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

October 2016

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

Abstract

This report contains and describes everything that I've done this year for my Bachelor of Technology (BTech) project. 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 equations used in RUS were originally written in the form of computer algorithms by Jerome H.L. Le Rousseau. His code was originally written in C programming language but was later translated into Python by Paul Freeman for ease of use, but this hindered the performance. I was tasked to improve the performance of the Python implementation using the tool Cython. My main focus was on the forward code as the inverse code calls the forward code multiple times to perform it's calculations.

Acknowledgments

I would like to thank my academic supervisors and course coordinator for their 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

1 Introduction

Resonant Ultrasound Spectroscopy (RUS) consists of two algorithms; these are the forward and inverse algorithms. The forward algorithm 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. The inverse algorithm is the reverse of the forward algorithm, it takes the resonant modes from the solid objects and finds the corresponding elastic properties.

The main focus for my project is on the forward algorithm because the main calculations/tasks are performed by the forward algorithm. The inverse algorithm makes multiple calls to the forward algorithm and thus improving the speed of the forward algorithm would result in an increase in the overall performance of both the forward and inverse algorithms. Small increases in the performance of the forward algorithm would result in a huge increase in the performance of the inverse. Both the forward and inverse algorithms were originally written in the programming language, C, but was later translated into the programming language, Python, due to the original code relying heavily on outdated versions of software libraries or libraries that were difficult to obtain and to install. Currently, the only 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. With the use of Python, it has made the installation process much simpler and easier to use because Python comes with a large collection of libraries which can be easily installed locally. The only disadvantage with using Python is its performance, it's significantly slower than the C implementation.

Cython is the chosen tool for this project because it allows for the interchangeability of both C and Python code within one file. This provides more flexibility in terms of writing the most efficient code. It also generates an executable, which can be very useful when portability is desired.

This report will document the improvements in performance 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 some sections of the code with a combination of Python, C and Cython specific code.

2 Resonant Ultrasound Spectroscopy (RUS)

2.1 What is Resonant Ultrasound Spectroscopy?

Going into the second week of this project, I was lucky enough to see first-hand how this experimental procedure worked. Seeing it in person has given me a better understanding of how RUS worked and how the resonance peaks are related to the code. It is a procedure used to determine the material properties of solid objects, this is done by studying the mechanical resonances of the solid objects. Something that I found really interesting with RUS is that it can be used on objects such as fruit. It can be used to determine the material properties of fruit to determine its ripeness level, knowing this makes it possible to determine when it is going to rot and when the most optimal time to consume is. The output consists of a wide range of varied resonant modes, which are determined by the solid objects “shape, elastic constants, crystallographic orientation, density, and dissipation” [1].

The general setup of this experiment is shown in Figure 1.

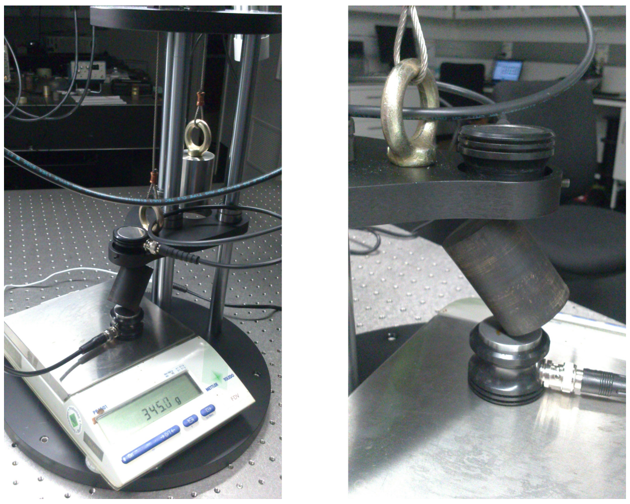


Figure 1: Experimental setup (Watson, L., & van Wijk, K. 2014, June)

Figure 1 is the closest photo that I could find that resembles what I saw. It is from a photo used by an ex-student here at the University of Auckland that actually worked on this project a couple of years ago. A diagram view of what happens during this experiment and the equipment used are represented in Figure 2.

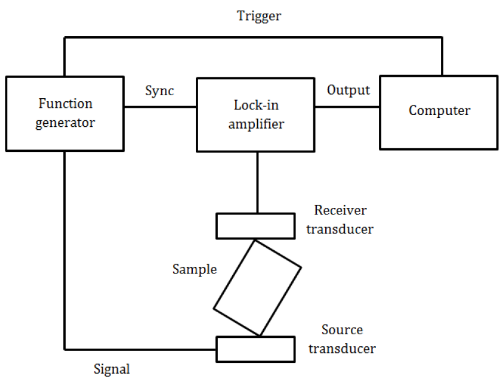


Figure 2: Diagram view of experimental setup (Watson, L., & van Wijk, K. 2014, June)

There were quite a few different equipment that are required for this setup, these include:

- Transducers - Two transducers are required, one for the source of varying frequencies and the other for measuring the response from the solid object^2. This can be clearly seen in Figure 1
- Function Generator - This is used to generate the varying frequencies that are sent through the source transducer^2
- Amplifier - this is used for amplifying and filtering the response from the solid object^2

Something else that cannot be seen in the photo is the gel or substance used to hold in place the solid object in between transducers. Kasper used honey during the demonstration but any sticky substance can be used.

The solid object, which is usually a parallelepiped or a spherical shape, is held tightly between two piezoelectric transducers. The source transducer then stimulates the objects by producing and sending a wide range of elastic waves of constant amplitude and varying frequencies through the solid object. The receiving transducer then detects the wide range of resulting frequencies, but what we are interested in are the resonant peaks. The resonant peaks occur due to the fact that the elastic waves that are sent into the solid objects correspond to the natural frequencies of the solid object itself. 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[3]. These natural frequencies are dependent on the shape, size and elastic parameters of the solid object. A sample of a typical output showing the resulting frequencies are shown in Figure 3. The resonant peaks are shown by the 3 vertical spikes in the graph.

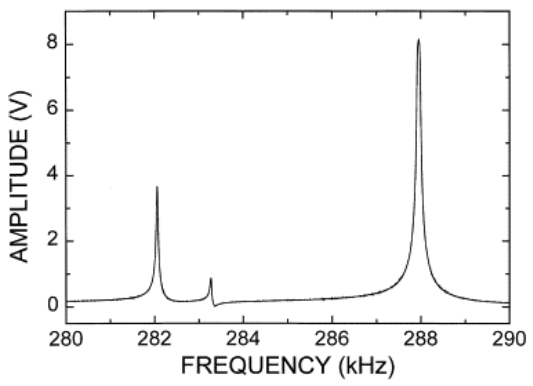


Figure 3: Example of experiment output (Schwarz, R. B., & Vuorinen, J. F. 2000)

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[4]. 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.

3 Installation of the C Implementation

3.1 Older 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 C implementation of the RUS code can be installed and executed[4].

The following is from the PDF file, "Zadler B. & Le Rousseau J H.L Installation and Help Guide to Fitspectra and RUS-inverse. Resonance peak fitting, forward modeling and inversion", it is an official guide on how to install the C implementation and is located on the RUS GitHub page. It is a brief interpretation of what is required to get the C implementation installed. There are a lot of necessary files that need to be installed and steps to take for the C implementation. The main files that needs 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.

3.2 Newer C Implementation

The previous subsection was written for my mid-year report before I had any hands-on experience with installing and actually running the C implementation on my local machine. Since then my supervisor, Paul Freeman, has created a newer version of the C implementation which was easier to install and run. It requires less of the libraries mentioned above to be installed by the user. Something that I want to put emphasis on is that the following is not an official guide on how to install the newer C implementation, it is just the steps that I personally went through to get it working.

Firstly, I started off by opening a terminal on a Linux based operating system and then checked whether the preliminary C/C++ tools were installed or not. I ran the following command to check this:

```
sudo apt-get install build-essential
```

This command checks whether it is installed or not and if it's not, it will install it onto my local machine. I found that this essential package is already pre-installed on Linux but it's good to make sure. When that was done, I installed lapack, this is essential for computing the calculations used in the RUS code,

```
sudo apt-get install liblapack-dev
```

The next step is to get the RUS C code from GitHub. I used the following command to install all the tools necessary to checkout the code from GitHub:

```
sudo apt-get install git
```

After I installed git, I checked my directory and made sure that I was working in the correct directory. I then cloned the RUS code from GitHub using the following command:

```
git clone https://github.com/PALab/RUS.git
```

From here, I changed my current working directory to the "RUS" folder. I used the command "cd RUS" to do this. I then checked out the "dependency_cleanup" branch, I did this by using the following command:

```
git checkout dependency_cleanup
```

I was then able to run "make". After running "make", I was able to see two executable files, these are "rus_forward" and "rus_inverse". I then ran:

```
sudo make install
```

After the above command, I was then able to run either the forward or inverse code using the following commands:

```
sudo make forward_example
sudo make inverse_example
```

The steps I took above are the full set of instructions that I took to get the newer C implementation working on my machine. It looks longer than the old implementation because what I have written for the old implementation was only a brief outline of what the required packages were during the installation process. The actual documentation for the old C implementation is a 32 paged PDF document. The newer C implementation is much easier to install and only requires 2 readily available packages, “build-essential” and “liblapack-dev”.

4 Cython

4.1 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[6].

4.2 The limitations of Cython

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.[5]

4.3 Alternatives to Cython

I looked at several tools that would help achieve what I needed for this project. 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[7]. 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[8]. 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[9].

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[10]. 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[10].

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[11]. 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 [12].

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[12], 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[13].

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 found 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.

The above are some of the alternatives to Cython that I looked into before I started the coding aspects of project. Now that I'm nearing the completion of the project, I decided to look at some more alternatives to Cython for future work or extension purposes. I looked into a few of the other tools available online and What I looked for was a tool with good online documentations and forum help. As I was looking, I found the tool "Boost.Python", it is quite similar to Cython but is much more flexible, easier to use and to implement. The more I looked into Boost.Python, the more that I found that it's simpler to use compared to Cython. It is very similar to Cython except it can implement C or even C++ functions into Python without any special tools that needs to be installed. You can just directly copy and paste the C code within the Python code without any modifications and from what I've read, it will work well with the different Python and C objects. There are definitely some extra code that are required such as:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(hello_ext)
{
    using namespace boost::python;
        [FUNCTION STARTS HERE]
}
```

(Boost.Python - 1.62.0. (n.d.))

I haven't personally tried it but from what I have been seeing and reading has suggested it could be a good tool to use. There is also a Boost.Python generator called Py++ that can be used to help generate these functions. This extra tool helps, checks and makes sure that the generated Boost.Python functions run precisely and accurately during run-time.

Boost.Python can also be used to incorporate and use Python libraries within C[13]. Since the main idea of the Python version is to reduce the dependencies on hard to obtain and to install C libraries, implementing Python libraries into C would help solve this problem and still run at C's performance.

I cannot guarantee that this tools would work as I haven't implemented and experimented with it myself but from when what I have read, it looks like quite a good tool to use. If I had another year to work on this project, I would look into Boost.Python as it seems like there is more potential compared to the others mentioned above. Instead of wrapping C functions within Python, I would try working with the C code and try to implement the Python libraries within the C code instead. I initially thought Cython was a really good tool to use but later found out that it was much more difficult than expected, so from just me reading on Boost.Python, I cannot guarantee that it will definitely work.

4.4 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 can only be installed onto specific operating systems, mainly Linux in this case[6]. Also, Linux is the only operating system that has the necessary libraries built into it's kernel. 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 easier. 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 the Python code directly to C, which is then compiled and linked to Python[5]. 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[6].

The wrapping process of Cython is all done manually via the Pyrex file. This allows for many different ways of implementing the source code. 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.

4.5 Cython Example

Here is a simple example of a Cython function on calculating prime numbers. It takes the max number as a parameter and then returns the prime numbers as a Python list.

```
1  cpdef primes(int kmax):
2      cdef int n, k, i
3      cdef int p[1000]
4      result = []
5      if kmax > 1000:
6          kmax = 1000
7      k = 0
8      n = 2
9      while k < kmax:
10         i = 0
11         while i < k and n % p[i] != 0:
12             i = i + 1
13         if i == k:
14             p[k] = n
15             k = k + 1
16             result.append(n)
17         n = n + 1
18     return result
```

(Basic Tutorial. Cython 0.25b0 documentation. (n.d.))

The main changes to the code above are from lines 1 to 3. On line 1, the function is defined as cpdef so that functions that are defined as def or cdef can call this function. "kmax" is defined as int, this is one of the reasons why Cython is much faster than Python, any object that is now passed into this function will be converted into a C integer. Lines 2 and 3 are actually added in and are declaring the variables as C variables with the variable type "int". The while loop cannot be changed much as it is quite similar to a C while loop but it uses variables n, k and i which are now actually C variables, so it actually increases the performance of the while loop. The code on line 16 is actually a Python statement but since variable "n" is a C variable, the performance of this Python statement is actually drastically improved on[15].

5 Cython version of RUS

5.1 Cython RUS Example

Please refer to the code segment in Appendix A. The code shows and explains the basics of Cython and how it works with the RUS code. The code shows that each section of code does not necessarily have to be either C or Python code, but rather a combination of the two with a mixture of Cython code that links the two programming languages together. The programming language of the code used within each function is dependent on how the functions are declared. There are many 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[16].

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[16].

cpdef

cpdef in Cython is used to tell the compiler that it is a combination of both C and Python code[16].

Defining each function correctly is important. Something that I've noticed when I was wrapping code is that if I don't have to define a function as cpdef, it's best that to just leave it as def. Sometimes defining it as cpdef without the need for it can actually hinder the performance slightly. It does take less time to build during the build phase of Cython but the run-time could be slower. This can especially be seen when you move from one machine to another without rebuilding the files. When I was testing the compatibility between machines without rebuilding the pyrex files first, it was actually slightly slower than the original Python implementation.

Sometimes defining functions as cpdef is the only option when the function is constantly being called by either def or cdef functions. Functions that are not frequently called by other functions should be kept as either def or cdef to help keep the code run efficiently. Examples of this could be the functions that I didn't wrap, mainly those that had the run-time of 0.00 seconds in the Python implementation. Keeping these functions as def instead of cpdef would actually keep the code efficient. Functions defined as def or cdef cannot call each other so sometimes the only option is to use cpdef.

I decided to use cpdef for the function (volintegral) that I wrapped for my demo because of the reasons mentioned above, this function is constantly being called by several other functions. Since some of the functions in the rest of the Python source code may not need to be wrapped/ rewritten, declaring volintegral as cdef and then wrapping pure C code would not be ideal because the other functions wont be able to call it.

Looking at the code under "Cython Implementation" in Appendix A, line 1 shows that I changed the originally defined "def" function to a "cpdef" function. As mentioned earlier, I did this because this function is constantly being called by other functions. These calling functions may not necessarily be called by a cpdef defined functions, they could be def or cdef so the only choice would be to define it as cpdef. Also on line 1, I defined the types for the variables l, m, n, and shape. By defining the types, Cython does not have to perform any extra background tasks to work out the types of each of the functions. In this case, the function will be compiled during the build phase rather than interpreted during run-time like in Python. Since it wont be interpreting the code at run-time like in Python, it increases the performance of the function. Also, the objects are also now passed in as C objects rather than Python objects, so it gains the speed of C. From lines 2 to 13, the code was unchanged. From lines 14 to 24, the code was directly copied and pasted from the C implementation, but Cython couldn't use the constant "PI" so I had to change it to what was used in the Python implementation. I chose to use "scipy.pi" and at the same time was a little bit worried about the performance from using scipy, but I checked the performance of this by comparing scipy.pi with the hard coded values of PI, correct to 6 decimal places. They both showed quite similar results so I just used scipy.pi. Something to note is that I didn't define the type for "dimensions" because it was something difficult to implement in Cython. This is outlined in more detail in the next sections.

5.2 My thoughts and experiences with Cython

One of the main reasons why I chose Cython was because it offered the possibility of having both C and Python code within one file. My initial thoughts were that I can limit the number of files that I had to distribute when the final files were uploaded onto GitHub. Having both languages within one file also allowed a lot of flexibility because I didn't have to worry about separating the code and then making sure that each of the external functions were called properly. Cython also generates an executable when it is built, although this feature is also available in the other Cython alternatives, mentioned earlier, an executable combined with the flexibility of having both languages within one file made Cython my number one choice.

Combining the reasons mentioned above and after reading some of the examples of Cython online, it initially gave me the impression that it would be an easy tool to implement into this project. I initially thought I would be able to complete it within a semester and a half but it took longer than expected as implementing this tool was not as easy as it suggested online. I ran into quite a few problems, which are outlined here:

for loops

I had some trouble directly copying and pasting a simple for loop from C into Cython's pyrex file. Using C's for loop was not possible, which forced me to use either Python's or Cython's implementation of a for loop. Although using a Python for loop was possible, the performance could be improved upon with the use of Cython's own implementation of a for loop. For example, copying and pasting the following code from the C implementation into the pyrex file:

```
for (ik=0; ik<k; ++ik)
```

would give the error "Expected ')', found '='". I couldn't find an explanation online, but I think the reason for this is because Cython needs to convert the for loop code into C's version of the for loop, which is then used by the executable or pyrex files. Cython throws an error with C's for loop code because Cython does not know how to ignore C's code for a for loop and throws an error instead.

Dealing with objects

Dealing with the difference between C and Python objects, variables and functions were extremely difficult. Despite having the capability of having both C and Python within one file, calling or using an object that belongs to the other language was a challenge. For example, when I try to store the C object, int *, into a variable that is defined in Python, I will get the error, "Cannot convert 'int *' to a Python Object". Initially, I thought Cython was as very powerful tool that would deal with the different object types automatically in the background, but this ended up not being the case. I was able to fix this by making the variables global, for example:

```
cdef int *itab;
global itab;
itab = alloc1int(r);
```

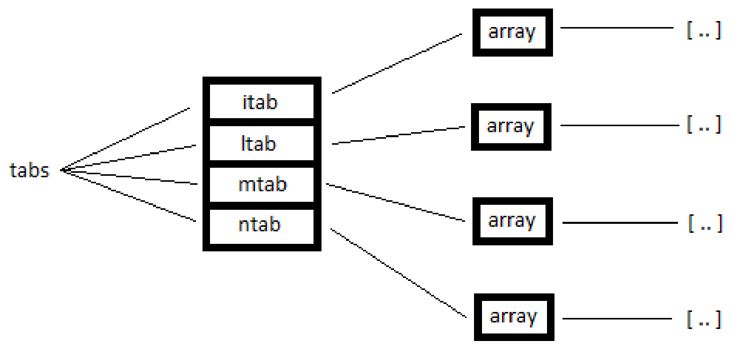
but this only works within that specific pyrex file. Since there are 4 other pyrex files that call each other, it would still throw the error mentioned above. Rewriting all these objects in terms of C objects only, would result in a repeat of the original C code. Also, as mentioned above, I found that some of the C code cannot easily be used in the pyrex files because Cython performs at its best when it's converting from Python or Cython code into an optimized C code. I've also found that functions that are declared as def are not exposed to functions that are declared as cdef, making function calls difficult. This can be fixed by declaring the functions as cpdef, but this would be at a risk of actually making the function slower than the Python implementation. This is because cpdef uses dynamic binding when an object is passed into the function. This performs extra tasks at run-time such as looking up the different arguments, thus this could potentially hinder the performance rather than improve it. This will depend on the object that it passes in and the code within that particular function[22].

Requires rebuilding

When I was testing the compatibility of the pyrex and executable files on multiple operating systems (Windows, Linux, OS X), I found that running the pyrex files alone without any rebuilding on that specific machine would result in speed performances that are actually slower than the original Python implementation. I think the cause of this could be because of some of the functions being declared as cpdef. Rebuilding the files would increase the performance to speeds that I would expect after wrapping. This does not apply to the executable files but I have sometimes found that its performance is slightly slower than the already rebuilt pyrex files by a few seconds.

Arrays in Cython

Creating a simple 1-dimensional or multidimensional array was the biggest challenge for me in this project. There were quite a few problems that I encountered, please have a look at Appendix B for all the possible combinations that I have tried. The example of the code in Appendix B is of the function index_relationship, I tried creating the arrays in this function because this function is in charge of creating "tabs", which I needed to split into itab, ltab, mtab and ntab. I attempted to split tabs to try and improve the performance further by statically typing these arrays individually. My thoughts were that tabs could be hindering the performance of the function and essentially the whole code. Splitting tabs up and defining the types for each individual tab would reduce and type determination process that Python (the Python interpretation step) does during run-time and thus increases performance. Trying to solve this issue was extremely difficult as creating a dimensional arrays in Cython was not easy. The main error that I got was the objects error described earlier ("Cannot convert 'int *' to a Python Object"), the other frequent error was when I was trying to create a 2 dimensional array, it kept giving me type errors. Here is a diagram which explains my understanding of the problem:



From what I understand, since the box with the text "array" in the middle is of type array and it's trying to store an array with an "int" ("[...]" is an array of ints), it throws a typing error because they do not match. If it was like this, where each "..." represents an "int", it would work perfectly fine because the types are all consistent:

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...

This shows that for our case, any implementation would not work because of how Cython deals with 2 dimensional arrays.

I have had a look online and there are ways to work around this problem but it would require an extensive understanding of how the underlying mechanisms of Cython works. I don't think I would have enough time to be able to do this within 8 months, especially when I found out about this problem during the second half of the project. It involves creating separate classes that deals with these types of functions and then incorporating it into our code.

As you have seen in Appendix B, I tried several different ways to implement an array in Cython. An explanation of what I tried are as follows:

Line 18 shows my first attempt at creating an array in Cython. I attempted to create an array that represented the Python and C implementations as closely as possible. They both created an array based on the variable "problem_size" for Python and "r" for C. Trying to create an array with these variables were not possible and resulted in the error: "Not allowed in a constant expression". Creating arrays in this way could only be possible if I had entered an integer into those brackets. Since there was no way I could work around this problem, I looked into creating the arrays with memory views. I decided to go with this approach after reading some of the recommendations from forums and on the Cython website. My implementation can be seen on line 20, but after working with this for a while, I was only able to create a 1 dimensional array that somewhat worked. I could build the files but I quickly saw some typing issues between Python and C during run-time, so I had to try something else. Lines 30 to 39 shows an example of a working 2 dimensional array that I found in an online forum, all my attempts after this is based on this example. I also attempted to use exactly what is shown here, but it resulted in typing errors again. Lines 43 to 63 shows more attempts that I've made after looking at forums, but after trying those I always end up with the same error, this error is "Cannot convert 'int *' to a Python Object". Lines 68 to 86 show the closest representation of an array to the C implementation. This is also the closest attempt that I got to creating the same arrays shown in the C implementation. The files are all able to build but as soon try to run it, it runs for 2 to 3 seconds before it crashes and shows a memory error. Since the code is used to create itab, ltab, mtab, and ntab, I thought it would be a good idea to implement it in the inverse and forward pyrex files instead and then from there, it can be used globally. Trying to implement it there resulted in the same error that I've been getting ("Cannot convert 'int *' to a Python Object"). The main reason why I have "global *" in front of the tab declarations is because it is supposed to fix the error "Cannot convert 'int *' to a Python Object". But for some reason would only work in the rus_tools pyrex file and not in the inverse and forward files.

The executable file

When I was choosing the tool to implement wrapper functions, one of the things that stood out when I was researching online was that fact that Cython generated "executable" files. This really intrigued me and I thought it would be very useful, especially for portability reasons but I found that it was quite deceiving. It is called an executable but it doesn't act like a normal executable that we use everyday, it still requires tying commands into a terminal for Linux and Mac or a command prompt for Windows.

I've been talking about the negative sides of Cython but there are also positives. A clear positive is its portability. It generates a .so file for Linux and Mac, and a .pyd for Windows, this generated file can run without Cython and only requires Python with a package like Anaconda, which the end users should already have installed if they are dealing with the RUS code. This solve one of the main motivations for the translation to the Python implementation because with the use of Cython's executable files, it has dramatically improved the ease of use and the installation process.

There was also a lot of information online on Cython, especially on their official website where there are a lot of information on the basics of Cython. I found almost everything that I needed to know from just their homepage alone. There were also some problems that I encountered but there is a large community of Cython users that have also experienced similar problems that really helped me solve some of the problems that I had.

Overall, Cython was a helpful tool that I got to experience this year for this project. It got me some speed improvements, but they weren't the speeds that I was expecting or have hoped for. Cython was a much more difficult tool to use than originally anticipated and if I knew of its difficulty level, I would have originally chosen another tool to for this project. Now that I'm nearing the end of the project, I found that there were much more disadvantages than there were advantages with Cython. A tool that I would have originally chosen over Cython if I knew of its difficulty level is Numba. During my initial research of different tools that would improve the performance of Python, Numba was a choice that I was considering. The only reason why I neglected using this tool was because most of it was done automatically. Since a lot of it was done automatically, I felt like it was not going to be challenging enough for a year long project. Now that I have experienced

Cython, I would have rather spent 3 to 4 weeks to implement Numba than to spend several hours on something that was near impossible to do within this time-frame. I spent most of semester 2 trying to find new ways to improve the already wrapped code further and actually got nowhere. I could have spent this time on further developments of the RUS project if I had chosen Numba over Cython.

Numba would have been my alternative choice if I got another chance to redo this project, but after nearing the completion of this project, I found that Boost.Python could be a better choice. Something that I would do differently with Boost.Python would be to wrap Python libraries into the C implementation rather than wrapping C functions into Python. Since the performance of C is what we desire, it would be best to stick with C and work from there. I would be very interested in seeing the performance if we wrap Python libraries within the C implementation.

5.3 Installation Process

The following installation processes are what I personally went through, they are not an official guide on the actual installation process. Your experiences may vary from what I've experienced and mentioned below.

5.3.1 Mac and Linux

Firstly, I checked whether the executable works on my machine. I checked this by using the "cd" command to change my directory to the directory that contains all the necessary Cython files that I had cloned earlier from the RUS GitHub page. I used the "ls" command to confirm that I had all the files. I then made sure that I had the correct executable for the python version that I have installed on my machine. I typed the command "python" into a terminal to check my python version. I then ran the following commands to check whether the executable worked:

```
python -c "import rus" forward
python -c "import rus" inverse
```

Both commands worked for me so I didn't have to install Cython. For documentation purposes, I assumed that it didn't work for me, so I went through the following:

I opened a terminal and made sure that I had Python and Anaconda (or something equivalent) installed. I did this by typing "python" and "conda list" into the terminal, if something appears after typing those commands, then I would know that I have them installed. I found that a Mac or Linux machine would have Python pre-installed. Anaconda wouldn't be pre-installed so I had to install Anaconda:

Mac

On a mac, I went to https://www.continuum.io/downloads and downloaded the Anaconda .pkg file. I then double clicked on the file and followed the instructions on-screen.

Linux

For Linux, it was all command based. I opened a terminal and typed:

```
bash ~/Downloads/Anaconda3-4.0.0-Linux-x86_64.sh
```

I then followed the on-screen instructions. Once that had completed, it said "Installation finished." and "Thank you for installing Anaconda!".

This next step is the same for both Mac and Linux:

Once Anaconda was installed, I installed Cython. I went to http://cython.org and downloaded the latest release of Cython. I then unpacked this downloaded zip file, opened a terminal and changed my current working directory to the unpacked Cython files. I then typed the following to install:

```
python setup.py install
```

I followed the instructions and when it completed, I checked whether Cython was installed or not by typing the following into the terminal:

```
cython
```

Once this completed, I changed my directory ("cd") to the Cython files that I cloned from the RUS GitHub repository. I then built the Cython files using:

```
python setup.py build_ext --inplace
```

Once it was done building the files, I was able to see .so files. Since these files were built on my system, I can guarantee that it work. I tested this by executing the executable, I did this by typing the following:

```
python -c "import rus" forward
python -c "import rus" inverse
```

Since Cython is now installed, I could also run the pyrex files, I can do this by typing the following commands:

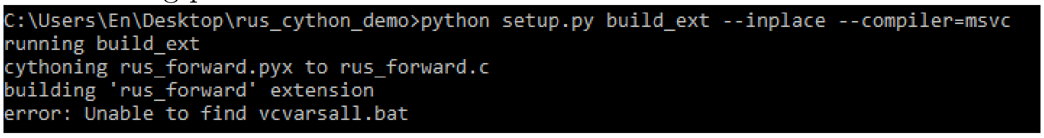
```
python rus.pyx forward
python rus.pyx inverse
```

I personally used python 3 but “Python” can be replaced with either “python2” or “python3”. This is important because of compatibility reasons seen in the “Program Compatibility” section of this report.

5.3.2 Windows

I found that Windows is the most difficult operating system to install Cython on. Before I started installing, I made sure that I had either Python 2 or 3 installed with Anaconda. I also checked whether the .pyd file works or not in command prompt or a developer prompt. Like for Mac and Linux, the executable works for me so I’ll just assume that it didn’t for documentation purposes. I also assumed that I didn’t have anything pre-installed. My personal installation process is as follows:

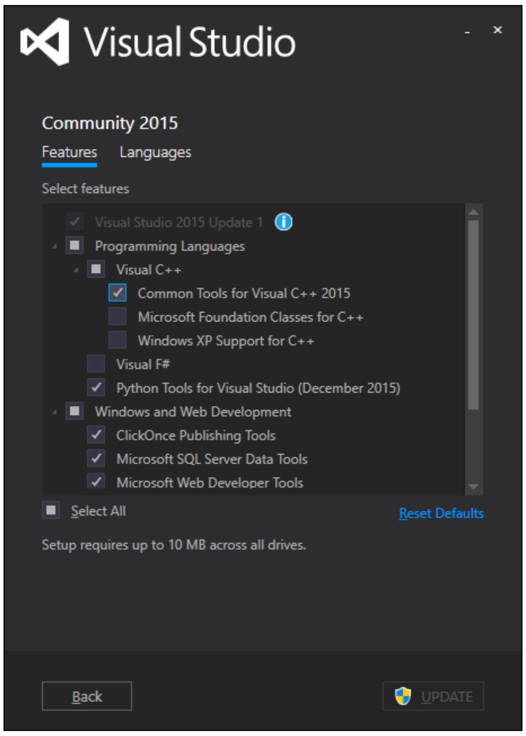
Firstly, I checked whether I had Python 2 or Python 3 installed for Windows. I found that either versions are fine but the 32-bit version would be the easiest and less prone to error option. Once Python is installed, Anaconda is required, again, the 32-bit version is the best option. After installing Python and Anaconda, I attempted to build the Cython files and ran into the following problem:



```
C:\Users\En\Desktop\rus_cython_demo>python setup.py build_ext --inplace --compiler=msvc
running build_ext
cythoning rus_forward.pyx to rus_forward.c
building 'rus_forward' extension
error: Unable to find vcvarsall.bat
```

This is actually quite a common problem. After spending some time searching for a solution online, I found that it is due to missing a C compiler that is required to compile the Cython generated C files. There are 2 solutions to this and these are specific to either Python 2 or Python 3. Both Python 2 and 3 require you to install “Microsoft Visual C++ Compiler for Python”, this is essential as it contains the missing C compiler that threw the error shown above. I found that if you have Visual Studio installed, it could be pre-installed or can be easily installed by checking a box:
I accessed these options by going into the Control Panel, then clicked on “Microsoft Visual Studio Community 2015”, then “Change”. A Visual Studio window then pops up and I clicked on “Modify”.

I then modified the setup.py file. This involves replacing the following from the Mac and Linux version:



- ✓ Visual Studio 2015 Update 1
- ■ Programming Languages
  - ■ Visual C++
    - ✓ Common Tools for Visual C++ 2015
    - ☐ Microsoft Foundation Classes for C++
    - ☐ Windows XP Support for C++
  - ☐ Visual F#
  - ✓ Python Tools for Visual Studio (December 2015)
- ■ Windows and Web Development
  - ✓ ClickOnce Publishing Tools
  - ✓ Microsoft SQL Server Data Tools
  - ✓ Microsoft Web Developer Tools

■ Select All

Setup requires up to 10 MB across all drives.

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
```

with this:

```
try:
    from setuptools import setup
    from setuptools import Extension
except ImportError:
    from distutils.core import setup
    from distutils.extension import Extension
```

From here the installation process will be different for the 2 Python versions. For Python 2 and Python 3.2 or older, I needed to change some of the default values for the C compiler so that I could use it in a developer's prompt or command prompt. Firstly, I opened either Anaconda Prompt or a Developer Command Prompt for Visual Studio 20XX (XX to denote the version of Visual Studio, e.g. if it is 2015 then the XX would be 15). I used both for testing purposes. I then enter the following commands:

```
SET DISTUTILS_USE_SDK=1

SET MSSdk=1
```

When that completed, I changed my directory to the Cython files. The command that I used in Anaconda Prompt and command prompt is “cd”, and in Developer Command Prompt for Visual Studio 20XX it’s “cd /d”. I then built the Cython files using the command:

```
python.exe setup.py build_ext --inplace --compiler=msvc
```

At this statge, I can run either the pyrex files or pyd executable. Commands that I used are below.

To run the Cython files using a Python version newer than 3.2, I downloaded and installed “Windows SDK (.NET 4)”. Once that was installed, I opened a Developer Command Prompt for VS20XX or Anaconda Prompt and entered the following commands:

```
set DISTUTILS_USE_SDK=1
setenv /x86 /release
```

Something to note about the commands above is that it cannot be used in Windows Command Prompt and its very specific to the bit rating. The above shows “x86”, which is for a 32-bit version, which I personal would recommend. The “x86” can be replaced with “x64” for 64-bit versions.

I then used the following commands to run the Pyrex files:

```
python.exe rus.pyx forward
python.exe rus.pyx inverse
```

To run the executable:

```
python.exe -c "import rus" forward
python.exe -c "import rus" inverse
```

Things to note for Anaconda Prompt:

Python 2 is the default Python version but it’s possible to download and install Python 3 by using the command:

```
conda create -n python3 python=3.X anaconda
```

The ’X’ can be replaced with the desired version. This took a while to install but when it completed, I was able to switch between Python 2 and 3. It uses Python 2 by default but when I wanted Python 3, I used the command:

```
active python3
```

to activate it and work in python 3. To switch back to the default, I used the command:

```
deactive python3
```

I really like how the version number is indicated in brackets at all times.

I personally would recommend installing the Cython implementation on either a Mac or a Linux machine. As you can see in the steps above, Windows requires much more steps in order to get it working. You are also be more prone to running into errors, especially during the building process of the Cython files. It is also quite specific to the bit rating of the packages you install, installing the wrong bit rating would result in a lot of unknown problems during the build phase. The setup.py file must also be changed so that the Cython pyrex files can be built correctly on Windows. I personally encountered a lot of problems with Windows and I would not recommend it. The errors are quite ambiguous and can cause unnecessary frustration. One of the errors that I encountered was the following:

```
error: "SyntaxError: Non-UTF-8 code starting with '\x90'
in the file rus.pyd on line 1, but no encoding declared"
```

This was an error that actually stunned me for a few days before I realized that it wouldn't work on the 64-bit version. I spent several hours trying to implement the solutions mentioned online. On the online forums they suggested that I had to declare the UTF coding at the top of the pyrex files. I tried to implement this but ended up with more and more errors.
The general syntax to declaring UTF coding is:

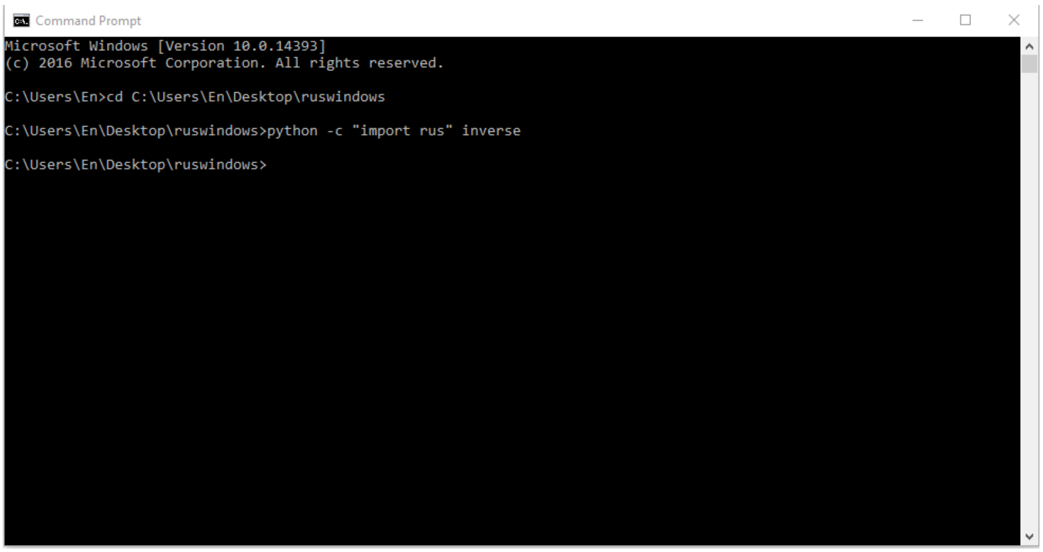
```
# coding=<encoding name>
```

I tried the follow and none of them worked:

```
# -*- coding: utf-8 -*-
# -*- coding: latin-1 -*-
# -*- coding: iso-8859-15 -*-
# -*- coding: ascii -*-
# -*- coding: utf-42 -*-
```

(PEP 263. Defining Python Source Code Encodings. (n.d.))

The most ambiguous error that I got from trying to install Cython on Windows is this one:



```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\En>cd C:\Users\En\Desktop\ruswindows

C:\Users\En\Desktop\ruswindows>python -c "import rus" inverse

C:\Users\En\Desktop\ruswindows>
```

I eventually found out that the error came from an incorrect Cython implementation in the rus.pyx file. Cython does not know how to interpret the following code:

```
if __name__ == "__main__":
```

It must be removed for everything to work. It is a normal Python code so it does not throw any errors during the building phase of Cython. It does not throw any errors during run-time either because it's Python code. This shouldn't be a problem because the rus pyrex files on GitHub works for all 3 operating systems, I wanted to include this just to show how much trouble installing Cython on Windows could be. This was actually a problem that I encountered when I first tried Cython on Windows. Due to it's ambiguity, it was very difficult for me at the time to figure the cause of this problem.

Overall, installing the RUS Cython files on a Mac or Linux machine is the safest and easiest option. I would not recommend running the Cython files on Windows due to it's lengthy installation and error prone nature. I am personally running a virtual machine on Windows with Linux Ubuntu Gnome installed to work on this project.

6 Profiler

During the testing phase of this project, I learnt about profilers. These are built in tools that allow people to test the performance of their code function by function. The results are displayed either on screen or to a output text file. I used Profilers throughout this project, most specifically, the built-in profilers to test the performance of the original C implementation, 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 what the most commonly used functions are 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[18]:

- 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_helper)
       4  0.544   0.136   8.603   2.151  rus_tools.py:230(
          dgamma_fill)
  214350  0.263   0.000   0.263   0.000  rus_tools.py:807(
     volintegral)
       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_fill)
```

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 [one of the headings shown above] e.g -s cumtime
```

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

```
python -m cProfile -s cumtime 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 Cython implementations. 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.

During the second half of this project, I was able to install and the profile the C implementation. I had a bit of trouble with profiling the generated C executable, so I asked Paul for some help. To profile the C code, gprof was required. This involved modifying the make file so that the terminal commands would include gprof and that an output file would be generated with all the profiling results. The make file has the following line of code:

```
CC = gcc
```

Changing this line to:

```
CC = gcc - pg
```

was the main changes made to allow profiling in C. Profiling is such a unique and useful tool that I wish I found out about earlier, especially for one of my courses where I had to write code for an assignment in the most optimized way. If I did not find out about this tool, I would have used print statements that printed out the total time taken to run each function. This approach would not had been as accurate as profilers and would not provide as much detail. I was lucky that my supervisor tasked me to look into this tool, as it has really helped me understand the code better and it has really helped me focus on particular parts of the code. This actually saved me a lot of time because I didn't have to worry about the functions that ran for 0.00 seconds. Wrapping these code would not have made a difference so I was lucky I was able to know this and not had to deal with them.

The profiling is also done on at least 100 iterations of the code so that I can avoid any extra overhead that the profilers could produce when it starts profiling. Running them at 100 iterations each 3 times and then computing the average also makes my profiling results more accurate. Please see the next section for results. Although, there will always be overhead with the use of profilers due to its intrusive behavior, it wont affect my results too much. I am only interested in whether my results are faster or not after wrapping and not really the "true" performance. Comparing the base Python results with the Cython results is all I need to check whether I am on track or not.

7 Performance between C, Python and Cython

7.1 Performance of my Demo

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' because it is one of the most frequently called and slowest function. It was something that I thought would be a good, simple and easy function to show my supervisors.

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.

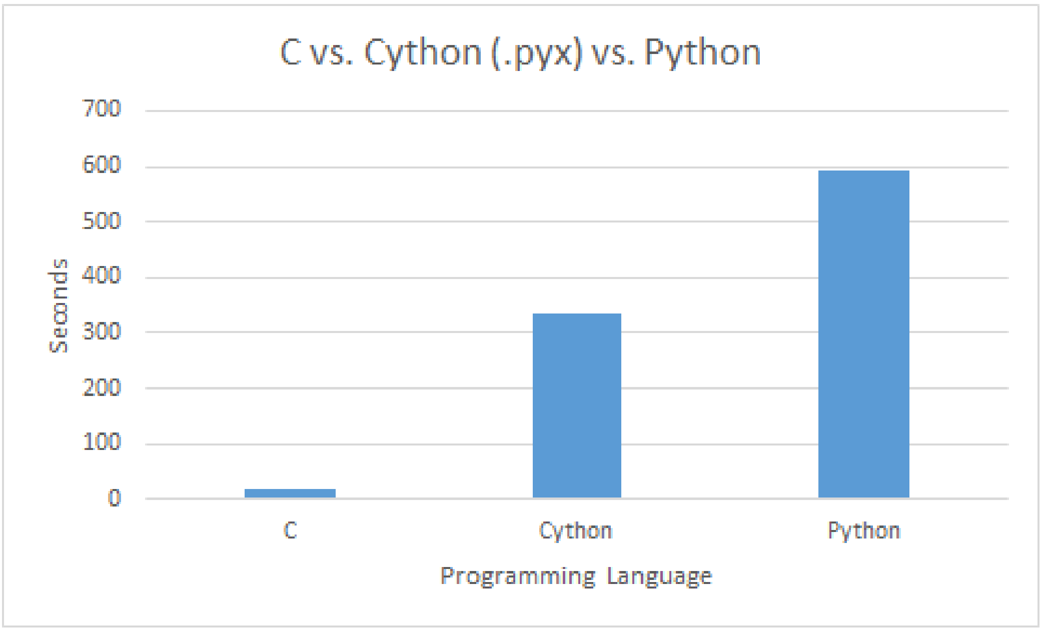
7.2 Final Results and Comparisons

The following shows my final results after wrapping the main functions. I tested the Cython implementation on the same machines so that I could keep the results as consistent as possible. I used two machines, a desktop for Windows and Linux, and a MacBook. I also made sure that my laptop was plugged into a power socket to keep the results consistent. The following are all measured in seconds, are performed at 100 iterations each and are for the inverse code only. I chose to do these tests on the inverse code because it was difficult to show the results for the performance of C's forward algorithm because the times were very low. Also, the inverse code calls the forward code multiple times so showing the inverse code should be enough to get a good representation of the performance increases.

7.2.1 Linux

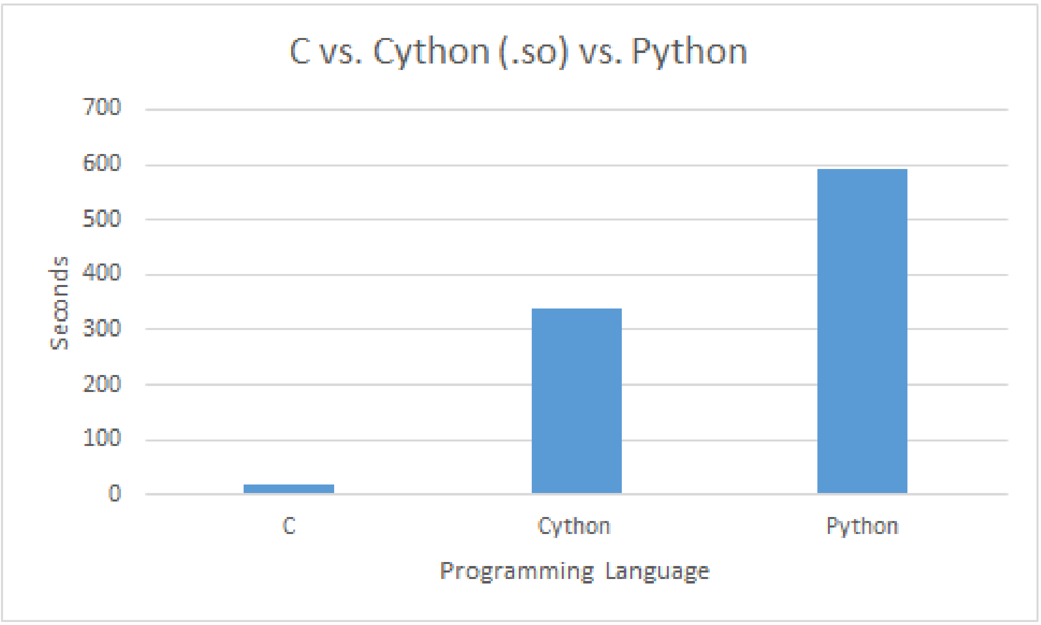
- The wall clock time: C vs. Cython (.pyx file) vs. Python

Overall	Run 1	Run 2	Run 3	Average
C	20.233	20.082	19.681	19.997
Cython	328.165	341.561	338.823	336.183
Python	586.143	598.744	591.379	592.089



- The wall clock time: C vs. Cython (.so file) vs. Python

Overall	Run 1	Run 2	Run 3	Average
C	20.233	20.082	19.681	19.997
Cython	346.459	334.156	338.566	339.727
Python	586.143	598.744	591.379	592.089

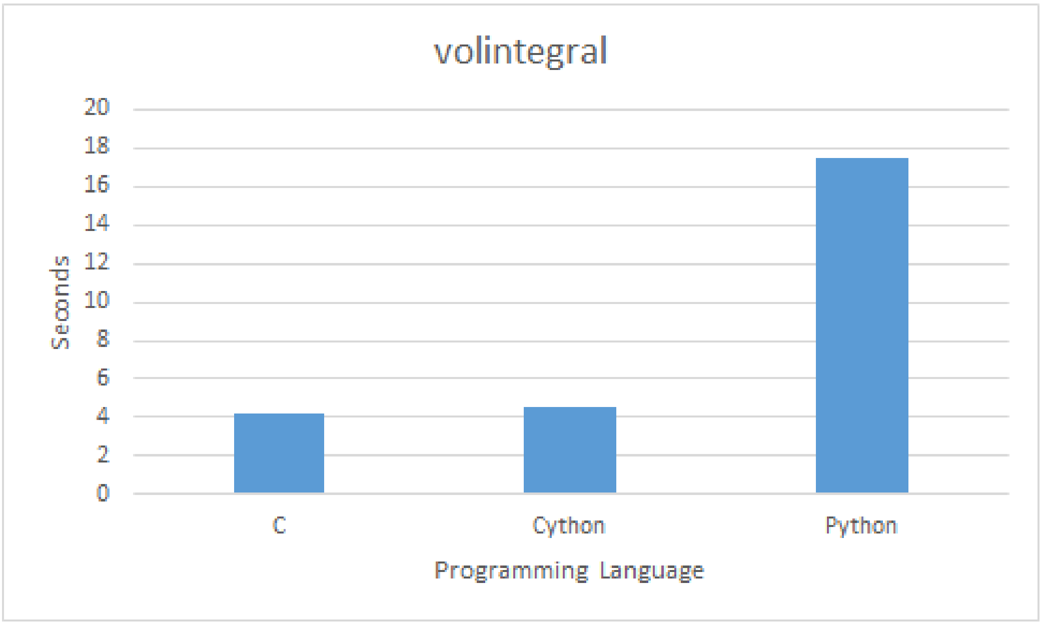


These wall clock results for the overall code performance shows that the Cython code is around 44% faster than the Python implementation. Although it is not as fast as C's performance, it can still help save the end-users quite a bit of time.

Here are some examples of the speed improvements that I have made to the most called functions. I chose to show the following functions because they are the main contributing factors to the performance of the overall code:

- volintegral

volintegral	Run 1	Run 2	Run 3	Average
C	4.300	4.260	4.120	4.230
Cython	4.236	4.980	4.450	4.555
Python	17.090	17.860	17.430	17.460



This is the fastest function that I got from using Cython. The performance of this function is quite close to C and is significantly faster than Python. I think the main reason why this function got the closest to C's performance is because it only performs simple calculations based on formulas. The main computations in the function are also not based on a lot of different arrays and lists. Although it does uses "dimensions", it only uses it to retrieve data and not store data, so it doesn't affect the performance too much.

- The gamma functions

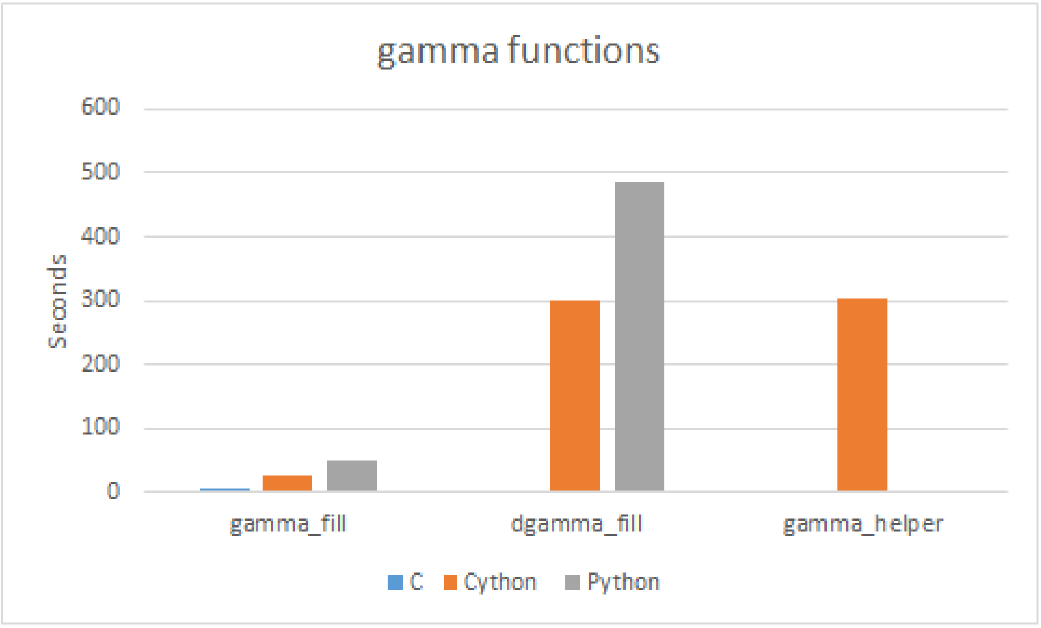
The gamma functions are a bit harder to show because the functions were split into two functions when it was translated into Python. It was split because the code used in "dgamma_fill" and "gamma_fill" use the same code. Paul has made a function called "gamma_helper" that both "dgamma_fill" and "gamma_fill" call. The following are the results for just C and Python for "dgamma_fill" and "gamma_fill":

dgamma_fill	Run 1	Run 2	Run 3	Average
C	1.980	1.910	1.860	1.940
Python	485.112	481.984	490.654	485.917

gamma_fill	Run 1	Run 2	Run 3	Average
C	4.830	4.810	4.830	4.823
Python	47.347	48.165	49.681	48.398

The following are the Cython results

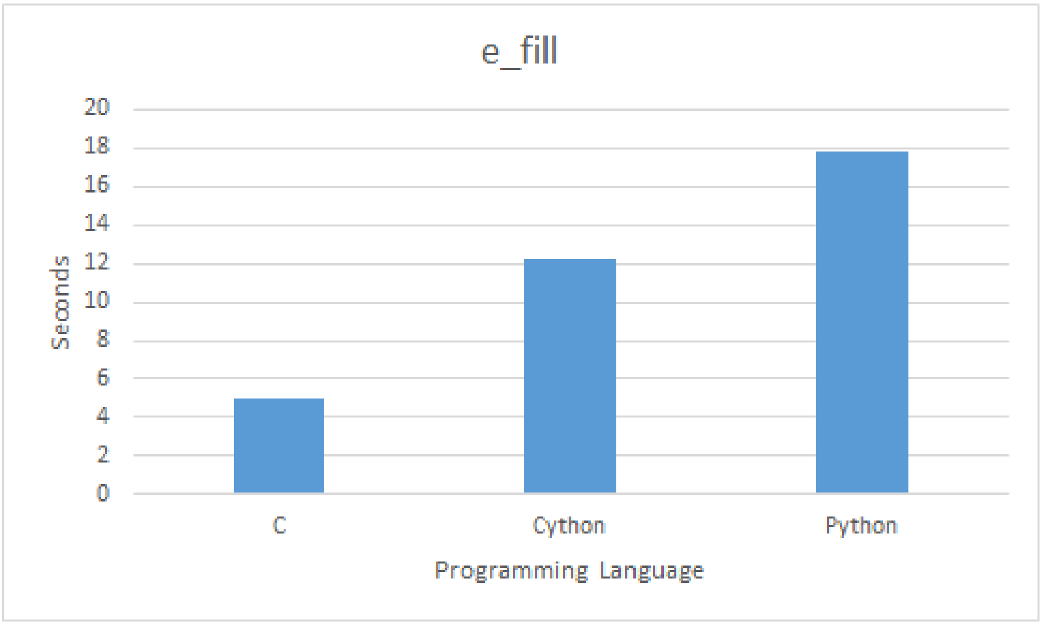
	Run 1	Run 2	Run 3	Average
dgamma_fill	302.288	298.585	300.965	300.613
gamma_fill	26.037	22.619	23.569	24.075
gamma_helper	309.435	301.222	303.213	304.623



The gamma functions did not get as close to C compared to volintegral. By looking at the code, it deals with a lot of computations that involves arrays, e.g. “tabs”, “dimensions” and “irk”. These as mentioned earlier, were hard to wrap and could be the sole reason as to why it didn’t get as close to C. But all the results show that it’s faster than Python.

- e_fill

e_fill	Run 1	Run 2	Run 3	Average
C	4.98	4.98	4.95	4.97
Cython	12.601	11.654	12.485	12.238
Python	17.746	17.658	18.165	17.856

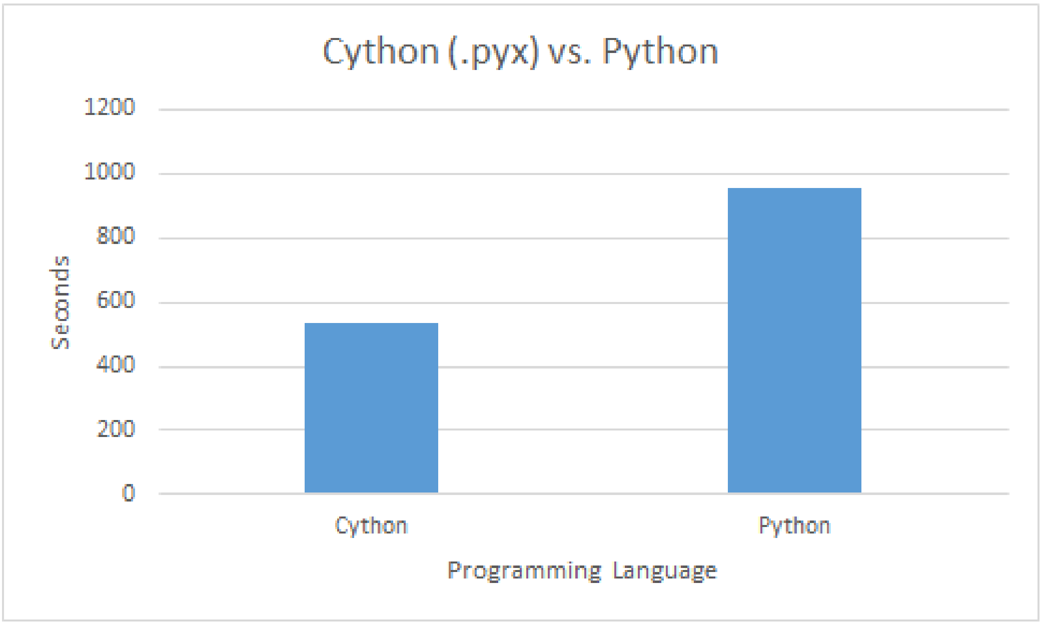


This function showed the least improvement after wrapping the code. This functions deals with a lot of arrays and actually creates its own array, performs computations and stores it into the newly created array. These arrays without wrapping them, would still run at Python's performance and thus shows little improvements.

7.2.2 Mac

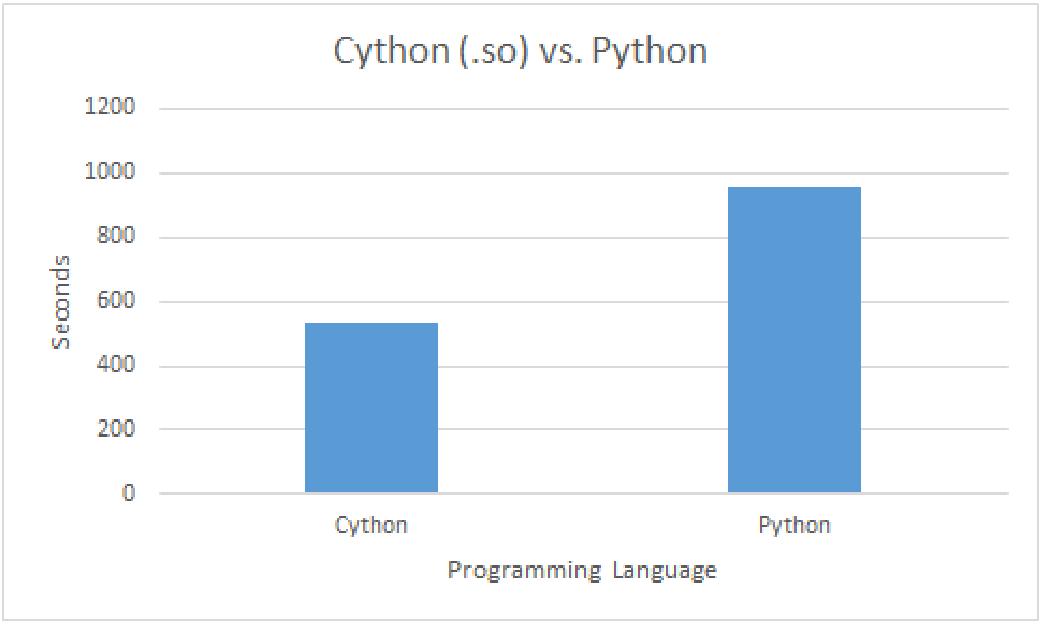
- The wall clock time: Cython (.pyx file) vs. Python

.pyx	Run 1	Run 2	Run 3	Average
Cython	531.656	528.774	536.849	532.426
Python	958.041	957.046	950.054	955.047



- The wall clock time: Cython (.so file) vs. Python

.so	Run 1	Run 2	Run 3	Average
Cython	542.632	523.159	536.038	533.943
Python	958.041	957.046	950.054	955.047

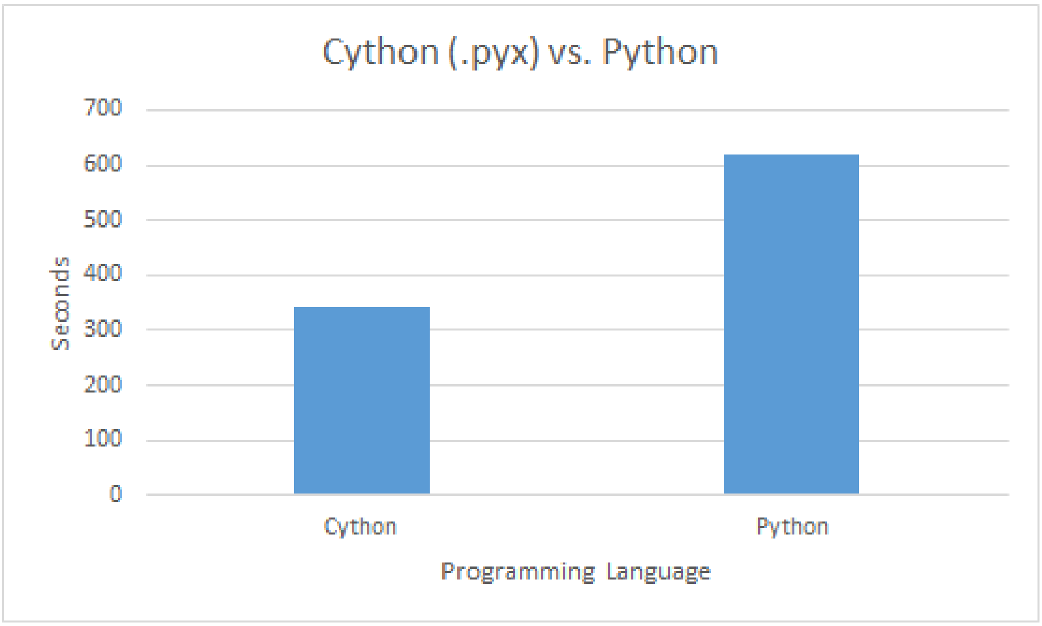


The overall performance of the Cython code on Mac was slower than on Linux and Windows.

7.2.3 Windows

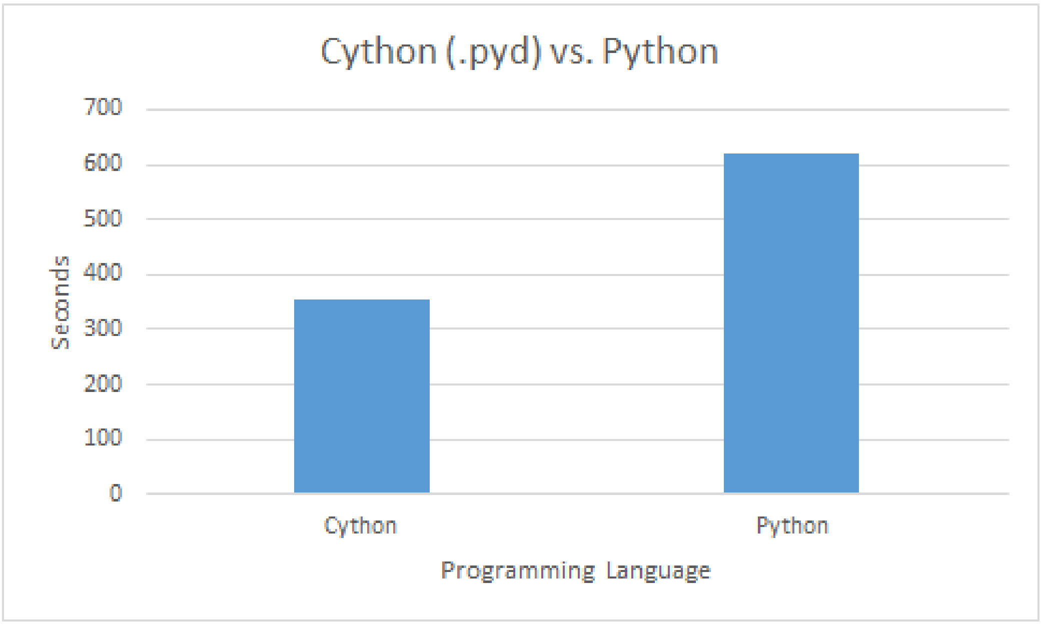
- The wall clock time: Cython (.pyx file) vs. Python

.pyx	Run 1	Run 2	Run 3	Average
Cython	349.627	340.098	336.409	342.045
Python	615.741	626.651	611.843	618.078

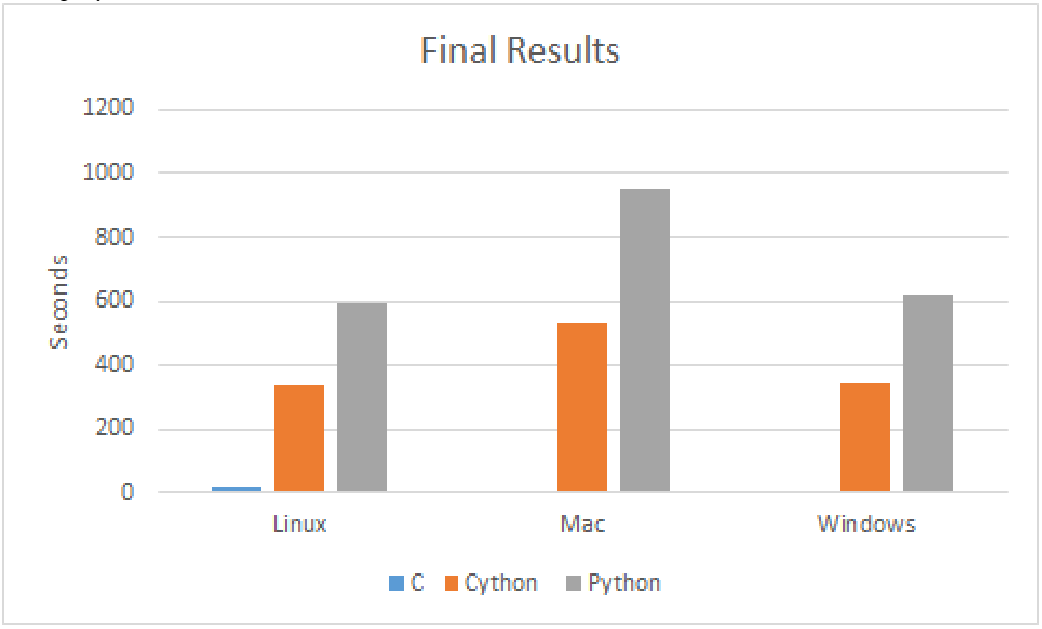


- The wall clock time: Cython (.pyd file) vs. Python

.pyd	Run 1	Run 2	Run 3	Average
Cython	356.599	348.872	357.205	354.225
Python	615.741	626.651	611.843	618.078



In summary, the following graphs shows all the overall times for each operating system:



The C results for Mac and Windows are not applicable as they cannot be installed on those operating systems, hence not showing in the graph above.

Something to note: these are just some of my results that I found. They are not official results. Although only 3 results are shown for each test, I ran several tests throughout the day and picked the fastest results. For some reason the Linux and Windows results fluctuated a bit, but I found that the Mac results were quite consistent. I would say that the final increase in performance is around 44% +/- 2%.

8 Program Compatibility

Mac:

Python Version	RUS	Cython (.pyx)	Cython (.so)
Python 2	✓	✓	✓
Python 3	✓	✓	✓

Although all of the above conditions work, there are a few things that needs to be considered. The above conditions only works if the Anaconda (or equivalent) package is installed. For troubleshooting purposes, if the Anaconda (or equivalent) package is not installed, it will throw the following error:

```
error: "ImportError: No module named 'Scipy'
```

Cython would need to be installed and the files rebuilt if you intend to run the pyrex files. Running them without rebuilding would result in a reduction in performance compared to the Python version. I found that it could possibly run slower than Python's performance if it's not properly built. Running these files without Cython is possible by running the executable (.so file). If the executable are built using Python 2, it will not work on Python 3, it gives the following error:

```
Symbol not found: _PyString_Type
```

This is an anonymous function that is not recognizable by Python 3. I found that it would be best to build it using Python 3 for backwards compatibility.

Also, the files that are built on a Mac OS are specific to Mac OS only, it cannot be used on Linux and Windows.

Linux:

Python Version	RUS	Cython (.pyx)	Cython (.so)
Python 2	✓	✓	✓
Python 3	✓	✓	✓

There are also a few things to note with the Linux version. They all also require the Anaconda (or equivalent) package to be installed for it to work.

For the pyrex files, just like for Mac, would require Cython to be installed and the files rebuilt to see any performance gain. The executable are similar to the Mac, but more specific, the executable built with Python 2 will only work on machines with Python 2 installed. The Python 3 executable will only work on machines with Python 3 installed, but I found that if your machine has Python 3 installed, you can also run the Python 2 built executable by running Python 2 instead of Python 3, e.g.

```
python2 -c "import rus" inverse
```

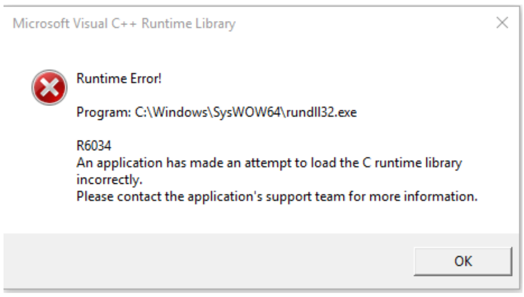
This makes it backwards compatible if you only have access to the Python 2 version of the executable.

Windows:

Python Version	RUS	Cython (.pyx)	Cython (.pyd)
Python 2	✓	✓	✓
Python 3	✓	✓	✓

Installing and running Cython on Windows was far more difficult compared to Mac OS and Linux. It involves more steps and modifying some of the code to get it to work, these are all outlined in the "Installing and Running the Cython Implementation" section under the Windows subsection. Again, Anaconda or something equivalent is required for any of the RUS implementations to work. The RUS code works on Windows provided that either Python 2 or 3 are installed, but this depends on which version you prefer or already have installed. The Pyrex files work as well but it requires that the Cython pyrex files to be rebuilt in order to benefit from the increased performance. The executable do work but Python and Anaconda (or equivalent) has to be installed for it to work. Cython is not required if you're just running the executable.

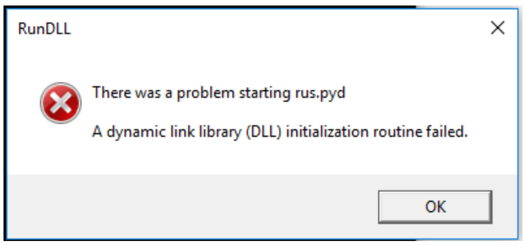
Something else to note with running the .pyd executable on Windows, its not a normal .exe executable file where you can simply double click to get it work. It can only be run from a command prompt or a develop prompt using the proper commands which are mentioned earlier in this report. Some people online has said that the .pyd files are simply .dll files, but running it like a .dll file wont work either and would result in the following errors:


Microsoft Visual C++ Runtime Library

Runtime Error!

Program: C:\Windows\SysWOW64\rundll32.exe

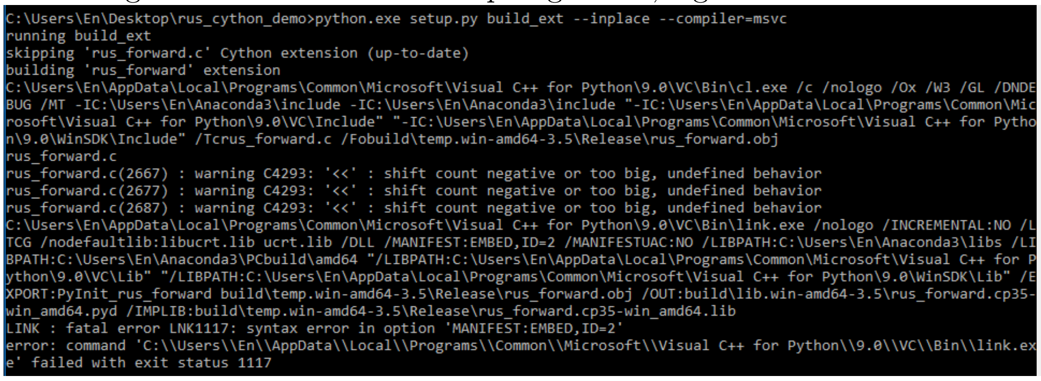
R6034
An application has made an attempt to load the C runtime library incorrectly.
Please contact the application's support team for more information.


RunDLL

There was a problem starting rus.pyd

A dynamic link library (DLL) initialization routine failed.

Overall, I think that it's best to run the Cython files on a Linux machine or on a Mac. Although they all do work, the Windows installation involves more steps during the installation phase, you have to modify the code and you're more likely to run into problems if you installed the wrong version, e.g. installing the 64-bit version rather than the 32-bit. I found that installing the wrong version causes some compiling errors, e.g.


```
C:\Users\En\Desktop\rus_cython_demo>python.exe setup.py build_ext --inplace --compiler=msvc
running build_ext
skipping 'rus_forward.c' Cython extension (up-to-date)
building 'rus_forward' extension
C:\Users\En\AppData\Local\Programs\Common\Microsoft\Visual C++ for Python\9.0\VC\Bin\cl.exe /c /nologo /Ox /W3 /GL /DNDE
BUG /MT -IC:\Users\En\Anaconda3\include -IC:\Users\En\Anaconda3\include "-IC:\Users\En\AppData\Local\Programs\Common\Mic
rosoft\Visual C++ for Python\9.0\VC\Include" "-IC:\Users\En\AppData\Local\Programs\Common\Microsoft\Visual C++ for Pytho
n\9.0\WinSDK\Include" /Tcrus_forward.c /Fobuild\temp.win-amd64-3.5\Release\rus_forward.obj
rus_forward.c
rus_forward.c(2667) : warning C4293: '<<' : shift count negative or too big, undefined behavior
rus_forward.c(2677) : warning C4293: '<<' : shift count negative or too big, undefined behavior
rus_forward.c(2687) : warning C4293: '<<' : shift count negative or too big, undefined behavior
C:\Users\En\AppData\Local\Programs\Common\Microsoft\Visual C++ for Python\9.0\VC\Bin\link.exe /nologo /INCREMENTAL:NO /L
TCG /nodefaultlib:libucrt.lib ucrt.lib /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO /LIBPATH:C:\Users\En\Anaconda3\libs /LI
BPATH:C:\Users\En\Anaconda3\PCbuild\amd64 "/LIBPATH:C:\Users\En\AppData\Local\Programs\Common\Microsoft\Visual C++ for P
ython\9.0\VC\Lib" "/LIBPATH:C:\Users\En\AppData\Local\Programs\Common\Microsoft\Visual C++ for Python\9.0\WinSDK\Lib" /E
XPORT:PyInit_rus_forward build\temp.win-amd64-3.5\Release\rus_forward.obj /OUT:build\lib.win-amd64-3.5\rus_forward.cp35-
win_amd64.pyd /IMPLIB:build\temp.win-amd64-3.5\Release\rus_forward.cp35-win_amd64.lib
LINK : fatal error LNK1117: syntax error in option 'MANIFEST:EMBED,ID=2'
error: command 'C:\\Users\\En\\AppData\\Local\\Programs\\Common\\Microsoft\\Visual C++ for Python\\9.0\\VC\\Bin\\link.ex
e' failed with exit status 1117
```

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

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].

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[13].

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"
    - Set variable a's PyObject_HEAD's "typecode" to an integer
    - Set variable a's "val" to 1
2. Assign 2 to b
    - Set variable b's PyObject_HEAD's "typecode" to an integer
    - Set variable b`s "val" to 2
```

```
3. Call binary_add(a, b)
    - Find the "typecode" of a from a’s PyObject_HEAD
    - When the variable type is found, retrieve a’s "val"
    - Find the "typecode" of b from b’s PyObject_HEAD
    - When the variable type if found, retrive b’s "val"
    - Call binary_add<int, int> (a->val, b->val)
    - After the addtion, store the results in "result"
4. Create a Python Object "c"
    - Set c’s PyObject_HEAD’s "typecode" to an integer
    - 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 GitHub

I started using GitHub Desktop to start learning the processes of how GitHub works. I found that GitHub Desktop was a bit difficult to use as there wasn’t much feedback. Most of the time, I spent searching online for solutions as I wasn’t sure if I had done it correctly or not. For example, when I click on “commit to master”, nothing happens after that so it was unclear whether I’ve done it correctly or not. The first time I got it to work, I ended up in the wrong branch due to the lack of feedback. Also, I was not able to see any of the files on GitHub Desktop, nor could I edit the files on the application. It just detects the changes and highlights the differences. If I had to change something, I had to change the files, then copy it to the GitHub Desktop directory and then commit the changes. I could have changed the directory of GitHub Desktop to the RUS files but I had to work on a lot of different machines so it wasn't centralized.

I found that Git Bash or a simple terminal are easier to use because you can edit the files using "vim" and there are more feedback after each step. I also knew exactly which branch I was committing to and I could also see all the files I was working with. It was also colour coded when something is to be committed or not. For example, when there are new files that needs to the committed, it will be green and when it is committed and pushed, then it will become red when you request for the status.

What I would recommend to others that are new to GitHub is to just dive into using Git Bash rather than an GitHub application like GitHub Desktop. Although GitHub Desktop looked easier, I found that it was much harder to use compared to Git Bash. Git Bash definitely looked harder initially but when I started to use it, I found it to be easier to use. Something that I regretted was not using Git Bash from the start, while I was learning how to use GitHub Desktop, I got it to work, but I didn't actually learn the whole concept of how it worked in the background. With Git Bash, I could see every step and actually understand what I was doing.

There were a few things that I hated about GitHub, these were the terminologies and concepts that they used. I found that the terminologies and concepts used were quite confusing at times. For example, push, pull, commit, clone, branches, master branch, my local branch, etc. I initially thought that commit means that I was saving the files and updates directly to GitHub, but later found out that commit only saves the changes and push is actually the one that pushes it onto the GitHub page. After pushing it, I thought it was on the page but had no idea I had to request a pull request so that the admins or owners was able to accept these changes. Something that took me a while to figure out was the fact that when I push the files to GitHub, I was actually pushing to my own account's version of the RUS code rather than the actually RUS GitHub branch. This is when I actually started to understand what a pull request was that actually transfers the files from my own account to the RUS account.

11 Weekly Log

Semester 1 Week 1 (Week starting 29 February 2016)

During the first week, we had a kick-off meeting with the Btech Information Technology coordinator Sathiamoorthy Manoharan (Mano). We discussed what we had to do and what was expected from us from the projects. Mano gave us the option of finding our own projects or choose one from a list that he gave us. I decided to choose a project from the list of eight that he sent out. By the end of the week, Mano assigned me with the Resonant Ultrasound Spectroscopy project with Kasper van Wijk and Paul Freeman as my supervisors.

Semester 1 Week 2 (Week starting 7 March 2016)

I had my first meeting on the 9th of March 2016 with my academic supervisors Kasper Van Wijk and Paul Freeman. I also met Elvis Chuah who is the other Btech student that is working on this project. During this meeting we discussed what the project is about and what we would like to contribute to the project. We mainly discussed the Graphical User Interface (GUI) and optimization aspects of the project. I ended up choosing the optimization side of the project, where I had to use wrapper functions to wrap C code within Python to increase the performance.

Semester 1 Week 3 (Week starting 14 March 2016)

We had our second meeting this week, this meeting consisted of our supervisor Kasper van Wijk giving us a demonstration in one of the physics labs of how RUS worked. After the demonstration/ meeting, I was in charge of writing and sending out an email that outlined what occurred during that meeting. The email that I sent out was:

“The following is a summary of what we did today in our second meeting:
Kasper gave us a demo of what RUS is so that we could have a better understanding of the project. My understanding of what we did today (please correct me if I’m wrong) is that through the experiment we were able to determine the properties of a solid object without destroying it. With the use of 2 transducers we were able to send signals with different frequencies into the solid object and measure the resonance of the solid object. What we are interested in for our project is the resonant amplitudes.

My goals for this week is to:

- Further my understanding of RUS with what I learnt today and additional research
- Get a better understanding of Paul's code
- Look at examples online on how to wrap C code within Python (through Cython? I can't remember what Paul recommended to look at)

We will have another meeting at the same time next week (Tuesday 22 March at noon)"

As a bit of homework, I started doing some independent learning of what RUS is and also started looking at the python implementation of RUS.

The following are my final goals and suggested dates of completion:

1. Research and recommend a Python method for wrapping C code (from Cython and other options). (goal: 5 April)
2. Read through all Python code in the repository (maybe make a list of which functions you understand and which you don't). (goal: 22 March)
3. Check out your branch of the GIT repository (goal: 30 March)
4. Further research into RUS (goal: ongoing)

Semester 1 Week 4 (Week starting 21 March 2016)

What I have done up to this point is research on RUS for better understanding and I've also gone through the Python implementation of RUS. I had some trouble understanding some of the code but Paul Freeman explained it to me. I also started cloning the GitHub respiratory to GitHub Desktop and started writing my Btech report. So far, I've written the introduction.

Semester 1 Week 5 (Week starting 28 March 2016)

My goal for this week is to do some research on how I should optimize the code using wrapper functions. I ended up choosing Cython because it allows both C and Python code in one file. It also generated an executable which I found could be quite useful. There were also quite a few forums online with people talking about how fast Cython was, which made my decision easier.

Semester 1 Week 6 (Week starting 4 April 2016)

This week, I described the tool that I'm going to be using for the project and started doing research and learning the basics.

Semester 1 Week 7 (Week starting 11 April 2016)

After learning the basics, Paul wanted me to continue learning Cython and to write a demo to show how it works. I also looked at Python profilers so that I could test and find out which parts of the Python code were the slowest.

Semester 1 Mid-semester break (Week starting 18 April 2016)

No meeting and mid-semester break. I was preparing for my introductory seminar that I will be presenting next week.

Semester 1 Week 8 (week starting 25 April 2016)

Continued to learn Cython and started to learn about Python profilers and implemented it. I also had my introductory seminar this week.

Semester 1 Week 9 (Week starting 2 May 2016)

We had another meeting and I presented the output of the Python Profiler implementation. This week I had to write my demo and present it next week. I was told to wrap one of the RUS functions for the demo.

Semester 1 Week 10 (Week starting 9 May 2016)

No meeting – Kasper was busy. I continued to work on my demo. My demo took longer than I originally thought to complete. I had a lot of complications compiling the Python code into Cython. New due date for my Demo is next week. I have also set a goal to complete a draft of my outline to my mid-year report. I also wrote a few paragraph of my mid-year report this week.

Semester 1 Week 11 (Week starting 16 May 2016)

This week, I presented my demo to me supervisors. I think they were happy with the performance of 'volintegral', I got it about 4 times faster than the original Python implementation. I also looked into and learned how to use GitHub so that I could check my code onto GitHub, but I ran into some trouble with some of the concepts of GitHub.

Semester 1 Week 12 (Week starting 23 May 2016)

During the meeting this week my supervisor, Paul, helped me with and explained GitHub. Now my demo is on GitHub.

Semester 1 Week 13 (Week starting 30 May 2016)

For the final week before the break, I was working on the code and Python profilers, so that I could test the performance of my wrapped code as I wrapped them. I couldn't profile the C implementation due to the difficulties of installing the C version, but my supervisor has sent me some results so that I could use them in my report and in my mid-year seminar.

————————————Inter-semester break————————————

During our inter-semester break, I worked on wrapping a bit more code and preparing for the mid-year seminar. We had our mid-year seminars on the 15th of July.

Semester 2 Week 1 (Week starting 18 July 2016)

This was the first week back from the inter-semester break, this week we didn't have a normal meeting because my supervisors wanted to discuss my mid-year report mark. I got some feedback and suggestions on how to fix some of the problems discussed during meeting. Semester 2 Week 2 (Week starting 25 July 2016)

We continued with our weekly meetings this week. This week I was tasked to check the compatibility of my code on the 3 different operating systems and also on Python version 2 and 3. Semester 2 Week 3 (Week starting 1 August 2016)

I presented my compatibility results to my supervisor. There were some issues with Windows and I spent this week trying to fix those error. I managed to get Cython installed and my code to build on windows. It generated all the necessary files, but unfortunately the executable could not run. Running the pyrex file was possible.

Semester 2 Week 4 (Week starting 8 August 2016)

During our meeting this week, our supervisor gave us a demo on how to use GitHub via a terminal. There was a bug fix that we needed to get from the PALabs/RUS repository into our code. Semester 2 Week 5 (Week starting 15 August 2016)

I started working on installing the C implementation so that I could run some tests. I also found that directly moving my code from one machine to the next without rebuilding caused the code to be running at speeds that were similar or slower than Python's implementation. This was a major problem that affected the how portable my code was. Semester 2 Week 6 (Week starting 22 August 2016)

This week I compared my codes performance to that of C's and I found out that some of the functions weren't as fast as C's speed, which was a big problem. I then started looking into why this is.

Semester 2 Mid-semester break (Week starting 29 August 2016)

During the mid-semester break, I tried to improve the speed of the already wrapped function. Unfortunately, I was still unable to improve the performance. I tried to implement an Cython array as the Python used a lot of different 2 dimensional arrays. This is where I found that implementing a simple array in Cython deemed more difficult than anticipated.

Semester 2 Week 7 (Week starting 12 September 2016)

This week I told my supervisor about the problem I have with the arrays. I still continued to work on it but my supervisor has suggested that the current performance could be the limit.

Semester 2 Week 8 (Week starting 19 September 2016)

I continued to try and improve the performance of the code but still could not improve the speed. My supervisor suggested I inline the functions from rus_alloc.c, but this resulted in a lot of errors due to Cython's limitations with C and Python objects. I also started working on writing my final report and thought about how I would test and present my results.

Semester 2 Week 9 (Week starting 26 September 2016)

I gave my self 2 days to try and improve the final codes performance but I had no break through. My supervisors has looked at my code and he has suggested that it may not be worth the time and effort because the increase in performance wont be significant. Since I was not able to improve the performance, I focused on my report. I took out a few sections of my mid-year report that I found was not too relevant to my final report and condensed my report down. This week I added in 4.5 pages of new content and the final word count was 6,004.

Semester 2 Week 10 (Week starting 3 October 2016)

This week I worked on getting my code into something that would be presentable and check it onto GitHub. I also worked on my final report, my current word count is 10610. When I was reintalling the Cython code for the installation of Cython section of this report, I was able to fix the executable problem that I had from a few weeks ago. It is not working fine with both the pyrex and executable files.

Semester 2 Week 11 (Week starting 10 September 2016)

I focused on my final report because my supervisors want a draft by the 17th of September so that I could get some feedback before the final report was due. I did a lot of testing this week for the performance section of my report.

Semester 2 Week 12 (Week starting 17 September 2016)

We had our final meeting this week to wrap up the whole project. This week I prepared for my seminar which I will be presenting next week. I also worked on my final report, which is also due next week.

12 Weekly Meetings

We had weekly meetings throughout this project. During each meeting we discussed the issues we had, what we did in the last week, what we wanted to do the following week and set some short or long term goals. I found that having these weekly meetings kept me on track with this project. Knowing that I had the meetings and deadlines to meet each week kept me motivated and on track with the workload. Without these weekly meetings, I think would fall behind on the work really quickly. Having these weekly meetings also gave me some experience with working in a team where we discussed and shared ideas, which is going to be very useful in life and especially when I start working. Ultimately, I found that having these weekly meetings gave me some structure and increased my overall productivity.

I also learned new things during these meetings because my meetings also included the other BTech student, Elvis, where he tackles the GUI aspects of the project. Although I am not working on the GUI aspects of the project, I can see his progress and actually learn a few things about the GUI aspects. Listening to what he had to say or the questions that he had also helped with my project because it gave me another perspective on how a specific code worked. Also, explaining or sharing what we had both learnt during some of the meetings also enhanced my understanding. Sometimes, my supervisors would ask me specific questions that would actually challenge my understanding further. I did not always know the answer but I tried my best to find out and present my findings in our next meeting.

These meetings were not too long nor too short. On average, they usually last around 15 minutes, which is very manageable each week. After each meeting, we send out a summary of what we talked about and what our goals were. I found this very useful when we started to write our mid-year or final reports because it helped with the writing process. Reading through these weekly summaries helped remind myself of the different ideas we discussed, what we did and what we learnt. I also had some of my problems along with some screenshots in the emails which I used in this report. These are all shown in the "Weekly Log" section of this report and it has shown the amount of work I am have done throughout this project.

These meetings also help improve my confidence. I am a very quiet and introverted person that doesn't really work well in a team, but having these meetings every week has helped with this. Each time we had our meetings I try to push myself into talking more and sharing my ideas. I found that I got more confident and comfortable after each meeting progressed, and I was able to share more ideas and ask more questions.

There were also some demonstrations from Paul on how to use GitHub during our meetings, which I found to be very useful because neither I nor Elvis has really used GitHub before. There was a demonstration before we started using GitHub followed by some instructions in the emails he sent out. We then attempted them and we asked questions the following week if we didn't understand it. He also showed us or fixed any problems we encountered during the next meeting(s).

As mentioned earlier, during each meeting, we are required to set some goals for the up coming weeks, these goals are quite specific and do not exceed 3 or 4 weeks. I found this quite useful because I was not stuck with doing the same thing each week, each week usually consisted of something new I had to do or something that I had to learn with some existing material.

I would highly recommend having weekly meetings in every project that we encounter. It doesn't matter if it's a large or small project, having weekly meetings would make the project run smoother with a better outcome in the end. When we first started this project, I personally was terrified about the idea of having weekly meetings. There were several reasons why:

1. I hated that it was every week
2. I thought that communicating online would be better
3. I was worried about the criticism that I would get weekly on my work
4. I thought it would get repetitive after a few weeks

Let me elaborate on these points, 1. I was worried that there was going to be too much work that I had to complete before the next meeting and that I would struggle with the workload. What I have discovered after a few weeks were that the tasks for each week were flexible. If I was busy that week or if I had other commitments, then I could talk to my supervisors and work out something that would mutually work. 2. Initially, I thought communicating online was better but I quickly found that meeting in real life was a better approach. There were things such as the peaks on the graphs that would be better explained on a whiteboard in person than online. 3. I've always feared criticism on my work and I would always avoid it when I could, but I've found that criticism has helped improve my work and it has opened my eyes to a new perspective to tackle the problem. Sharing my ideas with my supervisors to see what they think, I found actually gave me confidence in my work. I remember after my first demo, my supervisor was really happy with the speed that I got after wrapping one of the functions that actually motivated me. 4. I thought that meeting weekly when our work was due in a week or 2 was going to get repetitive but I found that even though our work was set for a couple of weeks to complete, I could tell them or show them what I've done so far. This not only confirms whether I am on the right track or not but it also gives me another perspective on how to solve that problem or what the most efficient way to approach the task is.

From these points, it has shown that weekly meetings are very effective and it can help with the tasks that I got. Although there were things that I terrified or was worried about, these weekly meetings has changed me and given me a bit more confidence when doing projects or group meetings. It has given me confidence because I was able to step out of my comfort zone and experience first hand what it felt like working on a project in a group. Since the meetings were every week, I felt that I got more and more practice that lead to more confidence. I still have a long way to go before I am very confident in working, talking or leading a group but this has shown me that I should always step out of my comfort zone and challenge myself to do better. If I had the choice of choosing whether I would have weekly meetings or not, I would definitely choose to have them.

13 What I wished I did differently

When I started this project, one of my goals for the first few weeks was to understand the Python code. Something that I should have done as well was to try and understand the C code. Maybe if I had a better understanding of the C code, I would be able to detect that Cython may not have been the best tool to use for this project. Also, understanding the C implementation early could have helped me wrap the code faster. I remember spending most of my time trying to figure out and understand how the C implementation worked rather than the actual wrapping itself.

As a requirement for BTech, we are all required to do 3 seminars, an introductory, mid-year and final seminar to showcase our progress throughout the project. I think a lot of the Btech students would agree that we should cut out the seminars part of the requirements, but I found that there are some advantages in doing seminars, these are they help build my confidence, I got to learn new things from other BTech students, I got a better understanding of my own project and I got some criticism from other supervisors. During the questions and answers part of my mid-year seminar, one of the other supervisors suggested/hinted that it would be impossible for me to improve the speed of Python to C's performance, this was something that I did not believe at the time and is something that I now regret not considering as a possibility. I chose to ignore what he said due to what I read online on Cython and what I experienced when I was working on my demo for my supervisors. When I was writing my demo I was able to wrap a function that got really close to C's performance, which is actually the closest I got to C's performance throughout my entire project. This actually gave me some confidence that I was able to wrap the whole code and this might have been the biggest reason why I chose to ignore him. Something that I should have done was wrap a function that performed more complicated operations rather than simple formulated calculations for my demo.

I really wished that I installed the C implementation and profiled it earlier on in the project. I was blindly wrapping the C code within Python without actually having a goal to work towards in terms of the performance increase for each function. After wrapping some of the functions, I thought that they would be the limit, but I was wrong. When I profiled the C implementation in semester 2, I found that a lot of the functions that I have already wrapped weren't as fast as I had hoped. The main reason why I didn't implement C implementation earlier was due to time constraints and because of the level of difficulty to install and run. I made it a goal to get the C implementation installed and running in semester 2. By this time the C implementation was much easier to install because my supervisor, Paul, created a new version of the C implementation by cleaning up the dependencies that the original implementation had. This was done during the inter-semester break and I was able to install it during semester 2.

During my initial wrapping of the functions before I compared the performance results to C, I ignored the creation of the arrays phase as they were implemented using numpy.array. The reason why I chose to ignore them were because I read that creating arrays using numpy.array was quite efficient compared to creating them in Python. My initial thoughts were that they were quite close to C's performance, but I have now seen that it isn't. As described earlier, creating arrays in Cython was extremely difficult and time consuming, if I had attempted to create these arrays using Cython earlier on in the project, I could have told my supervisors that it would be impossible to achieve C's performance early and looked into other solutions instead. I only found out that the numpy.array were not fast enough when I installed the C implementation, profiled it and compared it with Cython's performance. By this time, it was roughly halfway through semester 2 and looking at alternative solutions would not have been a good idea so I kept attempting to implement the arrays in Cython.

I looked into several tools that would help improve the performance of Python, these are all listed in the alternatives of Cython subsection. When I was looking into these tools, I should've taken some time to actually implement each tool into the RUS code. This would have taken me longer to do but if I had, I could have easily identified which tool worked best instead of finding that out halfway through the project. Learning these different tools early on in the project could have also helped me write better and efficient Cython code as all these tools are fairly similar to Cython.

I spent most of the second half of the project trying to find different ways to implement different aspects of the already wrapped code so that I could improve the performance further. My supervisor also hinted that the performance of the already wrapped code could just be the limit and trying to implement different aspects of the code, for example, the arrays, wouldn't of had a huge impact on the performance. This was something that I should have listened to, if I had listened and didn't try to implement the code further, I could have made more progress in terms of the project in the second half.

There were also a lot of different formulas that were involved, although I didn't necessary needed to understand the formulas used, I believe that it could have helped. Cython alone could be considered a language on its own and I think that if I was able to understand how all the formulas worked, I could have written more efficient code, by moving the code or by changing how each function was called. Most of the formulas were just copied and pasted from C or slightly tweaked without any real knowledge of how they worked.

14 Further implementations or improvements

Since I spent most of the second half trying to improve the performance of the already wrapped code, I ended up wasting a lot of time which I could've used to help further develop the RUS project. There were some things that I wanted to do for the project:

- Create a Makefile
  - After being exposed to the Makefile when I as trying to profile the C implementation, I really wanted to learn and try to implement it for Cython. I ran out of time so I wasn't able to learn and implement it.

- Dimensions problem
  - Kasper mentioned that there was something wrong with the dimensions as parameters in the code. This is where some of the dimensions variables weren't used but they were still shown or being used in some way. This is something that I could've helped with if I had extra time at the end of the project.
- Testing aspects of RUS
  - During our early meetings for this project, we had a choice between 3 parts of the project. These were the performance, GUI and testing parts. The testing part could have been something that I could've started on if things went according to plan and if I had some free time at the end. This part was to make sure that the code did what is was intended to do. This would have involve actually performing the physics experiment in the labs and comparing the results.

15 My Contribution to RUS

The Cython implementation has improved the performance of the Python implementation and has made it easier to distribute and use. As you have already seen in the performance section of this report, the Python implementation runs for 592.089 seconds at 100 iterations, which could be costly to the end users. The end users are likely to be geoscientists, these geoscientists run and test multiple samples each day, the more samples that they can run in the day, the more that they can accomplish. The Cython code is about 44% faster than Python so the geoscientists would roughly double their productivity. Although the newer C implementation is now easier to install, the Cython implementation is much easier to distribute and use due to its enhanced portability. It's executable can be used on any system that these geoscientists use, as their systems would definitely have Python and a package like Anaconda installed. Since it's an executable, no other installation is required, the geoscientists can just download and execute using the proper commands.

16 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. It also generates an executable which could be very beneficial for portability reasons.

Initially, before I started any coding/wrapping, I thought that after wrapping the functions, I would be able to get an overall time that was closer to the performance of C. In the end, I was not able to reach C's performance but I did get the code to run faster than the Python implementation. I am a little disappointed that I was not able to get it as fast as C's performance but I tried my best. Maybe if I had chosen another tool or had more time to implement the arrays in Cython, I would have been able to get something faster than what I have now.

Although the performance of the end result was not what I had hoped, I did contribute something to the RUS project. There is now something in between the C and Python implementations. I have created something that combined the best of both C and Python. I have increased the performance of the Python implementation to something closer to the C implementation and I've also made it easier to install which is what the translation into Python version was for.

Overall, I really enjoyed the project and I've definitely learned a lot of new skills that I will definitely take into the future. I feel that after completing this project, I have also developed and improved as a person. I used to find it really hard to work with people, communicate with people and present in front of people, but I've improved quite a bit this year. I can honestly say that this project has really helped with this and I'm very grateful. I couldn't have done this without my supervisors Paul Freeman, Kasper van Wijk and Sathiamoorthy Manoharan (Mano). Thank you very much for the constant support throughout this year. I really appreciate it.

17 Bibliography

1 Leisure, R. G., & Willis, F. A. (1997). Resonant ultrasound spectroscopy. Journal of Physics: Condensed Matter, 9(28), 6001.

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

5 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.

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

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

8 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

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

10 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

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

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

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

14 Boost.Python - 1.62.0. (n.d.). Retrieved October 12, 2016, from
http://www.boost.org/doc/libs/1_62_0/libs/python/doc/html/index.html

15 Basic Tutorial — Cython 0.25b0 documentation. (n.d.). Retrieved October 22, 2016,
from http://cython.readthedocs.io/en/latest/src/tutorial/cython_tutorial.html

16 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

17 PEP 263 – Defining Python Source Code Encodings. (n.d.). Retrieved October 24, 2016, from https://www.python.org/dev/peps/pep-0263/

18 26.4. The Python Profilers — Python 2.7.11 documentation. (n.d.). Retrieved from https://docs.python.org/2/library/profile.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.). Retrieved from http://nationalpubliclibrary.info/articles/Compiled_language

22 Cython def, cdef and cpdef functions Documentation. (n.d.). Retrieved October 18, 2016, from https://media.readthedocs.org/pdf/notes-on-cython/latest/notes-on-cython.pdf

18 Appendix

A. Example of Cython Code

C implementation:

```
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
```

B. Function: 'index_relationship'

```
cpdef index_relationship(int *itab, int *ltab, int *mtab,
                                int *ntab, int d, int *irk):
    """
    Creates and returns tabs and irk data.

    tabs and irk are used in future functions.
    This function populates them based on the
    values of d.

    TODO: Improve description of what this
    function is actually doing and why.
    """

    #ORIGINAL
    # Since this function creates and returns 'tabs', I will be
        modifying the following:
    # tabs = numpy.zeros((int(problem_size), 4), dtype=numpy.
        int64)

    # cpdef int tabs[problem_size][4]; GIVES ERROR: "Not allowed
         in a constant expression" - slow?

    # cdef int *tabs = <int *>malloc(problem_size * 4 * sizeof(
        int));

    """
    This works but type error because when called from e_fill,
        it requires an int.
    cpdef list[:, :] tabs = numpy.empty((problem_size, 4), dtype
        =list);
    http://stackoverflow.com/questions/19054756/arrays-of-arrays
        -in-cython
    """

    """
    http://stackoverflow.com/questions/25974975/cython-c-array-initialization
    Example:
    cdef int mom2calc[3]
    mom2calc[:] = [1, 2, 3]

    cdef int mom2calc[3][3]
        mom2calc[0][:] = [1, 2, 3]
        mom2calc[1][:] = [4, 5, 6]
        mom2calc[2][:] = [7, 8, 9]
    """
    #irk  = [0 for i from 0 <= i < 8] not needed for now

    # Separating and recreating 'tabs'

    """
    cdef int itab[r];
    cdef int ltab[r];
    cdef int mtab[r];
    cdef int ntab[r];
    """

    """
    cdef int *itab = <int *>malloc(r);
    cdef int *ltab = <int *>malloc(r);
    cdef int *mtab = <int *>malloc(r);
    cdef int *ntab = <int *>malloc(r);
    """


    """
    cdef array.array[int] itab = array.array('i', 0)
    cdef array.array[int] ltab = array.array('i', 0)
    cdef array.array[int] mtab = array.array('i', 0)
    cdef array.array[int] ntab = array.array('i', 0)
    """
# http://cython.readthedocs.io/en/latest/src/userguide/external_C_code.html
# https://github.com/cython/cython/wiki/FAQ
# http://telliott99.blogspot.co.nz/2010/12/cython-3-my-own-c-source-file.html

    """
    cdef r = 3*(d+1)*(d+2)*(d+3)/6;

    cdef int *itab
    global itab
    itab=alloc1int(r);

    cdef int *ltab
    global ltab
    ltab=alloc1int(r);

    cdef int *mtab
    global mtab
    mtab=alloc1int(r);

    cdef int *ntab
    global ntab
    ntab = alloc1int(r);
    """

    """
    C Version:
    cpdef int r = 3*(d+1)*(d+2)*(d+3)/6;

    itab=alloc1int(r);
    ltab=alloc1int(r);
    mtab=alloc1int(r);
    ntab=alloc1int(r);
    """
    cpdef int ir = 0;

    # k == 0
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 0 and m % 2 == 0
                                        and n % 2 == 0) or \
                       (i == 1 and l % 2 == 1 and m % 2 == 1
                                        and n % 2 == 0) or \
                       (i == 2 and l % 2 == 1 and m % 2 == 0
                                        and n % 2 == 1):
                        # tabs[ir] = [i,l,m,n]
                        itab[ir] = i;
                        ltab[ir] = l;
                        mtab[ir] = m;
                        ntab[ir] = n;
                        ir += 1;
                        irk[0] += 1;
    # k == 1
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 0 and m % 2 == 0
                                                   and n % 2 == 1)
                                                      or \
                       (i == 1 and l % 2 == 1 and m % 2 == 1
                                                   and n % 2 == 1)
                                                      or \
                       (i == 2 and l % 2 == 1 and m % 2 == 0
                                                   and n % 2 == 0):
                        # tabs[ir][:] = [i,l,m,n]
                        itab[ir] = i;
                        ltab[ir] = l;
                        mtab[ir] = m;
                        ntab[ir] = n;
                        ir += 1;
                        irk[1] += 1;
    # k == 2
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 0
                        and m % 2 == 1 and n % 2 == 0) or \
                       (i == 1 and l % 2 == 1
                        and m % 2 == 0 and n % 2 == 0) or \
                       (i == 2 and l % 2 == 1
                        and m % 2 == 1 and n % 2 == 1):
                        # tabs[ir][:] = [i,l,m,n]
                        itab[ir]=i;
                        ltab[ir]=l;
                        mtab[ir]=m;
                        ntab[ir]=n;
                        ir += 1;
                        irk[2] += 1;
    # k == 3
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 0
                        and m % 2 == 1 and n % 2 == 1) or \
                       (i == 1 and l % 2 == 1
                        and m % 2 == 0 and n % 2 == 1) or \
                       (i == 2 and l % 2 == 1
                        and m % 2 == 1 and n % 2 == 0):
                        # tabs[ir][:] = [i,l,m,n]
                        itab[ir]=i;
                        ltab[ir]=l;
                        mtab[ir]=m;
                        ntab[ir]=n;
                        ir += 1;
                        irk[3] += 1;
    # k == 4
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 1
                        and m % 2 == 0 and n % 2 == 0) or \
                       (i == 1 and l % 2 == 0
                        and m % 2 == 1 and n % 2 == 0) or \
                       (i == 2 and l % 2 == 0
                        and m % 2 == 0 and n % 2 == 1):
                        # tabs[ir][:] = [i,l,m,n]
                        itab[ir]=i;
                        ltab[ir]=l;
                        mtab[ir]=m;
                        ntab[ir]=n;
                        ir += 1;
                        irk[4] += 1;
    # k == 5
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 1
                        and m % 2 == 0 and n % 2 == 1) or \
                       (i == 1 and l % 2 == 0
                        and m % 2 == 1 and n % 2 == 1) or \
                       (i == 2 and l % 2 == 0
                        and m % 2 == 0 and n % 2 == 0):
                        # tabs[ir][:] = [i,l,m,n]
                        itab[ir]=i;
                        ltab[ir]=l;
                        mtab[ir]=m;
                        ntab[ir]=n;
                        ir += 1;
                        irk[5] += 1;
    # k == 6
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 1
                        and m % 2 == 1 and n % 2 == 0) or \
                       (i == 1 and l % 2 == 0
                        and m % 2 == 0 and n % 2 == 0) or \
                       (i == 2 and l % 2 == 0
                        and m % 2 == 1 and n % 2 == 1):
```

```
                        # tabs[ir][:] = [i,l,m,n]
                        itab[ir]=i;
                        ltab[ir]=l;
                        mtab[ir]=m;
                        ntab[ir]=n;
                        ir += 1;
                        irk[6] += 1;
    # k == 7
    for i from 0 <= i < 3:
        for l from 0 <= l < (d + 1):
            for m from 0 <= m < (d - l + 1):
                for n from 0 <= n < (d - l - m + 1):
                    if (i == 0 and l % 2 == 1
                        and m % 2 == 1 and n % 2 == 1) or \
                       (i == 1 and l % 2 == 0
                        and m % 2 == 0 and n % 2 == 1) or \
                       (i == 2 and l % 2 == 0
                        and m % 2 == 1 and n % 2 == 0):
                        # tabs[ir][:] = [i,l,m,n]
                        itab[ir]=i;
                        ltab[ir]=l;
                        mtab[ir]=m;
                        ntab[ir]=n;
                        ir += 1;
                        irk[7] += 1;

    print("irk[0]=" + str(irk[0]))
    print("irk[1]=" + str(irk[1]))
    print("irk[2]=" + str(irk[2]))
    print("irk[3]=" + str(irk[3]))
    print("irk[4]=" + str(irk[4]))
    print("irk[5]=" + str(irk[5]))
    print("irk[6]=" + str(irk[6]))
    print("irk[7]=" + str(irk[7]))

    return *itab, *ltab, *mtab, *ntab, *irk
    # ERROR: rus_tools.pyx:1254:33: Cannot convert
    'int *' to Python object if I use alloc
```