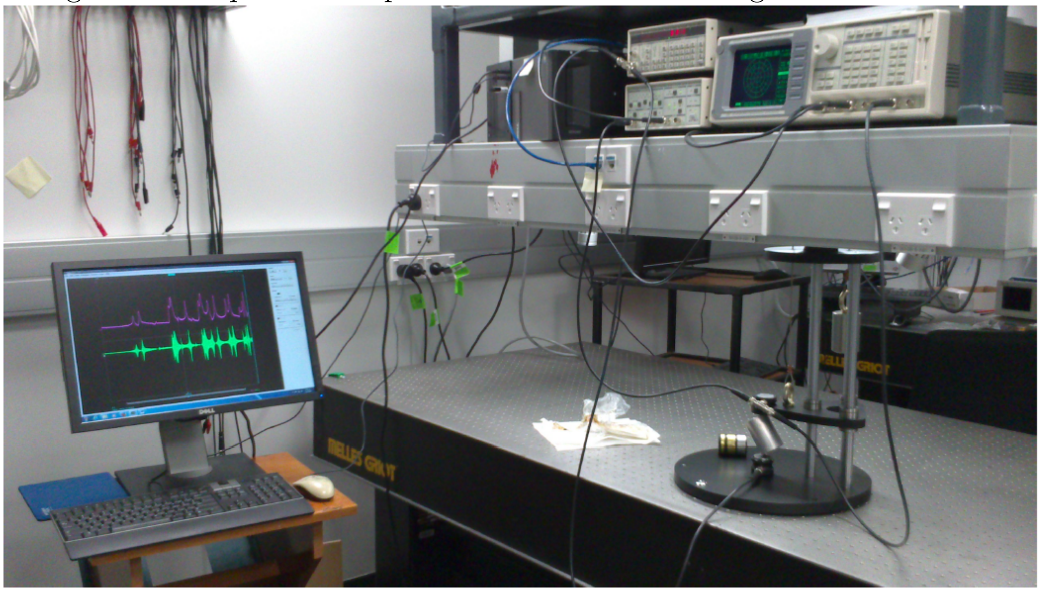
This report will document the improvements in speed of the Python implementation of the Forward-RUS algorithm by wrapping C and/ or rewriting functions within Python. This will be achieved with the use of Cython. In most cases, it will be necessary to rewrite sections of the code with a combination of both Python and C.

3 Resonant Ultrasound Spectroscopy (RUS)

3.1 What is Resonant Ultrasound Spectroscopy?

Resonant Ultrasound Spectroscopy (RUS) is a procedure used to determine the material properties of solid objects. This procedure consists of the study of mechanical resonances of solid objects. The output consists of a range of resonant modes, which are dependent and determined by the size, shape, elastic properties, crystallographic orientation and density of the solid object[1].

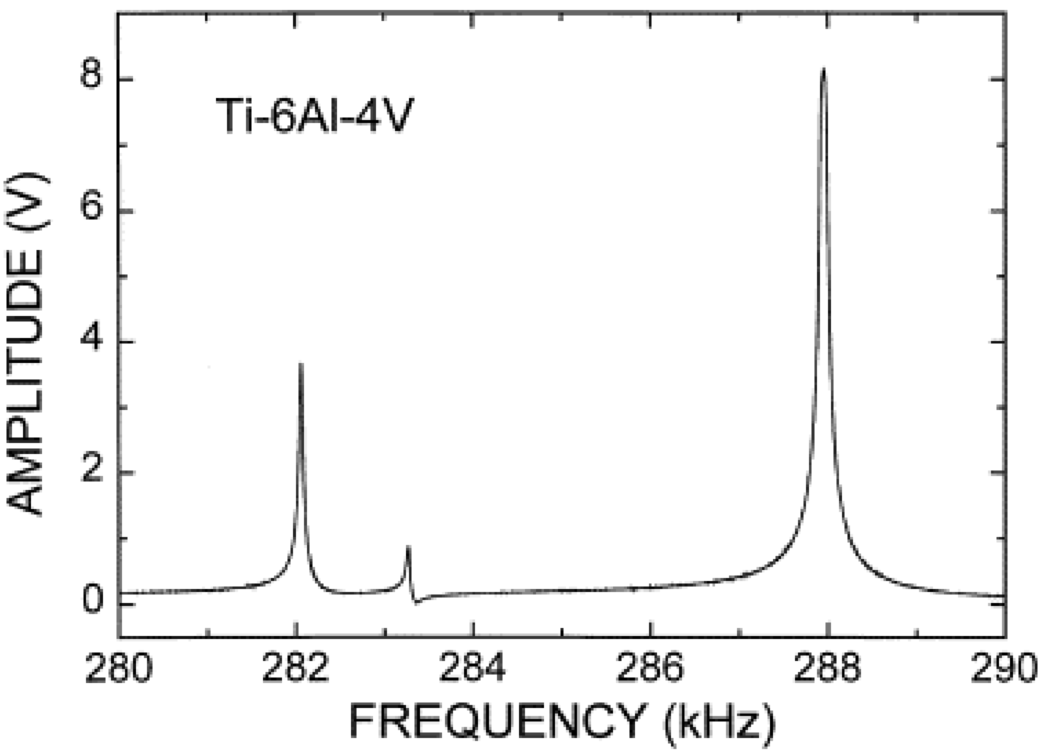
The general setup of this experiment is shown in the figure below[2]:



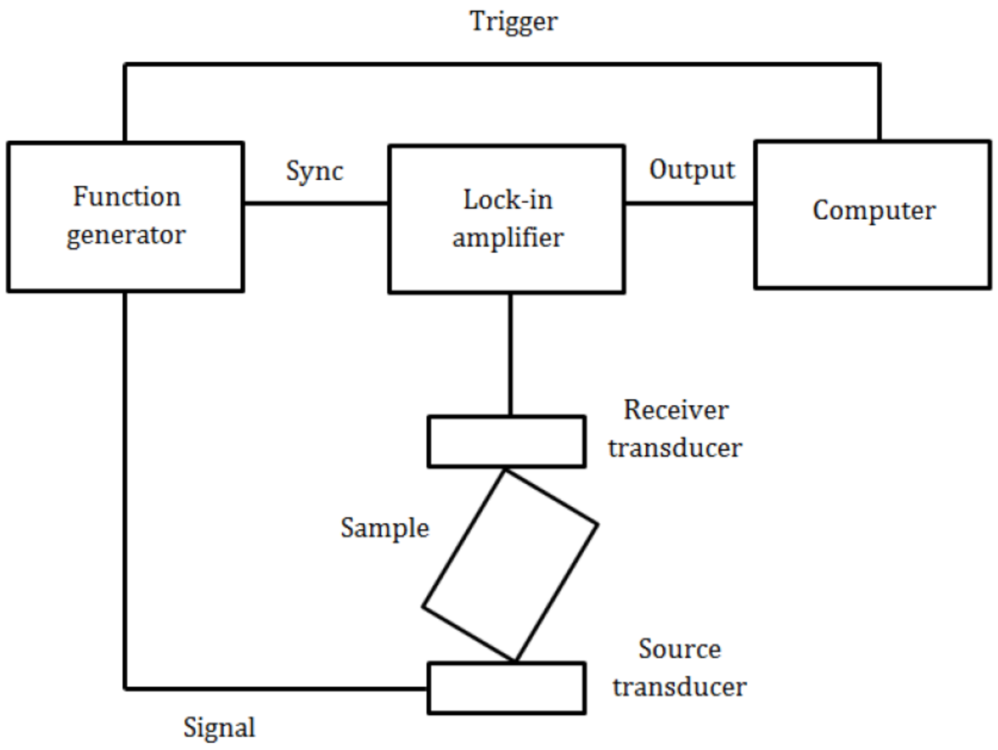
Several equipments are required for this setup, these include[2]:

- Transducers - Two transducers are required, one for the source of varying frequencies and the other for measuring the response from the solid object
- Function Generator - This is used to generate the varying frequencies that are sent through the source transducer
- Amplifier - this is used for amplifying and filtering the response from the solid object

The solid object, which is usually a parallelepiped or a spherical shape, is held tightly between two piezoelectric transducers. The source transducer then excites the object by producing and sending a wide range of elastic waves of constant amplitude and varying frequencies through the solid object. The receiver transducer then detects the wide range of resulting frequencies, but what we are interested are the resonant peaks. The resonant peaks occur due to the fact that solid objects vibrate at their natural frequencies, as soon as the range of frequencies are sent through the object, the frequencies that match the natural frequencies of the object will produce the resonant peaks. A sample of a typical output showing the resulting frequencies are shown below[3]:



A better illustration of what happens during this experiment and the equipments used are represented below[2]:



The resulting resonant peaks are recorded and used by computer algorithms that have been written and adapted/updated throughout the years. These computer algorithms are used to quickly convert between their elastic properties and their corresponding resonant modes[6]. The original implementation was written in C by Jerome H.L. Le Rousseau, it was later updated by Leighton Watson in 2014 and was translated into Python by Paul Freeman in 2015.

4 Installation of the C Implementation

The C implementation is difficult to install and can only be installed on specific operating system, mainly on a distribution of Linux. The C implementation is dependent on outdated versions of software libraries or libraries that are difficult to obtain and install. It involves a lot of compiling of various tools and libraries on each local system before the RUS code can be installed and executed[6].

There are a lot of necessary files that need to be installed and steps to take for the C implementation. The main files that need to be installed are:

- rusGuiScilab.tar.gz
- rusGuiMatlab.tar.gz
- forinv.tar.gz

These need to be moved to a specific directory and extracted. The installation of the above files require a lot of different steps and a lot of thorough testing to ensure that everything is installed correctly and is functioning. Along with installing those files, Scilab and PlotLib are also required and will need to be installed as suggested by the installation methods.

The C code also has quite a few dependencies on outside libraries and packages, which will also need to be installed during the Seismic Unix installation stage. Seismic Unix is required because it contains the following core libraries:

- libsu
- libpar
- libcwp
- Makefile.config

It is necessary for all these to be installed before the C implementation can be executed.

There are many other steps and files that needs to be installed and configured other than the ones mentioned above. This is all mentioned in a 32 paged installation, testing and examples guide that is uploaded onto PALab's RUS GitHub page.

As you can see in the very brief outline of the installation process of the C implementation, the dependencies and the actual C code itself require many steps to run on a local system. This was the main reason for the translation to a Python implementation[7].

5 Cython

5.1 What is Cython?

Cython is an optimization static compiler that is used for both Python and C/C++ programming languages. It allows for the interchangeability of both Python and C code within one file. This file is known as the Pyrex file, with the extension .pyx. Cython can be considered a language on its own and is a super-set of Python with additional support for calling C functions and declaring C type variables and class attributes. The source code in the Pyrex file gets translated into an optimized C code when compiled, this optimized code will then be used as Python extension modules. This means that the Cython compiled code will produce a very efficient program and provides a close integration with external C libraries[5].

6 The limitations of Cython and how I will limit them

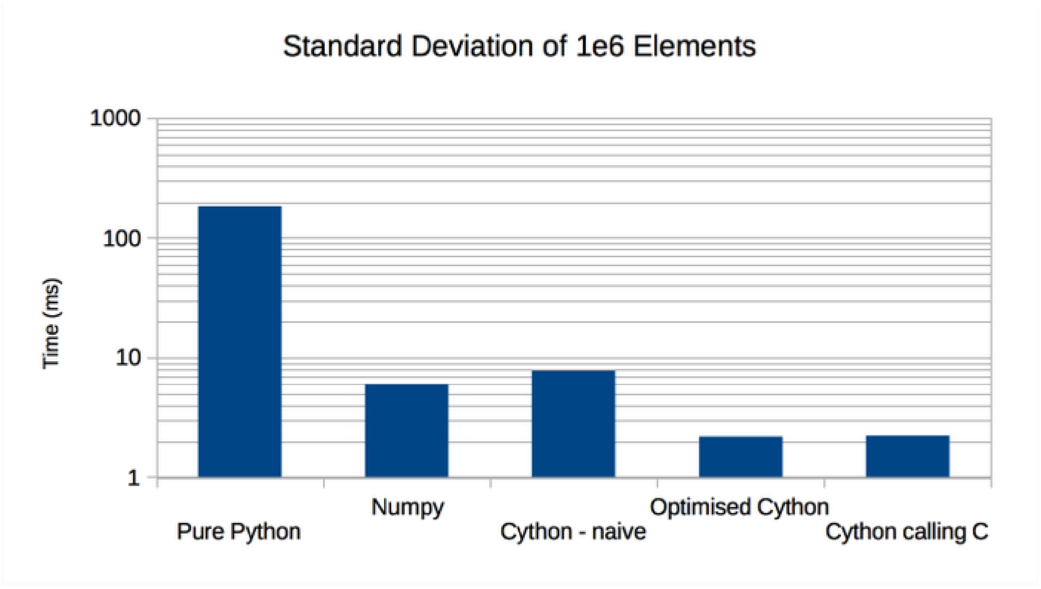
Cython has a few limitations, these include Cython's separate build phase and its compilation time. For my project, I am required to rewrite the Python code using Cython, wrap the C code within Python or just call external C codes directly. I decided to rewrite the code in a combination of C, Python and Cython because this approach reduces the compilation time. This reduces the compilation time because in situations when no changes in the C or Python code are made or when there is no new code added in, the compilation time would be near instant. This is because only the Cython code would need to be compiled which is very efficient because it's native to the Cython compiler. Also, code that was written in one language will not affect the compilation time for the code written in the other language because the compiler compiles the file based on the language that's last modified. For example, if I modify a code in cdef, the compiler will only compile the cdef functions and not def and cpdef. Another way of limiting this limitation is by using static type declarations, these are interpreted as valid Python and C code and are ignored by the Cython compiler at run time.[4]

6.1 Why Cython?

I chose Cython for my project because Cython provides a close integration with external C libraries, which is a very important part for my project because one of the main reasons that the original C code was translated into Python was because Python reduces the dependencies on hard to install C libraries. These libraries are usually hard to obtain, quite large and could only be installed onto specific operating systems, mainly Linux in this case[6]. Furthermore, Cython allows the combination of both C and Python source code within one file, allowing more flexibility and better implementation for the final product. Sometimes just calling the corresponding C functions are not sufficient because there could be libraries or implementations that are better in one language but worse in the other language. Cython also generates an executable, which I thought would be very useful for portability and ease of use for the end users. With the use of the executable, I would hopefully be able to allow the use of the Cython compiled program without the end users having to install Cython on their local machine.

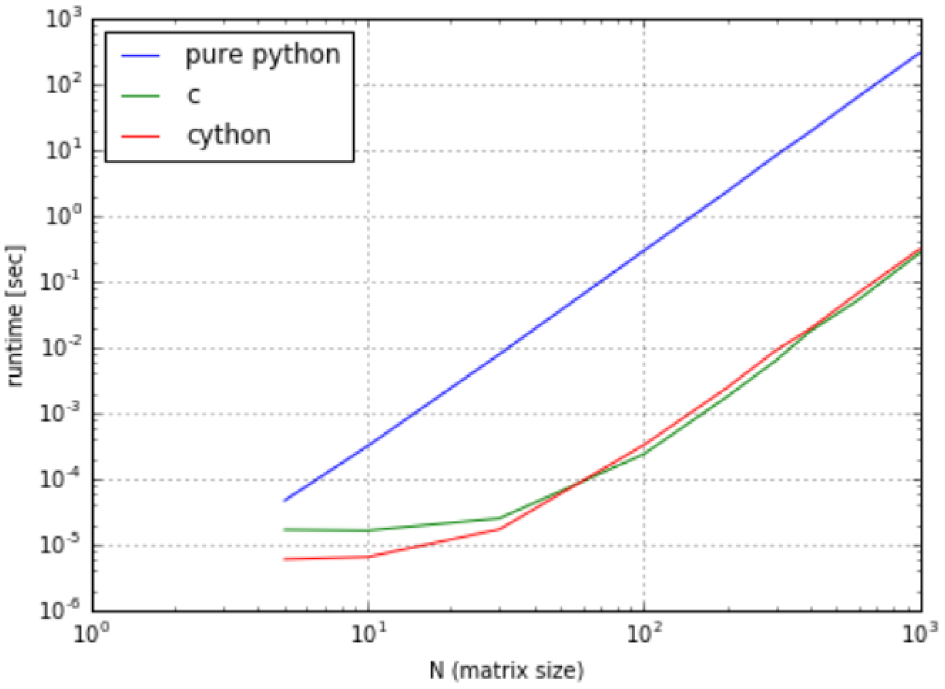
There are also other reasons why the C implementation was translated into Python. A reason is that Python is a good platform for scientific computations. There are two main reasons behind this, firstly, Python tends to be readable and very concise, making development of scientific computations very rapid. Secondly, Python allows access to its internals from C through the Python/C API. It has been found that Python is very inefficient when there are a lot of loops in the code, the main reason behind this is because of its dynamic nature. Cython solves this issue by compiling Python code directly to C, which is then compiled and linked to Python[4]. Also, due to its use of C static types, Cython is able to make numerous loops while running at C speeds, directly in Python code[5].

The wrapping process of Cython is all done manually via the Pyrex file. This allows for many different ways of implementing the source code. The Cython code can be done in a naive way by defining the static variable types, by rewriting the code in an optimized way and by purely calling C functions. Here is a graph that shows the performance of each of the different approaches[8]:



Due to the translation to the Python implementation, some of the source code have been implemented differently compared to the original C code. These include changes in variable names, changes in the use of arrays, the splitting of the different functions and many others. This means that simply calling the C functions is not always the best option. Thus, having the option of calling and manually implementing the code would give me a huge advantage in terms of implementing the most optimized code.

Another reason why I chose Cython is because of its performance. After wrapping the code using Cython, the performance of the final code would be very close to the performance of the original C implementation. Here is a diagram that shows Cython's performance compared to pure Python code:



6.2 Alternatives to Cython

I looked at several tools that would help achieve what I needed for this project. I did not look at other techniques to increase the performance of the Python code, other than wrappers, because my supervisor specifically wanted wrapper functions to wrap the original C code into Python. The tools that I looked at include:

SWIG

SWIG stands for Simplified Wrapper and Interface Generator, it performs very similar tasks as Cython but it is built for a variety of different high-level programming languages, which include both scripting and non-scripting languages. This means that it is not built purely for the use with C and Python. It is an interface compiler that connects programs written in C/C++ with other languages[10]. This gave me the idea of wrapping Python libraries into the C code instead, but the problem was a lot of the implementations involve writing scripts which can be used to either call the C or Python code. I chose

Cython over SWIG because I preferred to have a combination of both C and Python code within the same file to gain the maximum performance. I also considered the portability and ease of use for the end users, since SWIG only involves running a lot of different scripts and files that calls the different functions, I was worried that the end users would find it tedious to use.

Ctypes

This tool is very similar to SWIG, it provides foreign language (including C) libraries for Python. It provides an interface with external C code, which allows the user to load dynamic libraries and call the relevant C/C++ functions from pure Python code[11]. Ctypes constantly calls code, this means that it will be easy and efficient for simple tasks but as the project gets larger and larger, there would be more calls and callbacks to and from the Python code, making it a bottleneck, which could affect efficiency. RUS is quite large and will only get larger and larger when the “inverse” algorithm is implemented in the near future. Thus, Ctypes wouldn’t be the best option for this project.

Numba

Numba allows the use of high performance functions that are directly hard coded into Python. It uses annotations and array orientated Python code that is Just-In-Time (JIT) compiled to the underlying machine instructions with similar performance to that of C/C++, without having to switch languages[12].

It uses a JIT compiler which is a compiler that improves the performance of a program during compilation. This is done by compiling the byte-codes into its native machine code during run time. This allows the compiled code to be called directly instead of interpreting it at run time, and thus, increases the performance[13]. The JIT compiler can be automatically used for wrapping functions using

$$autojit$$

this allows for speed improvements because it is converted to a highly efficient compiled code in real-time[13].

Although these features of Numba can get close to C/C++ performance, it uses an automatic wrapping tool that is used to wrap the original Python code to increase its performance[14]. The main reason why I chose Cython over Numba is because for Cython, I can manually rewrite and/ or wrap the code myself, this allows for more flexibility. I can change or modify the code as much as I want to maximize the performance of each function.

Weave

Weave is part of the scipy package, it provides tools for including C/C++ code within pure Python code. When using weave, it has shown performance increases of up to 30 times faster than pure Python code. There are several ways of using weave, these include using:

```
weave.inline()
```

within the Python source code to allow the use of C/C++ code within Python.

```
weave.blitz()
```

which allows the translation of Python Numpy expressions into C expressions for faster execution, and

```
ext_tools
```

for building extension modules within Python [15].

I decided to use Cython over Weave because Cython allows the interchangeability of both C and Python code within the same file. Weave although does something very similar, it cannot mix between the two types of programming languages within each line. Also, Weave requires a C++ compiler and Numpy[15], which means that an additional compiler and package would need to be installed on the users machine, which is not what I am after. Weave is also not included in Python 3.x, which means that it wont be useful for a project like RUS because RUS is a ever-growing project with different features and implementations that are added to it each year.

SIP

SIP is a tool that is used to quickly write Python modules that interface and interact with C/C++ libraries. These are used as Python extension modules and are called 'bindings'. SIP uses a code generator and a Python module. This generator is used to process a set of specification files and generates the corresponding C/C++ code which is then compiled to create the bindings. The specification files are very similar to the C/C++ header files and contain descriptions of the interface of the C/C++ classes, functions and variables[16].

The main reason why I chose Cython over SIP was because SIP generates the bindings automatically, this means that I have no control over what is automatically generated. Sometimes these generated bindings hinders the performance rather than improving it, due to possible hidden background processes. There are also no interchangeability between C and Python code, which limits flexibility.

In summary, the different tools mentioned above have their own advantages and disadvantages compared to each other and to Cython. I chose Cython over the tools mentioned above because Cython allows the interchangeability between C and Python code within one file. The wrapping of the code is also done manually, which I find to be a huge advantage over the other tools because I have more control over each function and can change each function to make it as optimized as possible. Cython also generates an executable when compiled, this makes it very portable and can be used without having to install Cython on the local machine.

6.3 Use cases for Cython and how they apply to RUS

Developers over the years have successfully used Cython in many situations, from being used in high development programming firms, to creating small and personal projects. There are a variety of uses; the following shows some of the significant uses of Cython:

Data Transformation and Reduction

Cython can be used for either very small of very large amounts of data. For the use for small data sets that are repeatedly used numerous times (for example loops), the overhead of Python will be largely significant. Wrapping this function with C code or rewriting it in terms of Cython would remove this overhead and thus would make it more efficient. For the use with large amounts of data, Python has two problems:

- It requires large amounts of temporary data
- It constantly moves the results from the temporary data to the memory bus

These mean that there would be a huge bottleneck due to the large demand. For example, consider the following code

```
v = en.sqrt(x**2 + y**2 + z**2)
```

The 3 variables x, y and z will be squared and stored into the temporary buffers. There will be 3 temporary buffers because each squaring of the variables would need one each. Then another temporary buffer would be required when the adding process occurs. This process would constantly be repeated for all the different inputs that the program has to go through. Cython is very efficient with this because it doesn't use temporary buffers. Cython allows the possibility of wrapping C code within the function and allows the loop to run at their native speed.

This applies to RUS because the inverse code calls the forward algorithm multiple times. By wrapping the original C code within the Python implementation or by rewriting the functions in terms of Cython would allow the function to be running in the original native speed, which is the speed of C.

Optimization and Equation Solving

RUS deals with a lot of different formulas and equations; these equations are placed in a lot of different functions that are called several times. These functions rely on making new steps based on what was previously computed in the other functions. By declaring the types for each of the variables and functions, would decrease the call overhead that is made each time a function is called.

Arrays and Data Repacking

RUS requires generating a lot of different arrays that are used as 'matrices' when computing the elastic tensors. These can be seen in the functions that are involved with creating matrices for the isotropic, cubic, hexagonal, tetragonal and orthohombic shapes located in the rus_tools class. These again would require a lot of loops to store these generated data into an array. Cython is very good at dealing with large arrays and data repacking, using Cython would reduce the time it takes to loop through the arrays as described above[4].

6.4 Cython Example

The following shows and explains the basics of Cython and how it works. The following code segments are from the original C implementation, Python implementation and my implementation using Cython:

The original C code:

```
double  volintegral (double d1, double d2, double d3, int l,
                int m, int n, int shape)
{
  if ((l%2==1) || (m%2==1) || (n%2==1)) return 0.0;
  else
    switch (shape) {

      /* ell. cylinder shape */
    case 1: return 4.0*PI*pow(d1, l+1)*pow(d2, m+1)*
        pow(d3, n+1)/(double)(n+1)
              *doublefact(l-1)*doublefact(m-1)/doublefact
          (l+m+2);

    /* spheroid shape */
    case 2:  return 4.0*PI*pow(d1, l+1)*pow(d2, m+1)*
                pow(d3, n+1)
               *doublefact(l-1)*doublefact(m-1)*doublefact(n-1)/
               doublefact(l+m+n+3);

    /* rp shape */
    default: return 8.0/((l+1)*(m+1)*(n+1))*pow(d1, l+1)*
                pow(d2,m+1)*pow(d3, n+1);
    }
}
```

The Python Implementation:

```
def volintegral(dimensions,l,m,n,shape):
    global _memo_vol_max
    global _memo_volintegral

    hl = l//2
    hm = m//2
    hn = n//2
    small = hl < _memo_vol_max and hm <
    _memo_vol_max and hn < _memo_vol_max

    if small and _memo_volintegral[hl][hm][hn]:
        return _memo_volintegral[hl][hm][hn]

    # ell. cylinder shape
    if shape == 1:
        ds = dimensions[0]**(l+1) * dimensions[1]**(m+1)
        * dimensions[2]**(n+1)
        df_lm = doublefact(l-1) * doublefact(m-1)
        result = 4.0 * scipy.pi * ds / (n+1) * df_lm
        / doublefact(l+m+2)

    # spheroid shape
    elif shape == 2:
        ds = dimensions[0]**(l+1) * dimensions[1]**(m+1)
        * dimensions[2]**(n+1)
        df_lm = doublefact(l-1) * doublefact(m-1)
        df_all = doublefact(l+m+n+3)
        result = 4.0 * scipy.pi * ds * df_lm
        * doublefact(n-1) / df_all

    # rp shape
    else:
        result = 8.0 / ((l+1) * (m+1) * (n+1)) * ds

    if small:
        _memo_volintegral[hl][hm][hn] = result

    return result
```

Cython Implementation:

```
cpdef volintegral(dimensions, int l, int m,
        int n, int shape):

    global _memo_vol_max
    global _memo_volintegral
```

```
        hl = l//2
    hm = m//2
    hn = n//2
    small = hl < _memo_vol_max and hm < _memo_vol_max
    and hn < _memo_vol_max

    if small and _memo_volintegral[hl][hm][hn]:
        return _memo_volintegral[hl][hm][hn]

  /* ell. cylinder shape */
  if shape == 1:
  result = 4.0*scipy.pi*pow(dimensions[0], l+1)*
  pow(dimensions[1], m+1)*pow(dimensions[2],n+1)/
  <double>(n+1)*doublefact(l-1)*doublefact(m-1)/
  doublefact(l+m+2);

  /* spheroid shape */
  elif shape == 2:
          result = 4.0*scipy.pi*pow(dimensions[0], l+1)*
          pow(dimensions[1], m+1)*pow(dimensions[2], n+1)*
          doublefact(l-1)*doublefact(m-1)*doublefact(n-1)/
          doublefact(l+m+n+3);

    # rp shape
    else:
        result = 8.0/((l+1)*(m+1)*(n+1))*pow(dimensions[0],l+1)*
        pow(dimensions[1],m+1)*pow(dimensions[2], n+1);

    if small:
        _memo_volintegral[hl][hm][hn] = result

    return result
```

As you can see in the Cython example above, it is not purely either C or Python. It is a combination of both, with a mixture of Cython code that links the two programming languages together. It is also possible to wrap pure C code into the Python code by using the different Cython function declarations. These include:

def

def is the way functions are defined in pure Python. When a function is defined as def in Cython, the code that is in that specific function can only be written in pure Python code and return pure Python objects. The code will be treated as pure Python code only and will incur Python's overhead[18].

cdef

cdef is the way functions are defined for pure C code. All of the code inside this function must be written in pure C code and all variables must be statically declared[18].

cpdef

cpdef in Cython is used to tell the compiler that it is a combination of both C and Python code[18].

I decided to use cpdef for volintegral because this function is constantly being called. Since some of the functions in the rest of the Python source code may not need to be wrapped/ rewritten, declaring this function as cdef and wrapping pure C code would not be ideal because the implementation of the dimensions and how it is parsed into the function are different between C and Python.

7 Profiler

I used Profilers, most specifically, the built-in Python profilers to test the performance of the original Python implementation and my Cython Implementation. I used profilers rather than print statements because profilers list the performance of each functions, along with other details such as how many calls are made to that specific function. Profilers have helped me with my project because it allowed me to identify the most commonly called and used functions in the RUS code. Knowing the most commonly used functions allowed me to focus on specific parts of the code that, when wrapped, would provide the most increase in performance.

The output of the profilers consists of the following headings[17]:

- Number of calls ('ncalls')
  - This shows the number of calls made by each function

- Total time (tottime)
  - This is the total time a compiler spends within a specific function. This excludes the time from the function calling other functions
- Percentage of call (percall)
  - There are two type of percalls, the first one is
    * The result of dividing tottime with ncalls, and
    * The result of dividing cumtime with primitive calls
- Cumulative Time (cumtime)
  - This is very similar to tottime but it also includes the time that the function spends in the other functions that it calls
- Filename, line number and function name (filename:lineno(function))
  - This provides details on the filename the profiler is working on, line numbers of the functions examined and the name of the function.

Under each of the above headings are the results for each function. Here is an example of the output you get after profiling:

```
Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 1043250    8.571    0.000    8.805    0.000 rus_tools.py:241(gamma_h
       4    0.544    0.136    8.603    2.151 rus_tools.py:230(dgamma_...
  214350    0.263    0.000    0.263    0.000 rus_tools.py:807(volinte...
       2    0.248    0.124    0.277    0.138 rus_tools.py:866(e_fill)
       2    0.115    0.058    9.868    4.934 rus_tools.py:443(formod)
       3    0.108    0.036    0.261    0.087 __init__.py:1(<module>)
       2    0.039    0.019    0.792    0.396 rus_tools.py:944(gamma_f...
```

This is a small section of the output file. There are many more lines of output that show both the internal (anonymous) functions and the functions shown in the source code.

To use and run the built-in Python profiler, we can use a simple command in a terminal (for Unix based operating systems). This command is:

```
python -m cProfile rus.py inverse
```

Sorting the results of the result is also possible by adding

```
 -s tottime
```

into the command above. The final terminal command should look like this:

```
python -m cProfile -s tottime rus.py inverse
```

There are three types of Python Profilers; these are ‘profile’, ‘cProfile’ and ‘hotshot’. I chose to profile with cProfile for this project because cProfile produces less overhead and works with both Python and C implementations. Since it works for both pure Python and C codes, I was also able to use it for Cython. Also, since I am using the same profiler, it will help me keep my profiling results as consistent as possible.

After wrapping and rewriting the functions, Profilers are a good way to test each function for its performance. Profilers provide a lot of details related to each of the functions performance, but it is still required to run a wall clock timer on the entire source code. This is because there could be background processes related to Cython that we are not aware of, that could hinder that performance of the overall code if we are not careful.

8 Performance between C, Python and Cython

During week 8 of semester 1, I was tasked to write a little demo showing how Cython worked. I wrote a demo that wrapped one of the functions of the RUS code. I decided to wrap the function ‘volintegral’, this was shown earlier. I used profiling to test the performance of the code for the Python implementation of volintegral, it showed to be running at approximately 0.263 seconds:

```
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
214350   0.263   0.000     0.263    0.000  rus_tools.py:807(volintegral)
```

After wrapping the function, I ended up with the speed of 0.057 seconds:

```
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
214350   0.057   0.000     0.060    0.000  rus_tools.pyx:808(volintegral)
```

These times fluctuate and ranges from 76 to 80% faster than the pure Python Code.

After my demo was demonstrated to my supervisors, my supervisor tested the code's performance against the C and pure Python code. The results are as follows:

- Speed of C
  - 29.545 seconds
- Speed of the Cython Implementation after wrapping the functions volintegral and half of gamma_helper
  - 13 minutes and 55.518 seconds
- Speed of Python
  - 19 minutes and 16.073 seconds

These all ran at 100 iterations each, ran simultaneously and on the same machine.

As you can clearly see, the C implementation is much faster than the other implementations. By just wrapping one of the main functions that are constantly being called, it has shown a very large increase in performance compared to the pure Python code. The goal is to get as close as possible to the original C implementation's performance.

9 Why Python is slower than C

As already mentioned earlier, the main motivation for wrapping C code into and/or rewriting the original Python code was because Python is much slower than C. There are several reasons why Python is slower, these are outlined below:

9.1 Python Programming Language

9.1.1 Global Interpreter Lock (GIL)

A Global Interpreter Lock is a mechanism that is used by the Python interpreter. It limits multiprocessing by making sure that only one thread executes a Python bytecode at a time. This is used to ensure that the Python Object model is safe against concurrent access. This decreases Python's performance because the GIL makes system calls incur a large overhead, which is much more significant on a multi-core processor[19].

9.1.2 Interpreted at runtime

Python is interpreted at runtime rather than compiled. When compiled, a compiler can optimize the code for repeated or unused operations, which can increase the performance of the code[16].

9.1.3 Dynamically typed

Python is a dynamically typed language rather than statically typed. This means that during execution, the Python interpreter does not know the variable types of each of the defined variables. This makes programming more convenient for the programmer, but this hinders the performance.

For example, by looking at the following C code:

```
int a = 1;
int b = 2;
int c = a + b;
```

```
The sequence of events:
1. Assign <int> 1 to variable 'a'
2. Assign <int> 2 to variable 'b'
3. Call binary_add<int, int> (a, b)
4. Assign the results to <int> variable c
```

During compilation, the compiler would know that the variables a, b and c are integers. The compiler can then add the two integers and return another integer which is a simple value located in memory.

The Python version is as follows:

```
a = 1
b = 2
c = a + b
```

The sequence of events:

1. Assign 1 to variable 'a'
   (a) Set variable a's PyObject_HEAD's 'typecode' to an integer
   (b) Set variable a's 'val' to 1
2. Assign 2 to b

    (a) Set variable b's PyObject_HEAD's 'typecode' to an integer
    (b) Set variable b's 'val' to 2

3. Call binary_add(a, b)
    (a) Find the 'typecode' of a from a's PyObject_HEAD
    (b) When the variable type is found, retrieve a's 'val'
    (c) Find the 'typecode' of b from b's PyObject_HEAD
    (d) When the variable type if found, retrive b's 'val'
    (e) Call binary_add$< int, int > (a- > val, b- > val)$
    (f) After the addtion, store the results in 'result'

4. Create a Python Object 'c'
    (a) Set c's PyObject_HEAD's 'typecode' to an integer
    (b) Set c's 'val' to 'result'

The Python interpreter will not be aware that these are integers and will treat these variables as Objects. During each execution of the Python code, the Python interpreter has to inspect the 'PyObject_HEAD' to find the information on the type of the variable. When the type of the variable is found, the appropriate addition routine is called. After computing the result, a new Python Object must be created to store and hold the return value. By comparing the sequence of events that occur at the assembly level, Python's sequence of events involves much more steps compared to C. The more events that are running in the background, the longer the program takes to execute[20].

9.2 C Programming Language

There are two main reason why C is faster than Python, these include the fact that C is compiled rather than interpreted and C is statically typed. These two reasons are somewhat related because the compiler checks the variable types during the compiling stage. Compilers convert the source code into machine code, code that is compiled tend to have better performance compared to those that are interpreted because the overhead of the translation process is much higher for interpreters[21].

10 What I have done so far

I have been having weekly meetings with both of my supervisors, each week I have been setting weekly goals that are to be accomplished by a date that was mutually agreed on with my supervisors. Here is a brief outline of the tasks that I have accomplished:

- Read the sources suggested by my supervisors for a better understanding of Resonant Ultrasound Spectroscopy
- I did additional research to have a deeper understanding of the project
- I went to a live demo to understand the basics of what the experimental setup looked like and how it works
- Decided and negotiated with my supervisors and the other Btech student, Elvis Chuah, on what we wanted to contribute to the project
- Looked at and tried to understand the Python Implementation. I also installed Python and Anaconda on my system to test and run the code
- Looked at and tried to understand the C code. As mentioned earlier in this report, the C implementation is very difficult to install. This means that I have not installed and tried the C implementation yet.
- Learned how to use GitHub
- Researched and looked at ways to accomplish the tasks
- Researched and learned Cython
- Research and used Python Profilers to get a better idea of which functions in RUS is hindering its performance
- Wrapped a function of RUS as a demo for my supervisors to demonstrate how Cython worked
- Used Python Profilers to test the performance of the Cython wrapped function
- Uploaded my demo onto GitHub
- After demonstrating the demo, I have wrapped 5 other functions but 3 of which did not show a huge improvement in performance. I will have to do additional research and find an alternative approach in wrapping those functions

11 Plans for Semester 2

I have done the research and I am aware of what I have to do for this project. For semester 2, I want to completely finish the wrapping/ rewriting of the Python implementation and move onto the next phase of the project. When the final wrapping/ rewriting is complete, I am planning on making the executable for Cython to work on both a Unix based operating system and a Windows based operating system. My supervisor, Paul Freeman has tested the executable that I generated on a Mac on a Linux operating system, but it unfortunately did not work. This is something that I want to develop further and make sure that the executable file generated by Cython would work on Mac, Linux and Windows. Cython generates an executable with extension .so and .pyd. The .so file for for Unix and .pyd is for windows. If I am successful in creating the executable, it would make the user experience much more convenient, the users won't have to install Cython and can run the executable directly from the terminal on a Unix machine or by command prompt and/or double clicking on a Windows machine. If my conversion into an executable is unsuccessful, then I will have to create a MakeFile to allow the users to compile the program.

Currently, the version that I am currently working on and the version that I uploaded on GitHub requires the user to compile pyrex (.pyx) files with a Setup.py file that I wrote. Compiling the source file (pyrex file) requires the user to install Cython and would take quite a bit of time when the RUS project is further developed in the future.

When the wrapping and rewriting of the code is complete, I am required to move onto the next phase of the project. This would involve more research and looking at further developments of the RUS code. One of the tasks that my supervisor Kasper has suggested is to look at mapping out the misfit functions for the elastic case of the modes of an isotropic sphere.

I am also wanting to install and run the C implementation. I think that this will be very beneficial for me as I will get a better understanding of the C code and I will be able to profile it. Profiling the C implementation will be very beneficial because it will give me a goal to reach in terms of pushing my wrapped/ rewritten code to the limits. My wrapped/ rewritten code should be as close as possible to the C implementation's wall clock speed.

12 Conclusion

In conclusion, the RUS project deals with many different formulas and calculations that involve a lot of repeating and calling of different and specific functions within RUS. I've looked at several different implementations for wrapping C code within Python and have decided to choose Cython to increase the performance of the Python implementation because it allows the interchangeability between C and Python within one file called a Pyrex file. There is still a lot of work that needs to be done for this project, such as completing the warpping of the Python code, MakeFiles for GitHub and further research and development of the RUS code.

13 Bibliography

1 Li, G., & Gladden, J. R. (2011). High temperature resonant ultrasound spectroscopy: a review. International Journal of Spectroscopy, 2010.

2 Watson, L., & van Wijk, K. (2014, June). Resonant Ultrasound Spectroscopy of Anisotropic Shale Samples. In AGU Fall Meeting Abstracts (Vol. 1, p. 4289).

3 Schwarz, R. B., & Vuorinen, J. F. (2000). Resonant ultrasound spectroscopy: applications, current status and limitations. Journal of Alloys and Compounds, 310(1), 243-250.

4 Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2011). Cython: The best of both worlds. Computing in Science & Engineering, 13(2), 31-39.

5 Cython: C-Extensions for Python http://cython.org/

6 Freeman, P. (2015). A Python Implementation of the Forward RUS Eigenvalue Calculation Algorithm.

7 Zadler B. & Le Rousseau J H.L Installation and Help Guide to Fitspectra and RUS-inverse. Resonance peak fitting, forward modeling and inversion

8 The Performance of Python, Cython and C on a Vector — Cython def, cdef and cpdef functions 0.1.0 documentation. (n.d.). Retrieved from http://notes-on-cython.readthedocs.io/en/latest/std_dev.html

9 A Speed Comparison Of C, Julia, Python, Numba, and Cython on LU Factorization (IT Best Kept Secret Is Optimization). (n.d.). Retrieved from https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization?lang=en

10 Simplified Wrapper and Interface Generator. (n.d.). Retrieved from http://www.swig.org/

11 15.17. ctypes — A foreign function library for Python — Python 2.7.11 documentation. (n.d.). Retrieved from https://docs.python.org/2/library/ctypes.html

12 Numba — Numba. (n.d.). Retrieved from http://numba.pydata.org/

13 IBM Knowledge Center. (n.d.). Retrieved from
https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/
com.ibm.java.zos.70.doc/diag/understanding/jit_overview.html

14 Numba vs. Cython: Take 2. (n.d.). Retrieved from https://jakevdp.github.io/
blog/2013/06/15/numba-vs-cython-take-2/

15 Weave (scipy.weave) — SciPy v0.17.1 Reference Guide. (n.d.). Retrieved
from http://docs.scipy.org/doc/scipy/reference/tutorial/weave.html

16 SIP - Python Wiki. (n.d.). Retrieved from https://wiki.python.org/moin/SIP

17 26.4. The Python Profilers — Python 2.7.11 documentation. (n.d.).
Retrieved from https://docs.python.org/2/library/profile.html

18 Cython Function Declarations — Cython def, cdef and cpdef functions
0.1.0 documentation. (n.d.). Retrieved from
http://notes-on-cython.readthedocs.io/en/latest/function_declarations.html

19 GlobalInterpreterLock - Python Wiki. (n.d.).
Retrieved from https://wiki.python.org/moin/GlobalInterpreterLock

20 Why Python is Slow: Looking Under the Hood. (n.d.). Retrieved from
https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

21 — National Public Library - eBooks — Read eBooks online. (n.d.). Re-
trieved from http://nationalpubliclibrary.info/articles/Compiled_language