# Software modeling of the propagation of electromagnetic beams through different media

Authour: Kjetil Rørvik Bruket
Supervisor: Dhayalan Velauthapillai

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2014


# Acknowledgements

I would first like to thank my supervisor, Dhayalan Velauthapillai, for the patience he has shown towards me, and his guidance and contribution that has made this work possible.
A special thanks to Bjørn Spjelkavik, Even Melum, and any others whose original Fortran functions were used in my solution.
I would also like to thank my family for the support and understanding they have given me during the course of this project.

# List of acronyms

**GCC** GNU Compiler Collection
**HIB** University College of Bergen/Høgskolen i Bergen
**IDE** Integrated Development Environment
**JVM** Java Virtual Machine
**TM** Transverse-magnetic
**TE** Transverse-electric
**UIB** University of Bergen/Universitetet i Bergen

# List of figures

# Table of contents

# Chapter 1 Introduction

## 1.1 Motivation

Propagation and focusing of electromagnetic beams through layered anisotropic medium is of interest in the field of optical data storage, where thin layers are mounted on glass substrates, display technology, where polarised light passes through liquid crystals, and in biological and material sciences, where objects are portrayed through thin sheet glass. In the case of computing a focused or diffracted field of a three-dimensional optical problem, it would be required to evaluate two-dimensional integrals with amplitude and phase functions. Obtaining numerical results for these kinds of problems, is a difficult task because of the rapidly oscillating integrands and the singularities in the Fresnel transmission and the reflection coefficients. If a layered medium is an anisotropic crystal, the numerical analysis is complicated significantly, as it gives rise to birefringence and mode coupling.

Research within this field is jointly being conducted by the Department of Engineering at the University College of Bergen [HIB] and the Department of Physics and Technology at the University of Bergen [UIB]. Their researchers have obtained both exact and asymptotic results for propagated and focused fields in uniaxial crystals, and they have been adopting various techniques to obtain numerical results. The double integrals, which can be used for obtaining the results, can be reduced to single integrals by means of applying parabolic approximations to the phase and amplitude functions, and even though the approximations tend to give numerical results, the procedure is time-consuming. As such obtaining numerical results from this sort of procedure might take a few minutes to several computing hours. To date, they have typically written software in Fortran, in order to achieve the numerical results, and this intermediary data is then used as input data for the MATLAB software, in order to present the final results.
This process of having to code one program for producing results that are passed to another, just to able to produce the final results, is convoluted, the use of different programs in series for producing the final result creating unnecessary work for the user. As such, the researchers have requested a singular software solution that will handle both the numerical calculations and the graphical presentation of the propagated electromagnetic fields from the initial data, which describes the characteristics of the medium and the type of electromagnetic wave.

## 1.2 Thesis overview

Chapter 2 describes the theoretical background for the solution, as well as introducing the problem that this solution is trying to solve. Chapter 3 analyses the problems related to the thesis, and describes the development methodology, and tools used for developing the application. Chapter 4 discusses the design of the application and how it was implemented. Chapter 5 evaluates the thesis and the solution. Chapter 6 concludes the thesis and provides suggestions for further improvements to the solution. The appendices provide explanations for each of the dialog windows and instructions for compiling the application.

# Chapter 2 Background

## 2.1 Transmission of Gaussian beams into uniaxial crystals

The researchers at UIB and HIB work with theories for the transmission of Gaussian beams into uniaxial crystals, which are a type of anisotropic material. The studies involving the propagation of Gaussian beams mainly focus on the propagation in isotropic medium, based on scalar theory and paraxial optics. The results of these studies may in some cases be applied for uniaxial crystals, as when a transmitted Gaussian beam is applied to the surface of a uniaxial crystal, the beam will be split into an ordinary beam, and an extraordinary beam, with different propagation directions at the interface [GAU1][GAU2], a property called birefringence. The equations for these are quite complex, consider solving for the exact solution of the transmitted electric field $E^t(r,t)$ at a three-dimensional point $r$ of a uniaxial crystal, where $E^{ot}$ is the ordinary electric field, and $E^{et}$ the extraordinary electric field:

$$E^t(r,t)=\Re\{[E^{ot}(r)+E^{et}(r)]\mathrm{e}^{-i\omega t}\}$$

The solution for this requires solving multiple double integrals which would take considerable time even on a computer.[GAU2]

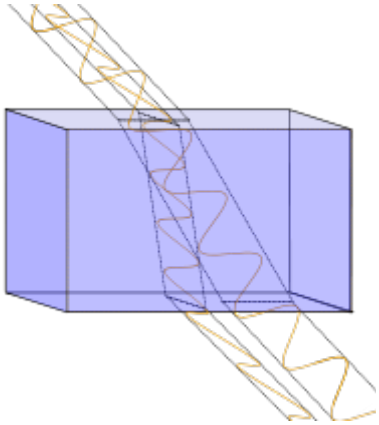

Figure 1: Rays passing through birefringent material, Wikipedia, Birefringence, by Mikael Häggström

One case of using the results of studies that focus on propagation in isotropic medium, is when the plane of incidence of a linearly polarised Gaussian beam is along its electric field [TM field], as opposed to when the plane of incidence is normal to the electric field [TE field]. When it is a 2D TE field, the problem becomes a scalar problem, when it is a 2D TM field, the problem becomes a vectorial problem, in which case, the electric field can be expressed in a form similar to that for scalar Gaussian beam propagating in an isotropic medium [GAU1]:

$$E^{et}(x,z)<\int_{-a}^{a} G(x')\mathrm{e}^{iH(x')}dx' \quad \text{[Equation 1]}$$

where

$$G(x')=E_0 f_E^{e,x}\frac{[k_x^s(x')]}{\{2\pi h^{en}[k_x^s(x')]\}^{\frac{1}{2}}}\exp(-x'^2\frac{}{2}\sigma_0^2) \quad ,$$

$$H(x')=h^e[k_x^s(x')]+\frac{\pi}{4}sgn\,h^{en}(k_x^s)$$

Also, when one assumes that the aperture size $a$ is larger than the beam waist $\sigma$ , one may use also use the paraxial approximation to express an approximated equation in even simpler terms [GAU1]:

$$E^t(x,z)=E_0 f_E^x(k_x=0)[\frac{\sigma_0}{i\sigma(\tilde{Z})}]$$
$$\times\exp\{-\frac{[x-\delta(z-z_0)]^2}{2\sigma^2(\tilde{Z})}\}$$
$$\times\exp(ik_1\{Z+\frac{[x-\delta(z-z_0)]^2}{2R(\tilde{Z})}\})$$
$$\times\exp[i\frac{1}{2}\arctan(k_1\sigma_0^2/\tilde{Z})] \quad \text{[Equation 2]}$$

## 2.2 Problem statement

As the researchers at UIB and HIB derive new equations for making an approximation of the exact results of an original equation, they will also develop programs according to their theories for calculating numerical results to compare these with the exact results.These programs have been written in Fortran, and their complexity varies from operations performing single for-loops, to several nested for-loops.

The Fortran programs that have been written up to date, are very static in their function. They will output one or several files, this depends on the program the user would run, these files containing two columns of numbers that would be visualised on a 2D graph plot, the two numbers of each row representing the coordinate point. The numbers that are calculated by the programs are dependent on internal variables, but once a program has been compiled, the user has no way to alter these himself. This means that if the user wants to alter the behaviour of the program, by altering the variables that are used by the program, to set up a different scenario, he would have to go into the program's code, alter these here, and compile and run the program over again.

The software they have been using for the graph plot is the MATLAB software, so after a Fortran program has executed, the outputted files would be loaded into MATLAB in order to visualise the results. If the user wants to prepare for a different scenario, he would repeat the above steps, which, in the case of a Fortran program running a computational-heavy task, would require waiting a long time until the program has executed. This is the case with some of the programs, and waiting around to manually load the outputted files into MATLAB would be trying on one's patience.

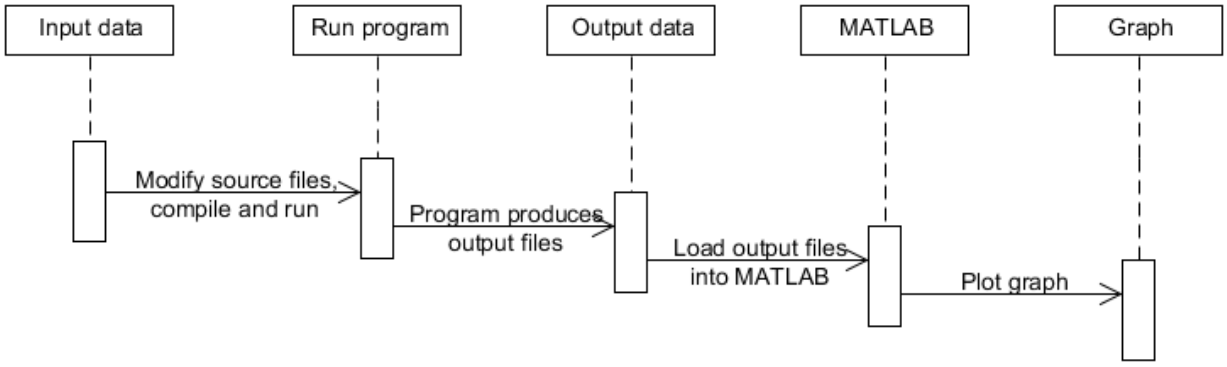


Figure 2: Workflow in the current situation

# Chapter 3 Problem analysis

For every time the user wants to try out a new scenario, instead of interacting between a program's source code, running the program and loading the data into MATLAB, it would have been easier for the user to simply input the values into a program, and have this program display the results of calculations once they were done in the form of a graph plot. So, a graphical user interface that would accept the inputted values, and simulate, replicate, or call the functions of the original Fortran programs, to then draw the graph plot, was needed.

One can then divide the problem in these parts:
- Graphical user interface
- Linking with Fortran systems
- Graph plotting

## 3.1 Graphical user interface

As the application would be run on a GNU/Linux desktop environment, one would have to choose a graphical library that is compatible with the desktop environments that are used by the different GNU/Linux distributions, the most prominent ones being KDE and GNOME. For this there are many graphical libraries that are compatible with most desktop environments, such as GTK+, wxWidgets, and Qt. Due to my former experience with Qt, not being familiar with other graphical libraries for GNU/Linux, and with Qt being well-documented[QT], the choice fell quickly on Qt, making the main implementation language C++. Additionally, Qt does not only provide support for graphical development, but also has several neat additional functions that are not restricted to graphical development. The version of Qt that was first considered was 5.0, but where as I had formerly used Qt 4.8, and 5.0 seemingly being different from 4.8 and having less documentation than the latter version, I settled on Qt 4.8.

## 3.2 Linking with Fortran systems

Because these Fortran programs already existed, and their source code was provided to me by my supervisor, trying to simulate their function by coding the logic all over again, would be unnecessary. Instead, I looked to integrate the Fortran programs into the application, by taking advantage of the existing source code.

Three different strategies in regards to the existing Fortran programs were considered, letting the application edit the Fortran source files in regards to the input data, compile and run, have the application run the programs and input data by commands in the background, or extract the functionality of each program into their own functions, calling these functions from the application.

The strategy of letting the application edit and compile the source files was considered a bit reckless. Entering incorrect values as input could have made the program uncompilable, a scenario that would be hard to handle, and there would also be the possibility of injecting malignant code. In addition, the application would have to wait for the compiler to finish, then run the program and wait for that to finish, before the results could be displayed. There was also the issue of where the source files would be located, whether they should be part of the internal resources of the application, to be saved to the file system after having been edited, or whether they were on the file

system, with defined blocks to be edited.

For the strategy of having the application run the programs and input commands, one would have to modify the programs to accept input from the standard input device, but one would also have to define for the application what kind of input the program is expecting. A configuration file for each program was considered for this task, which would have let the user select the configuration file and then be prompted with input fields for each input command. The concerns about this solution was whether if the file defined in the configuration really existed in the file system, whether it was an executable, and whether the correct amount of input fields had been defined, the program being paused indefinitely because it would still await input.

What was ultimately chosen was to extract the functionality of the programs into functions, which would be called from the application. This wouldn't mean that every function would have to be implemented in the main implementation language, but that we could write the interface in Fortran and still be able to call Fortran functions from C++. There are different approaches to this, most famously dynamic linking, in which code is compiled into non-executable machine code, but loaded together with the executable when you run the executable file, allowing one to write references to functions that you won't find within the scope of your code. Another way is by use of language binding, which functions similar to dynamic linking, except the code written in a secondary programming language may be compiled together with the main implementation language. As I had no experience in dynamic linking, I tried to give language binding a shot.

## 3.3 Graph plotting

The Qt libraries provide no native way for displaying graph plots in a Cartesian plane. This may be explained by developers of Qt applications wanting their own design for displaying plots, or graph plots having different methods for being displayed, that the developers of Qt have chosen to not decide on a standard method. However, Qt has a powerful graphics library, so it doesn't prevent developers from writing their own tools for displaying graphics the way they want to. For someone to generate a graph plot for the use in a Qt application, they would have to write the functionality themselves, rely on a third-party library or even an external program. For Qt, some third-party libraries that can be used to display graph plots are called Qwt and QCustomPlot, and external programs that could be used would be gnuplot and MATLAB.

The plots that are required for this solution didn't seem so advanced for me, and I was reluctant on using third-party libraries or external programs, as these would rely on being installed onto the user's working environment, so I decided to write the functionality myself.

## 3.4 Implementation language

A minor argument towards choosing the implementation language, was that the application could also be cross-platform, not being restricted for use on the GNU/Linux platform. A sounding response to that would be to use Java, as Java provides an extensive library and any code written in Java is cross-platform, unless directly accessing functions from the operating system. However, most programming languages are cross-platform, how an application is written, and how it is compiled, determines whether or not it is cross-platform. Java applications traditionally relies on the Java Virtual Machine [JVM], which provides automatic garbage collection by detecting objects

that have been discarded and purges them from memory, but JVM also makes compatibility with other programming languages awkward, as there might not be support for compiling code written in other programming languages into bytecode which is what JVM uses internally to produce machine code.

On its own, C++ is cross-platform, having been standardised by the International Standardization Organization, and so the C++ Standard Libraries are available across any C++ compiler that conforms to the ISO standard and function the same. The problems with C++ comes across when it comes to memory management, GUI development, and accessing functions of the operating system. The two latter problems often come hand in hand, as it would be the operating system that is in charge of drawing the GUI elements, and for the memory management, any object that is instantiated, meaning that its memory is allocated to the heap, should also be destroyed. The Qt framework solves the problems of GUI development and also accessing functions of the operating system. This does not imply that Qt has a solution for every such function, but it does for the more important ones, such as getting the current time, or creating new threads. It has no solution towards the memory management problem, but how it solves the two other problem is a strong argument for choosing C++ for the main implementation language. Another argument for choosing C++, is the possibility of accessing external entities that don't come via include directives or are defined otherwise in the code.

Because the application also need to access Fortran functions, this also means some work needs to be done writing interfaces in Fortran, making Fortran into a secondary implementation language.

## 3.5 Development methodology

It is hard to define the methodology I used during this project. I tried to keep in mind the practices of Agile software development, though the Agile methods are not entirely suited to a single-man team, where Agile methods promote cooperation with other team members. This does not mean that you can't derive any valuable lessons from the Agile methods, and you're not meant to follow a specific Agile method obediently either, but pick and choose between the practices that benefits your team the best[ASD], leaving me free to adapt to a lone developer working method. Agile values collaboration with the customer, and this means that the customer, or a representative for the customer, is part of the development process, being presented with working software at regular intervals and giving feedback based on this. In the scope of this project, my supervisor was the customer and should have had frequent updates on the progress.

Concerning design practices, Agile values simple designs, requirements only changing according to needs. Keeping a simple design meant that instead of running a program by the application, after having it read the proper configuration, the application could just call the functions that had already been coded.

Something the Agile methodologies also value, is Test-Driven Development. Under this practice, any code to be written has to be preceded by a test that confirms its validity. If the test does not pass, one has to assume that either the code has not been written or it does not conform to the test's parameters, and if the test passes before the code has been written, then there's something wrong with the test. However, writing tests may not be so intuitive when you're developing user

interfaces. For code that is easily testable, there shouldn't be any problem writing tests, although I opted out of writing unit tests as I'm not familiar with any testing frameworks for C++.

## 3.6 Development environment and tools

Because the main use for the application would be on GNU/Linux, the operating system the application was developed and tested on was Ubuntu 12.04, running on VMware Player 6.0.1. Choosing to run the operating system on a VM is not related to the project, but sharing this information may help understand the constraints of the work.

The Integrated Development Environment [IDE] used for developing the application was the NetBeans IDE 8.0 for C/C++. This IDE also provides tools for helping develop a Qt application, and although the Qt Creator IDE could have been used, which is designed entirely for developing Qt applications, I felt NetBeans was more robust, where NetBeans also has support for syntax highlighting in Fortran source files.

The compilers used were from the GNU Compiler Collection [GCC], the IDE being set up to use the g++ compiler for both C++ files and Fortran files. In addition, qmake was used for setting up makefiles for the application, dependent on the Qt project files. qmake is also in charge of converting Qt-exclusive syntaxes which are not part of the C++ syntax, such as signals and slots, into proper C++ code. Both of these actions happen automatically when targeting the Qt project file.

# Chapter 4 Implementation

## 4.1 Overview

The application consists of a control window defined by MainWindow. A central widget to MainWindow is GraphPlotter, and is used to plot graphs from data that arrives from the threads. The dialogs are activated by menu options from MainWindow, and receive input which is sent to the threads. The threads interact with the Fortran functions.

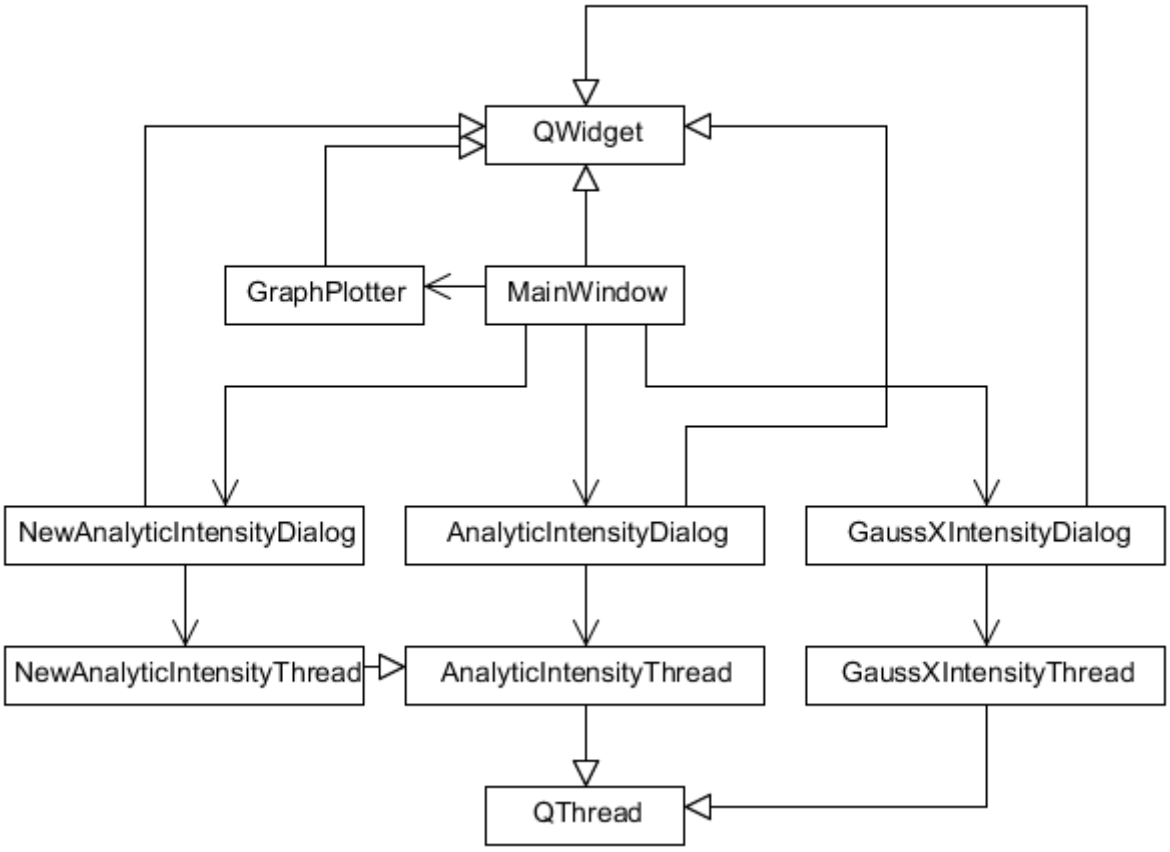


Figure 3: The class structure of the solution

## 4.2 MainWindow

This class inherits from QWidget, its purpose is to function as the controller of the main window of the application, from which the other UI elements are controlled. Its UI elements consists of a menu bar (QMenuBar), the visual representation of GraphPlotter, text boxes to change the properties of GraphPlotter, and a visual representation of QTableWidget, which displays the actual values of the graph being visualised in GraphPlotter.

MainWindow's menu bar has two menus (QMenu), "File" and "Routines", each menu containing a number of actions (QAction). File contains two actions, "Save as image..." and "Exit". The "Save as image…"-action is connected to a function that will cause GraphPlotter to open a file dialog to

accept a filename and directory for an image file, the “Exit”-action is connected to a function that will force the window to close, which will stop the application. “Routines” contains three actions, “2D TM Gaussian beam”, “2D TM Gaussian beam (simplified)”, and “2D TM Gaussian beam (non-truncated, chi << 1)”, which will each open their own dialog window.

GraphPlotter occupies the majority of the space of MainWindow’s window, and on its right-hand side one will find text boxes to modify the properties of GraphPlotter, which are the focus and scales GraphPlotter uses to draw the graph.

When the threads belonging to the dialog window for each routine have finished processing, MainWindow will receive QVectors from these which it will pass on to GraphPlotter, forcing GraphPlotter to draw its contents, and will also fill out the QTableWidget with the values returned.

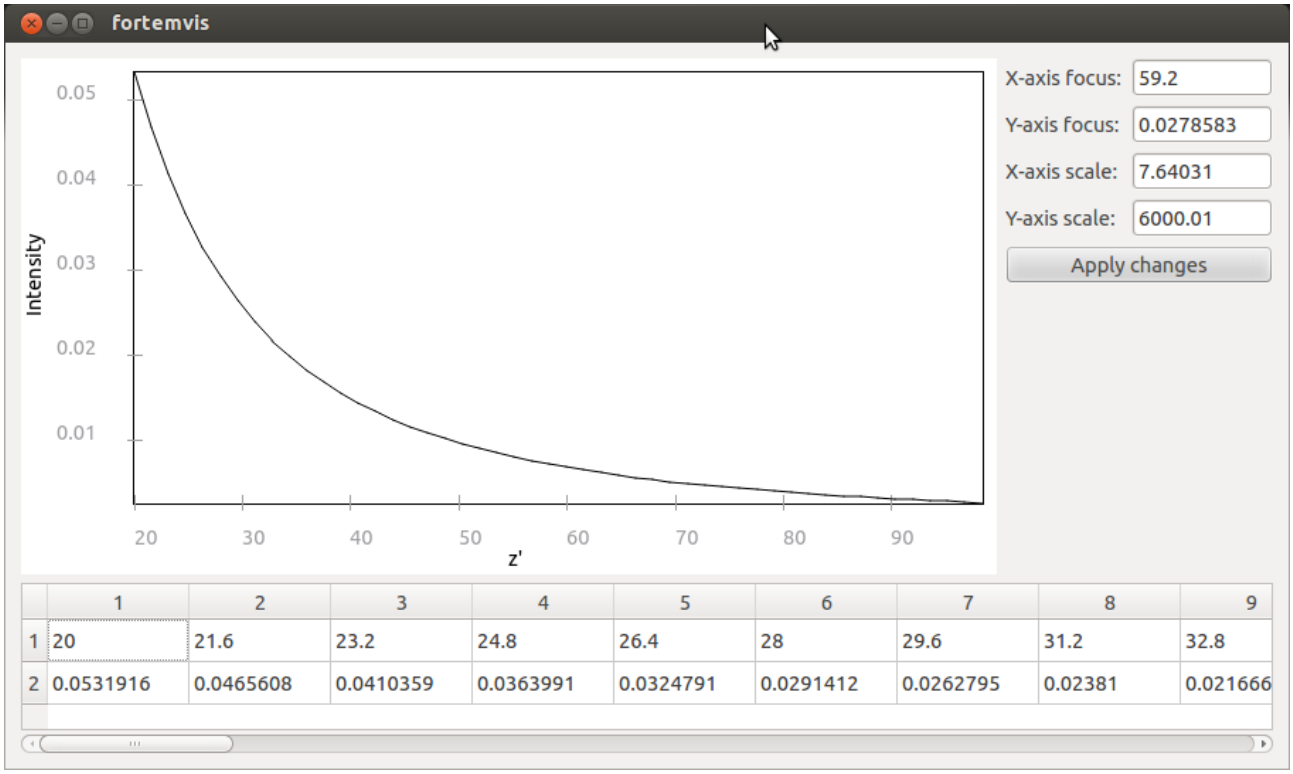


Figure 4: Screenshot of MainWindow’s dialog window

## 4.3 GraphPlotter

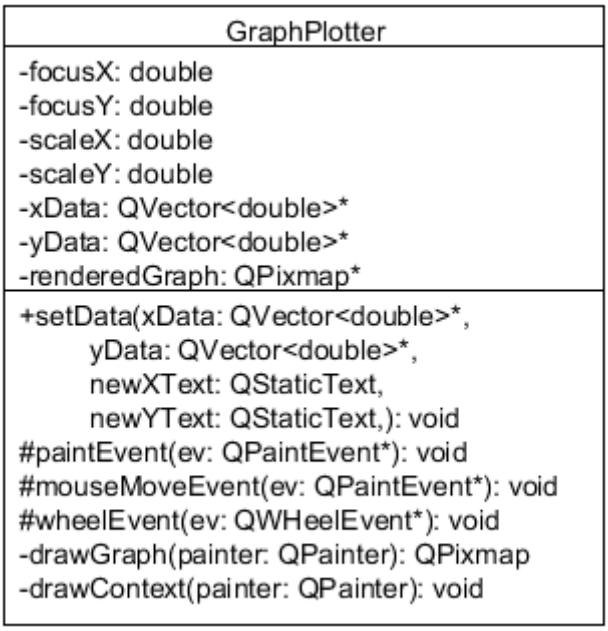


Figure 5: Class diagram for GraphPlotter

This class inherits from QWidget, its purpose is to draw the graph from the values returned from the threads belonging to the dialog windows. The draw context of the graph is determined by two QVectors, one for the x-coordinates and one for the y-coordinates, a focus, which determines the real point which should be in the exact middle of the draw context, and two scales, which determines the pixel-distance between each x-coordinate and y-coordinate respectively.

The QVectors of GraphPlotter are set by the public function setData(). setData() saves the QVectors to itself, and goes through each vector to find the minimum and maximum values of both vectors, which it will use to determine the focus of the graph and an appropriate scale. The focus is determined by the addition of the half-distance between the maximum and minimum of the graph with the minimum for both vectors (eg. $focus_y = min_y + \frac{max_y - min_y}{2}$ ), placing the focus in the absolute middle of the graph. The scales are determined by evaluating the width and heights of the draw context against the distance between the maximum and minimum x- and y-coordinates of the graph(eg. $scale_y = \frac{height}{max_y} - min_y$ ). This ensures that the entire graph is covered by the draw context when it is being drawn. setData() ends by forcing GraphPlotter to redraw its context. setData() also accepts context text for the axes as QStaticText , which GraphPlotter

GraphPlotter draws its context by overriding QWidget's paintEvent(), which calls drawContext(). drawContext() creates a new object of the type QPixmap, on which it will draw the graph and its context, which paintEvent() will use to draw upon the widget. This QPixmap-object is also used when saving an image from MainWindow.

drawContext() calls drawGraph(), which creates its own QPixmap-object. This QPixmap-object is

needed to prevent drawing lines onto GraphPlotter's own QPixmap-object, having the lines exit the context of the graph. Another thing that deemed the QPixmap-object necessary, is Qt's native painting engine, which can interpolate the drawing basis of a drawing object to proper pixel points, only ignoring drawing objects that fall entirely outside the context. This mean that if you were to tell Qt to draw a line between two points that fall outside the draw context, but they still intersect the context's rectangle, it will still draw a line.
drawGraph() continues by going through each pair point in GraphPlotter's vectors, to convert the point values into a pixel point and draw lines from one point to the other, based on the focus, the scales for the coordinates, and the draw context's height and width (eg. $pixel_x = scale_x(entry_x - focus_x) + \frac{width}{2}$ ).

After drawGraph() is finished, drawContext() continues by drawing visual information around the graph, which are tick marks accompanied by numbers, to represent exactly where a visual graph point is in relation to the data. Setting an arbitrary, constant value to distance the tick marks based on the real coordinates of a pixel point, would cause the tick marks and their numbers to not be drawn when the scale is too big, and pack them too tightly together when the scale is too small. And on the other hand, setting a constant value based on the number of pixels in a direction, would cause the rendered numbers to have subjectively confusing forms, eg. 0,5496, 0,8371 could have been rendered next to eachother.

To remedy this, drawContext() will need to find a number by which it can space the tick marks out without them not being rendered, or being packed too close. This can be solved by using an order of magnitude, and drawing the ticks based on the real coordinates of a pixel point, which will then evenly space out the tick marks according to the real coordinates. As the scales determine the distances in the rendered graph, it would be proper to use these to determine the order of magnitude, however as the scales determine the pixel distance between each unit, and we want a distance between each pixel, the order of magnitude would be based on the inverse of the scales, giving us a unit distance between each pixel.

As viewing values in the power of ten is more intuitive than a series of numbers distanced by a seemingly random number, the base 10 logarithm is used on the inverse of the scales to determine an exponent, and this exponent is rounded down to find an order of magnitude for the power of 10, $x = \lfloor \log_{10}(\frac{1}{scale_x}) \rfloor$ , $y = 10^x$ . The distance $y$ determines is still too small, a tick mark drawn for every real $y$ , where one pixel would contain many $y$ , but this is just simply solved by multiplying $y$ by 100.

To determine where drawContext() should start drawing tickmarks, and where it should end, the boundaries of the draw context are converted to real values, and then rounded, based on the order of magnitude, into the nearest number that fits within the draw context, and drawContext() starts drawing tick marks based on these rounded numbers, starting from one rounded boundary and ending at the other. Because the tick marks might still get too closely packed together, when the spacing between the pixel is below a certain threshold, $y$ is multiplied by 500, giving a power of 5 instead.

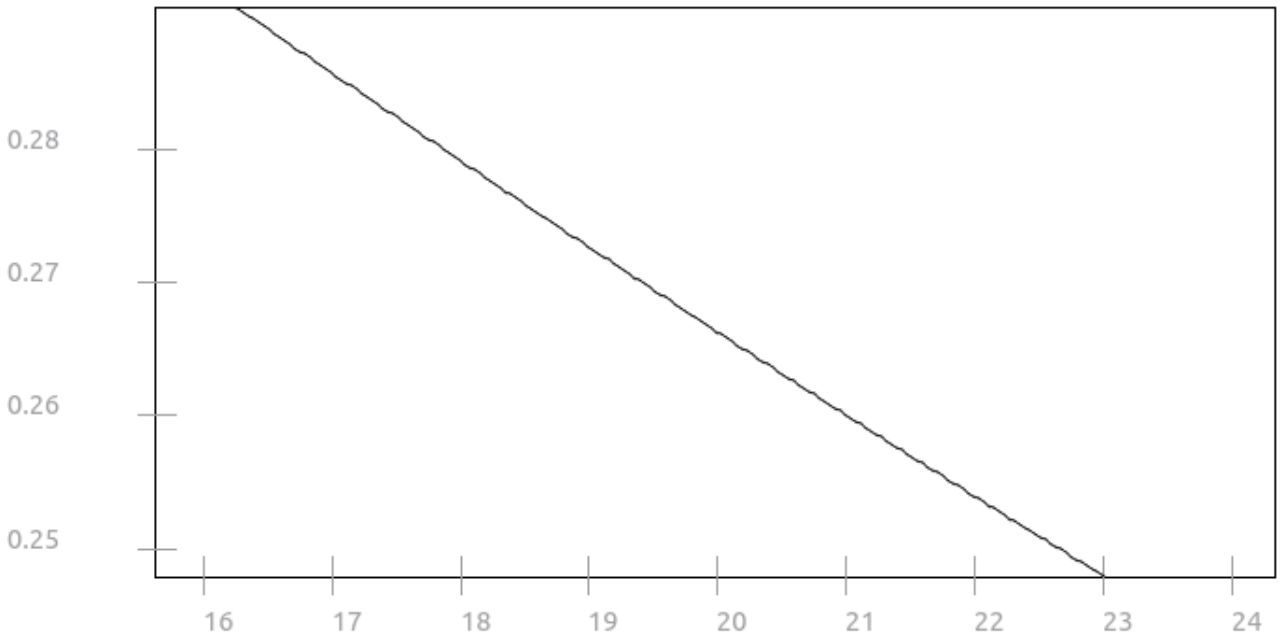


Figure 6: Power of 10 rule in effect

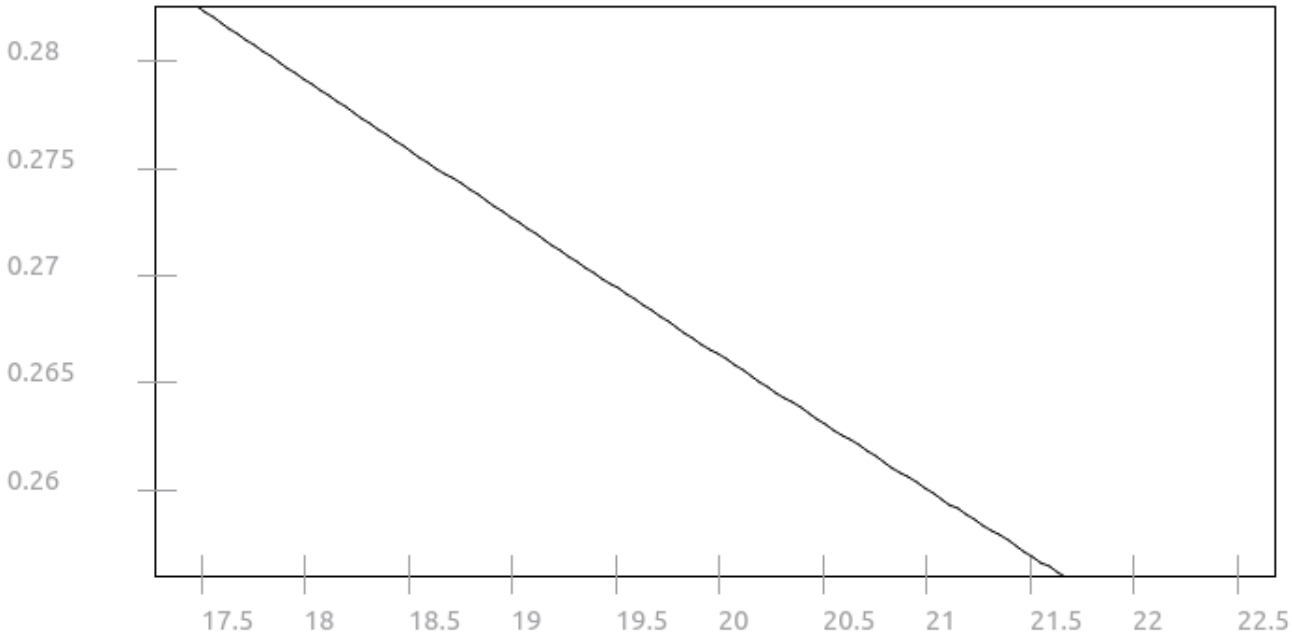


Figure 7: Power of 5 rule in effect

After having drawn the graph and the visual information for the graph, drawContext() will then return to GraphPlotter's paintEvent(), which will draw the QPixmap-object on the entire widget.

In the context of visualised graphs, one would expect y-coordinates to increase upwards. Qt, along with many other graphical drawing systems, places the pixel point (0,0) in the top-left corner of its context, and increases y-pixel coordinates downwards. For all operations working on the y-coordinates in GraphPlotter, this means it has to treat the y-pixel coordinates opposite of how it treats x-pixel coordinates, so where x-pixel coordinates would be increasing along with x-

coordinates, y-pixel coordinates decrease when y-coordinates increase. Consider,

$pixel_x = scale_x(entry_x - focus_x) + \frac{width}{2}$ , which for y-coordinates would be

$pixel_y = scale_y(focus_y - entry_y) + \frac{height}{2}$ .

GraphPlotter supports zooming and moving the graph around with the mouse.
When the mouse-wheel, or a similar input device, is scrolled inwards or outwards while the mouse pointer is over the widget, the scales will either increase by a multitude of 1.1, or decrease by a multitude of 1.1. This is handled by wheelEvent().
When the left mouse button is held down, the focus point changes based on which direction the mouse pointer is moving and the scales, conforming to mouse movement at any scale. This is handled by mouseMoveEvent().

Using GraphPlotter's saveAsImage() will open a file dialog to select a directory and a filename for an image file. The file may be saved as any of the writable image formats of Qt, which includes JPEG, PNG, and BMP, however the user will have to write the extension in themselves, with .jpg, .png, .bmp, or choose an existing image to overwrite. This seems to regrettably be a design flaw with Qt's QFileDialog, and a workaround to this might be more trouble than it's worth.

## 4.4 Dialog windows and threads

The dialog windows inherit from QWidget, and the threads from QThread. There exists a QDialog class in the Qt libraries, but because hitting the Escape button when the focus is on a QDialog window would force the window to close without a possibility to ignore this event, using QDialog was undesirable (as explained in 4.5.a).

The dialog windows interact with threads, which interact with the Fortran functions. The dialog windows are activated by selecting the relevant menu option of MainWindow, and are modal, meaning that unless you close them, you will be unable to interact with any other UI element of the application. In this solution, there are 3 dialog windows, each dialog window having their own thread, each based on the different Fortran programs I received from my supervisor:
-**GaussXIntensityDialog**
-**AnalyticIntensityDialog**
-**NewAnalyticIntensityDialog**

The purpose of each of these dialog windows is covered in Appendix A.

Each dialog window contains a number of textboxes for writing in input, a push button to press to start a thread, and a progress bar that will be visible during the thread's execution. All dialog windows except GaussXIntensityDialog also have a choice of what kind of result to be displayed in GraphPlotter and the QTableWidget of MainWindow.

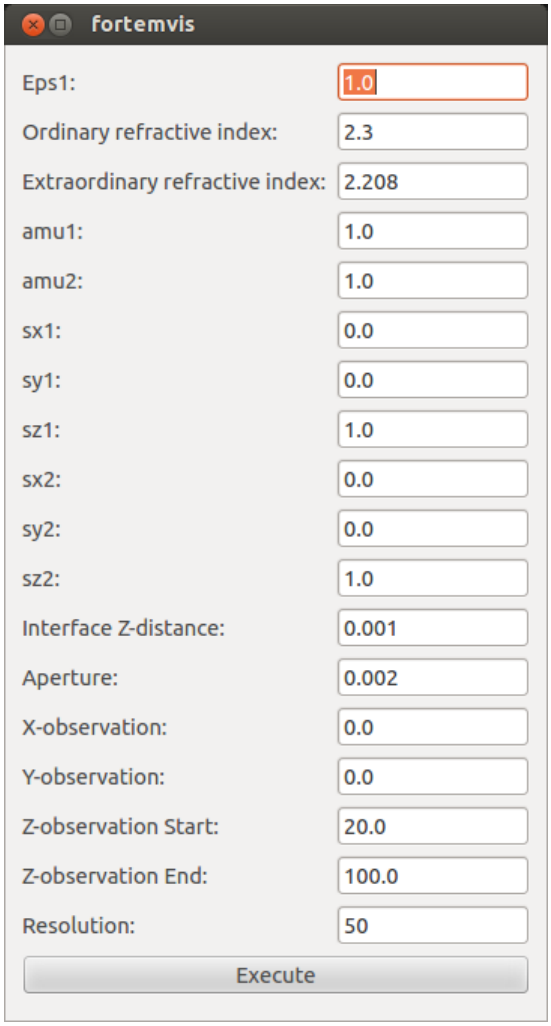


Figure 8: Screenshot from GaussXIntensity's dialog window

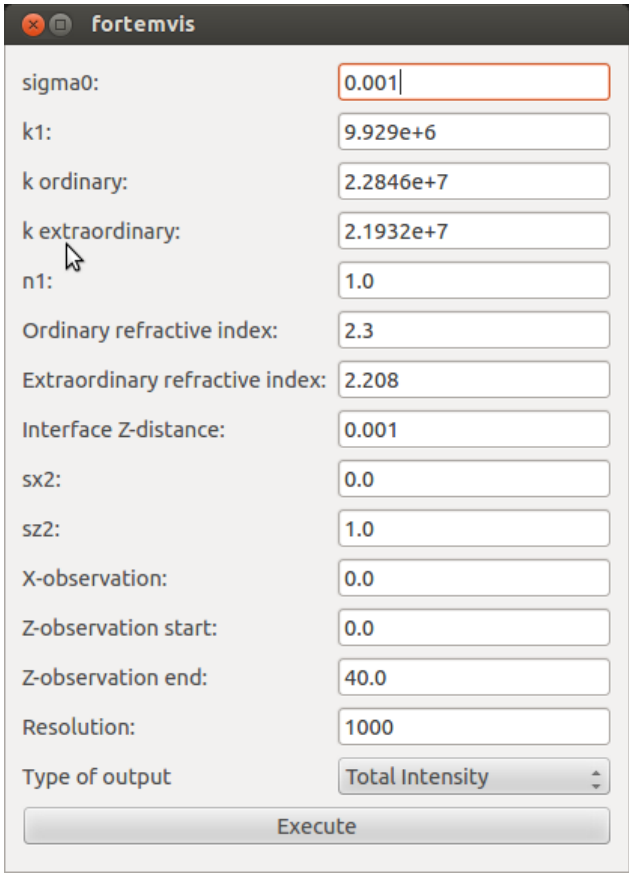

Figure 9: Screenshot from AnalyticGaussDialog's dialog window

When the dialog window's push button is activated, the input in the textboxes will be validated to check if erroneous values have not been inputted, and if there are none, the input is passed over to the thread belonging to the dialog window, and the thread will start executing. At this point the progress bar belonging to the dialog window will be visible, updating based on the thread's progress. The progress bar is important to let the user know that the program didn't freeze, and how many operations are left before he can view the results.

The reason why the threads are necessary to the application, is because normally every operation a Qt application performs is done on a single thread, in some cases called the UI thread. This includes all UI operations and all calculations, so if a series of operations are being performed for a relatively long time, any planned updates to the UI will have to wait until these operations are done. This will make the application unresponsive until it has performed the offending operations, and can give the user the impression that the application has frozen. As some of the operations the threads currently perform may prove to be rather time-consuming, especially when the resolution (number of distinct points) of the graph is increased, executing these operations on the UI thread would undoubtedly cause the UI to freeze up.

## 4.5 Interacting with Fortran functions

GCC, when the gfortran driver has also been installed, is able to compile Fortran source files together with C++ source files, but the g++-compiler will specifically not recognise certain native Fortran operations when linking object files together, such as write, pause, or print, aborting the

linking process if it encounters these operations.

In C/C++, one may define references to entities that will only be checked whether they exist at the time of linking the object code together, not when the source files are being compiled into object code, this is possible by the use of the extern-keyword. A particular case for C++, is defining the extern “C”-directive, to prevent what has received the negative moniker, name mangling. Many compilers use name mangling as a way to prevent name conflict when linking object code, as in C++, functions may share names (function overloading). Name mangling is undesirable if you want to access a specific entity that has not been processed as C++ code. Essentially, what the extern “C”-directive tells the compiler, is to treat the entities as C entities, because unlike C++, C does not support function overloading.

Something special that GCC does with names for functions or other entities under compilation, is to convert uppercase letters to lowercase, and add underscores to the name for the object code. Take for example the Fortran function ANALYTIC(), which would become analytic_() in the object code. This means one cannot just directly refer to an entity with the name you’d encounter in Fortran from C/C++, but you would type in the name as how the compiler would compile the name for object code.

The GCC compiler will resolve different data types used by Fortran and C/C++ into the same data type in the object code. For example, Fortran’s DOUBLE PRECISION and REAL*8 are the same as C/C++’s double. However, Fortran may also use complex numbers, defined by COMPLEX, but C itself does not support for complex data types, and to use complex data types in C++ requires including one of the standard libraries. Also, where Fortran would define an array by DOUBLE PRECISION array(54,3), C/C++ would define an array using double array[3][54].

In the standard format for writing Fortran code, you are required to include a margin before any operation, and every line is limited to a certain number of characters. This means that a function name, together with the names of its parameters, may be very limited, making it difficult to pass multiple parameters to a function. One solution to this, is using the Fortran COMMON-keyword. This keyword defines a static container, which can hold a number of objects, and all Fortran functions in a program can access the container and its contents by using the same definition. In C, a similar object would be struct, although there’s no need to define struct in every function that wants to use it. For the GCC linker, struct objects and the COMMON-container may be the same, so by defining a struct object with the same name, same size, and the same type of variables and variable names, as what a Fortran function would use for a COMMON-container, a C++ program would be able to access the container, eg. COMMON/EPSBLOCK/eps1,eps1o,eps1e, where eps1, eps1o, and eps1e in a function would defined as DOUBLE PRECISION, would be extern struct { double eps1, eps1o, eps1e; } epsblock_ ;.

Fortran supports passing functions as parameters to a different function, similar to C++, but in the case of Fortran, this is much simpler. C++ would need to convert the function to a pointer, defining the parameters it should accept and its return type, in Fortran it just needs to pass the name of the function. In fact, Fortran treats all function parameters as pointers, meaning that an original variable that is passed to another function, would change if this function modified the parameter variable.

Because of uncertainty of how native Fortran functions operate on numbers compared to C++'s standard library functions, the complex data type, and passing functions as parameters to functions, interaction functions that use the various Fortran functions were made in Fortran to simulate the behaviour of each program, simply returning non-complex numbers. These interaction functions are called from the threads.

The Fortran programs the logic of the Fortran interaction functions were based on, had for-loops that start from one defined value, and end at another defined value, with a defined number of steps between the two values for the for-loop to iterate over. For every such step, output would be deposited into various files defined in the programs. In this solution, instead of depositing the output into files, the interaction functions accept parameters that is equivalent to the output of the original programs, and repeating the same logic as inside the original for-loops, calling any other Fortran functions that the original program called. Because the parameters are treated as pointers, when the function returns to the function that called it, the values for these have changed. Here the threads use a for-loop instead, storing the variables into array-structures under each iteration. There are also some interaction functions that simply help setting up any common variables that would be constant under the execution of the thread's for-loop.

### 4.5.a Sample behaviour of a thread

When a thread starts executing, it will modify the relevant Fortran COMMON-containers to the input it has received from the dialog windows. Then based on the resolution the user has defined for the outputted graph, the thread will step as many steps as defined by the resolution, from the defined z-observation start, to the defined z-observation end. For each step, the thread calls the interaction function it controls, passing variables for the interaction function's output. The modified variable and observed z-point are stored onto QVectors, the QVectors being passed out with an emitted Qt signal.

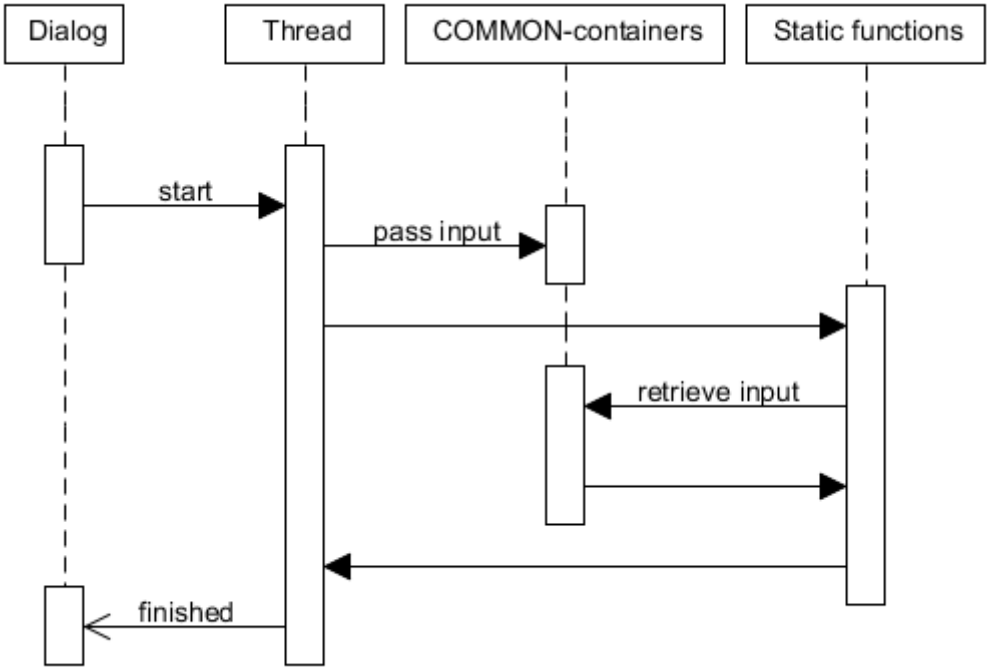


Figure 10: Sequence diagram for the interaction with Fortran functions

Because the COMMON-containers are static to the program, care has to be taken to not allow multiple threads to work on the data at the same time. Due to them being static, they are not thread-safe, and two threads modifying an entry at the same time would lead to data corruption. This means that any other threads has to be blocked from executing, the solution used by this application is by making the dialog windows a thread belongs to modal, and preventing the user from closing the dialog window while the thread is executing, to prevent the user from taking any action that would start another thread. The Fortran functions not being thread-safe is regrettable, as the program could have benefitted from several threads calling the same function with different parameters on a multi-core processor, reducing the time a user has to wait for the calculations to finish.

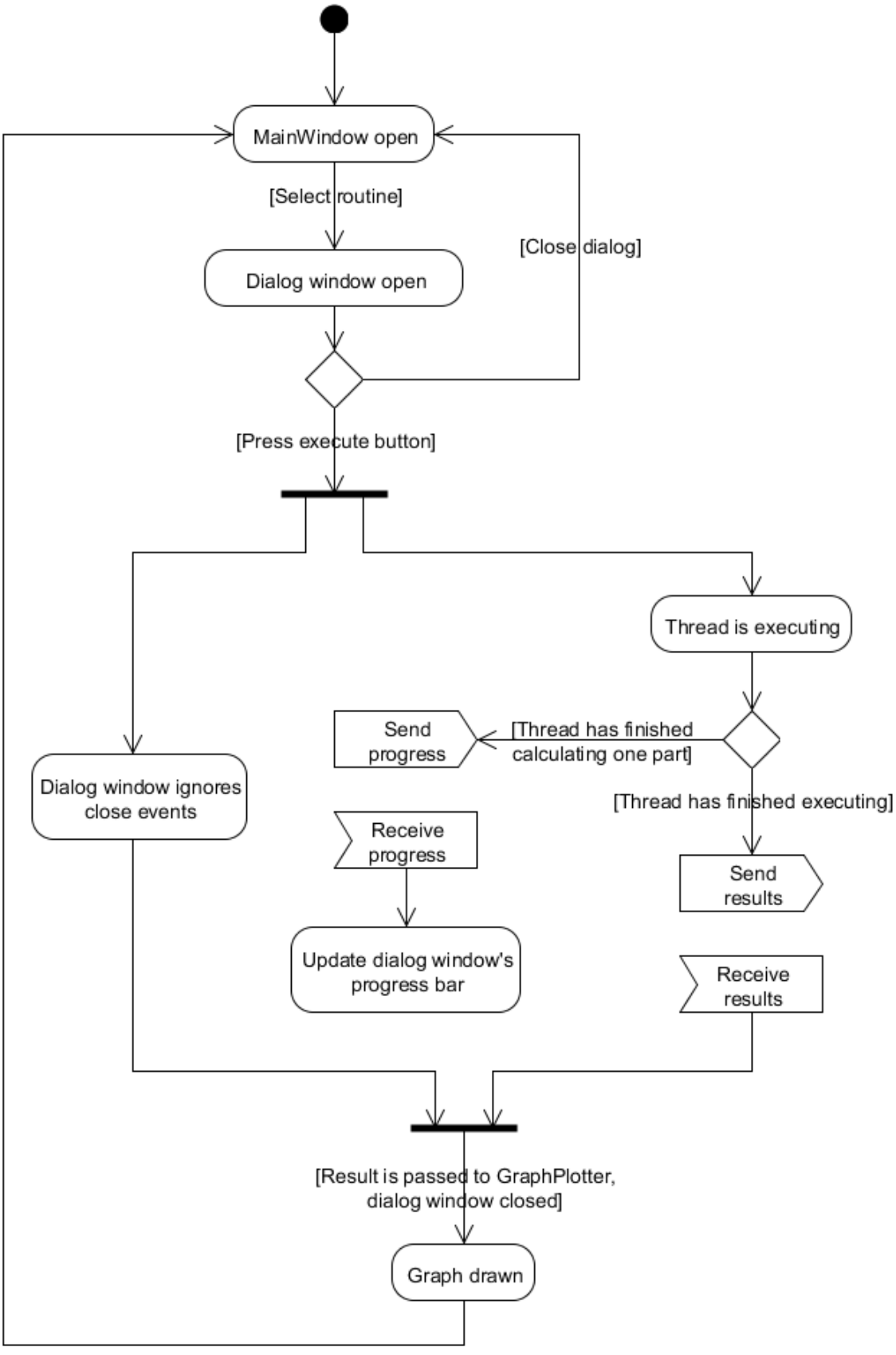


Figure 11: Overview of internal program logic

# Chapter 5 Evaluation

## 5.1 Results

Taking the parameters for the numerical results in [Transmission of a two-dimensional Gaussian beam into a uniaxial crystal], for a LiNbO3 crystal, and applying them to the input for NewAnalyticIntensityDialog, an attempt to replicate the figures 2, 5, 6, 7 was performed, as the access for the function that calculates the transmission of a non-truncated 2D TM Gaussian beam for negative crystals is achieved through this dialog window.

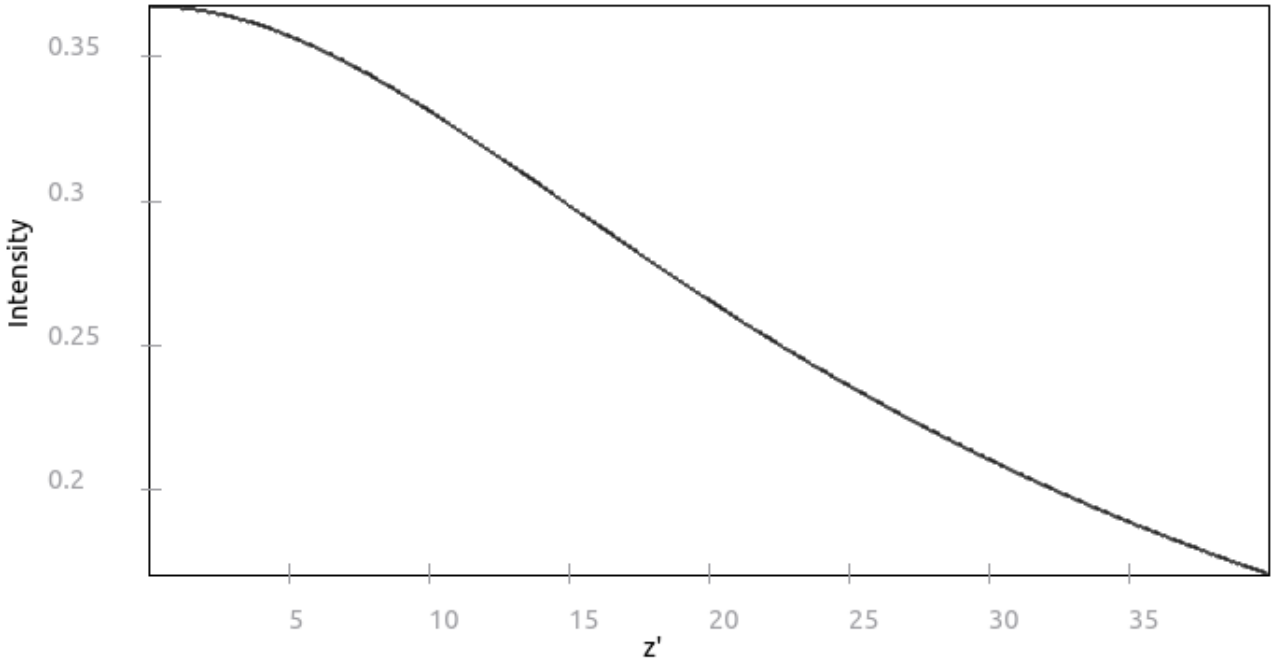


Figure 12: Total intensity 1 where $\theta = 0$ ( $s_x = 0.0$ , $s_z = 1.0$ )

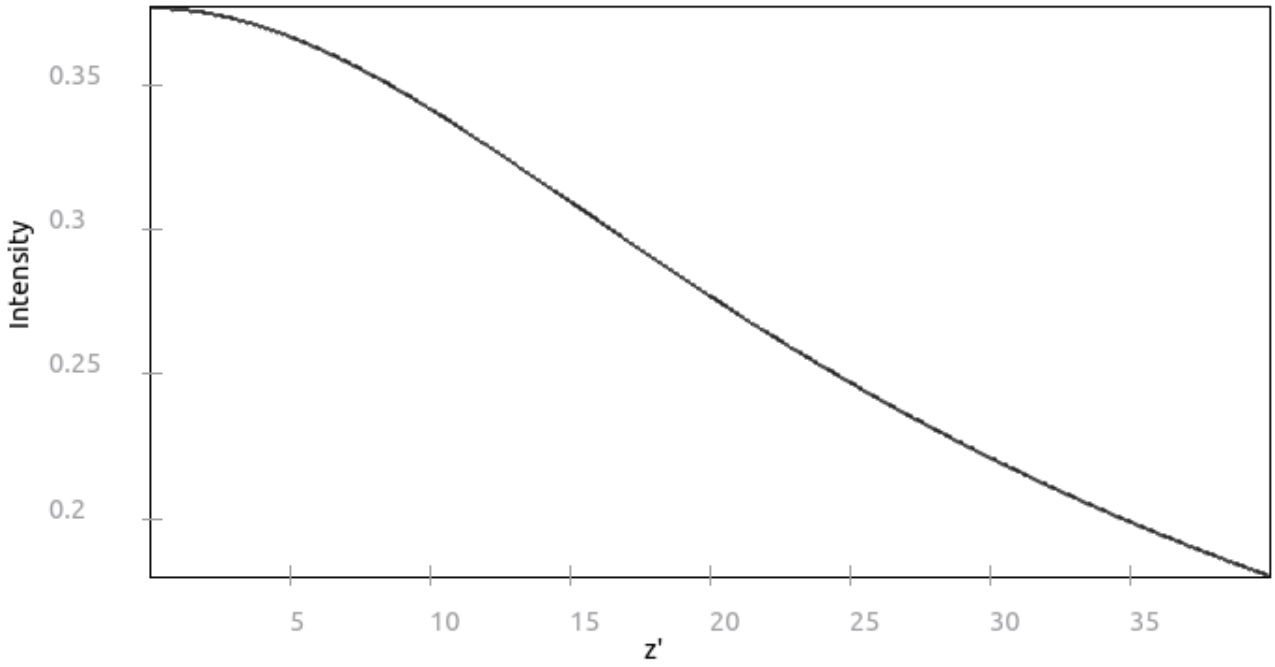


Figure 13: Total intensity 2 where $\theta=\frac{\pi}{4}$ ( $s_x=0.766$ , $s_z=0.766$ )

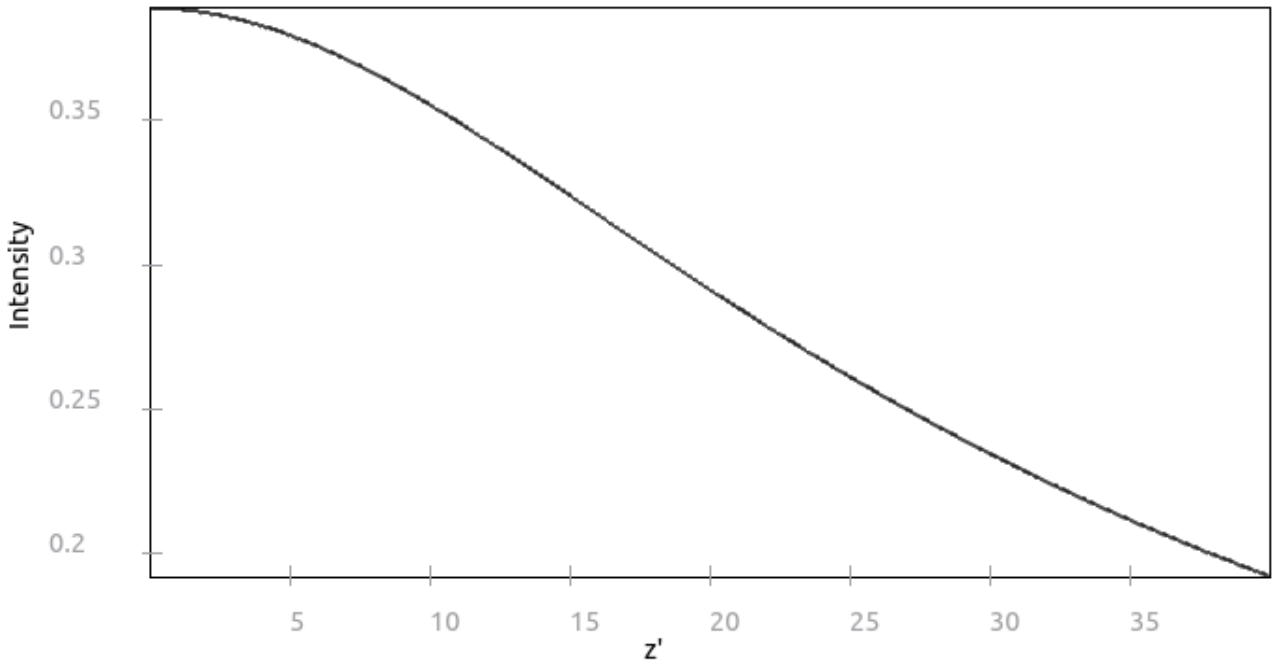


Figure 14: Total intensity 3 where $\theta=\frac{\pi}{2}$ ( $s_x=1.0$ , $s_z=0.0$ )

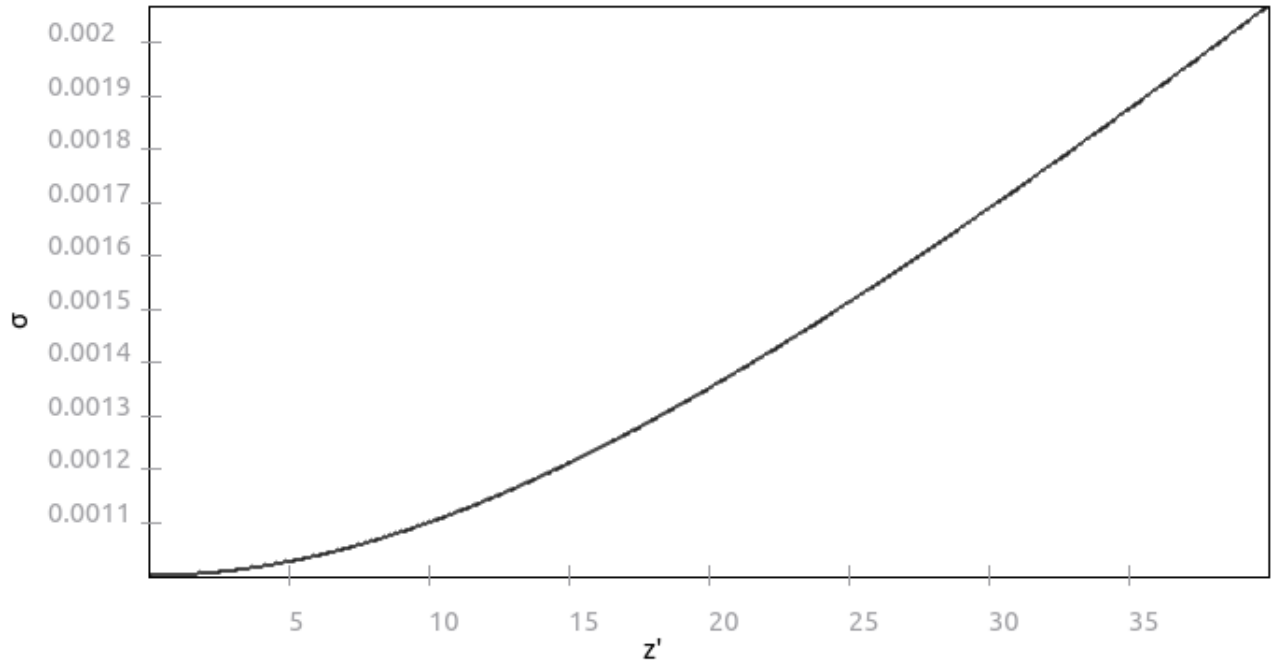


Figure 15: Beam width $\sigma$ , where $\theta=\frac{\pi}{4}$

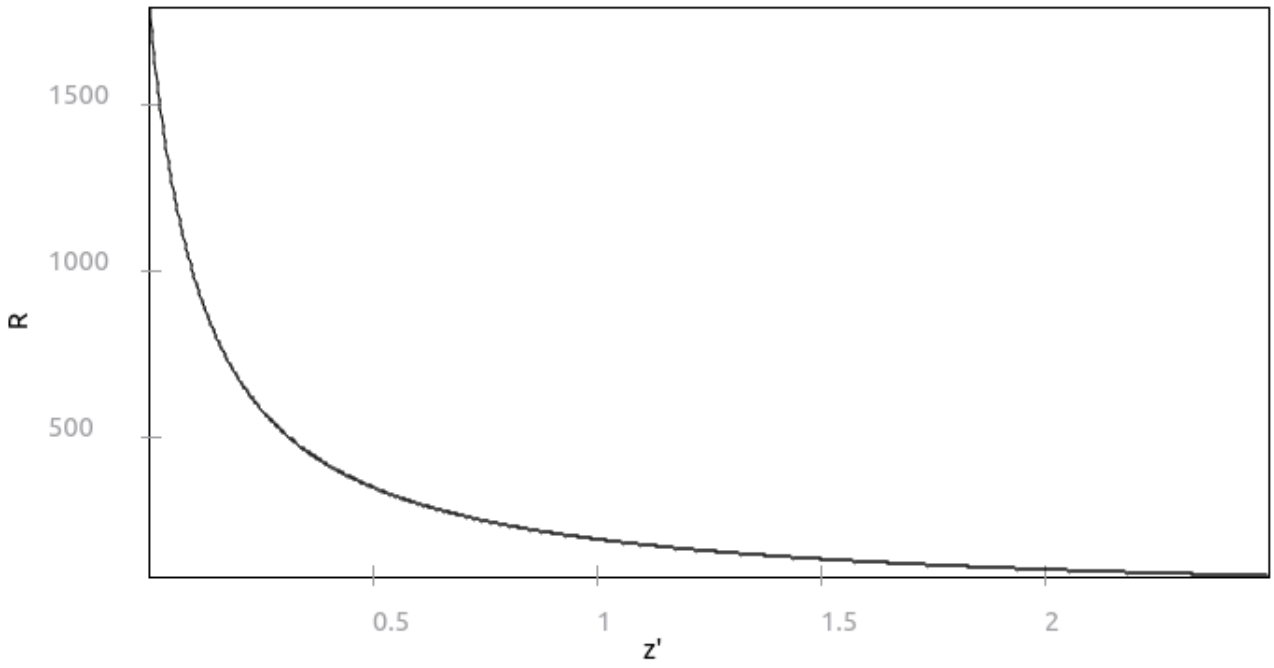


Figure 16: Curvature R of transmitted beam, where $\theta=\frac{\pi}{4}$

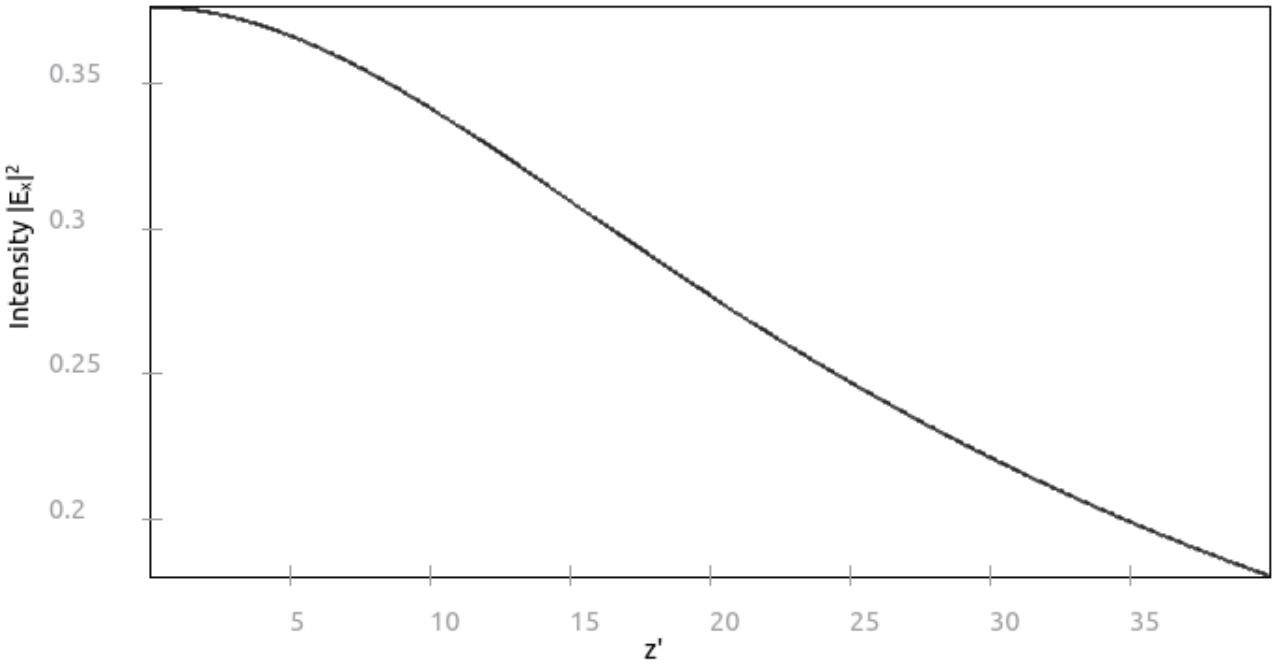


Figure 17: Copolarized axial intensity, where $\theta=\frac{\pi}{4}$

No attempt to understand the full implications of these graphs is made here, these are just for demonstration purposes, using known values, to demonstrate the correctness of the solution against existing results.

In the interest of seeing how the functions accessed through GaussXIntensityDialog and AnalyticIntensityDialog compared to the function accessed through NewAnalyticIntensityDialog, the same input values were used, where $\theta=0$ .

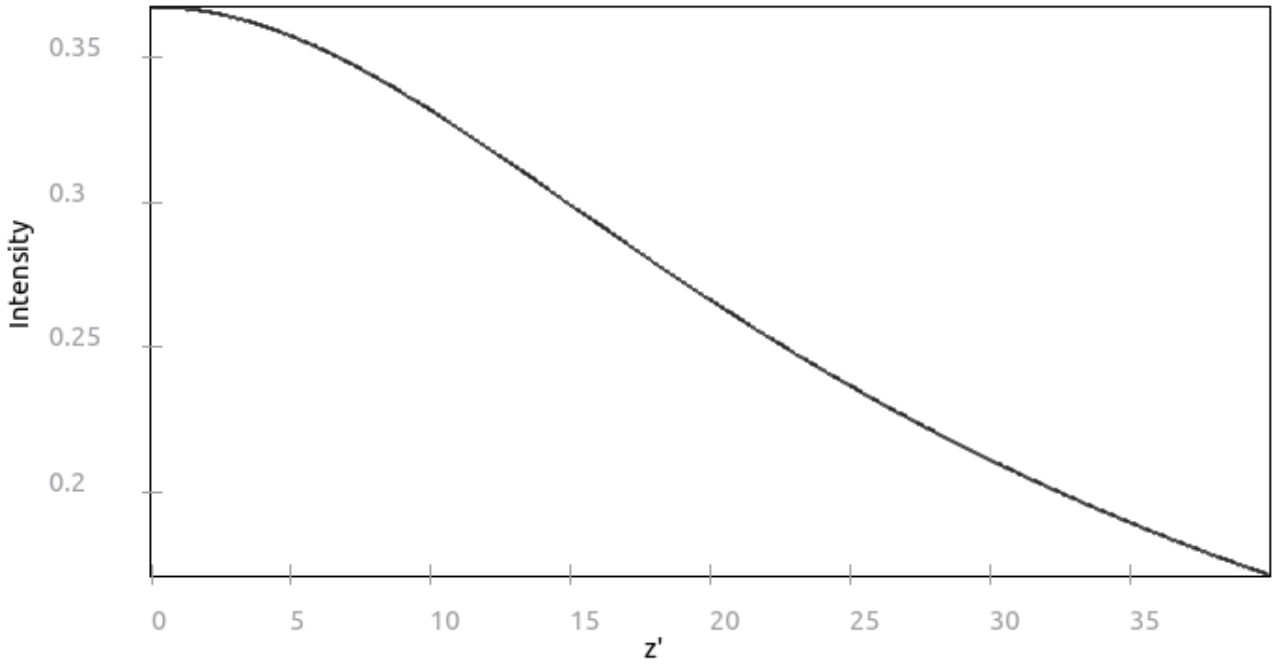


Figure 18: Output due AnalyticIntensityDialog

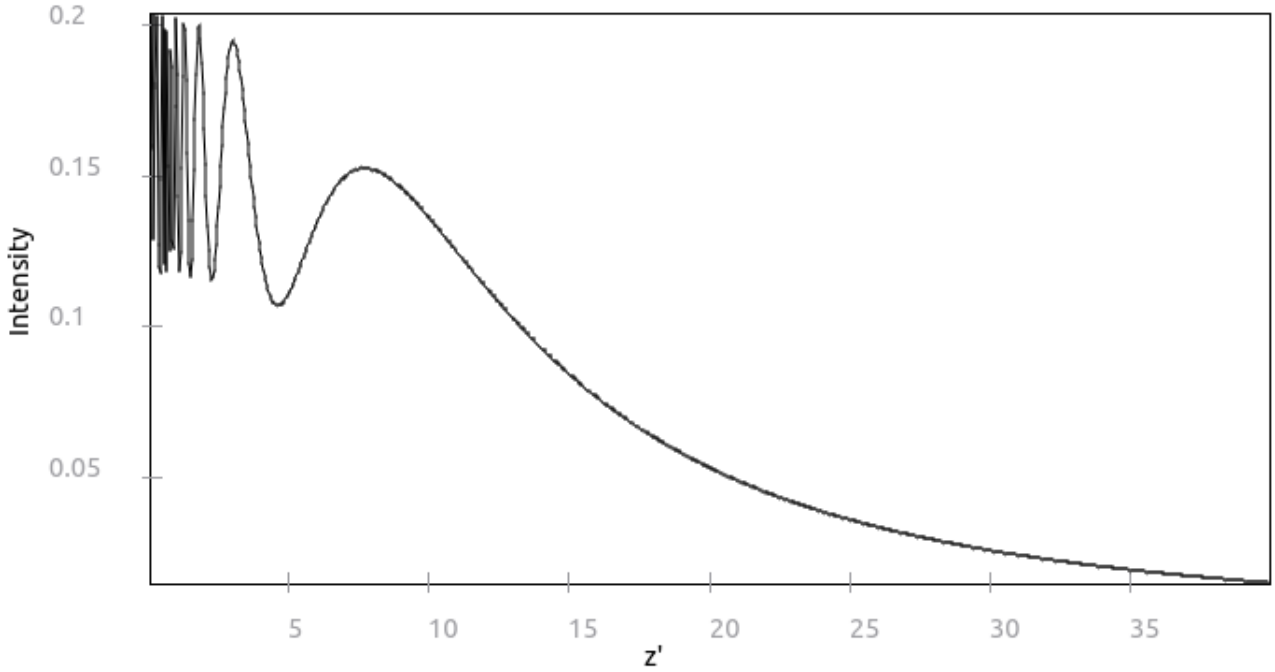

Figure 19: Output due GaussXIntensityDialog

Although the output due AnalyticIntensityDialog has a similar form to the output due NewAnalyticIntensityDialog, the output due GaussXIntensityDialog expresses a more complicated, but weaker transmission. It is not the purpose of this thesis to explain why this happens, but this output remains true to that of the original Fortran program. As the two other functions rely on greater approximations than GaussXIntensityDialog's function, this explains some of the difference.

## 5.2 Testing

Because a lot of the application revolves around the UI and graphical elements, which carry properties that depend on how the window of the graphical element is visualised, it is hard to test the graphical elements without getting visual confirmation. This meant that to see if a graphical element was being displayed and working correctly, the application needed to be compiled and ran to confirm the functionality and visuals. At the application's current complexity, this was non-trivial, the compile time not being so long that I had to wait for the application to run. Each time a major UI element was introduced, it's functionality would be tested in this way, to ensure that any graphical elements belonging to it was rendered, and that any actions performed towards the interactive elements were registered and handled.

Because I didn't use a testing framework, there was no white-box testing, which is to automatically tests the internal structures of a program using testing instructions. The reason I chose to not white-box test the application was because I was unfamiliar with any C++ testing frameworks, and uncertain of how the testing frameworks would work towards Qt objects and the Fortran functions. Since the source code for classes defined as Qt objects technically isn't C++-compliant until after qmake has made them into meta-object code, which is C++-compliant, it would be difficult to predict whether a testing framework would take this into account when executing the tests. And because the Fortran functions are static, and for C++ considered, only exists in the object code, the situation is similar to the former situation.

Black-box testing, which is to check what the program does, not how it does it, was however performed. In the beginning, a second-degree polynomial function was used to check whether GraphPlotter would render the function correctly, and what would happen if the scales and focus was changed around. Because the lower bound for the polynomial function was a bigger negative value than the positive value of the upper bound, it was expected that the function would slope down from the left, and then slope up to a lower point than leftmost part. This way, it would be possible to see if the x- and y-coordinates were not rendered the wrong way around. For the Fortran functions, the output of the threads interacting with these was compared with the output of the original Fortran programs.

If the application crashed, debugging with Qt's debug tool would be used, to trace the steps the application would take along the path of actions that had made it crash. In most of the cases where application crashed, it was due to the Fortran interaction functions, where trivial problems such as neglecting to define a function name was the culprit.

# Chapter 6 Summary

## 6.1 Conclusion

With this thesis, I have defined the problem and subproblems, analysed how the subproblems could be solved, and shown how Fortran can used in a graphical application and how a graph would be plotted besides on a series of two-dimensional values. The work with the thesis has provided important lessons to myself, and given me insight for the physics behind birefringent material.

At the start of the thesis work, I had originally opted for modifying the existing Fortran programs to accept console input, and use Qt's QProcess to interact with the programs, with directives provided by a configuration file. But all the time before switching the strategy, I was concerned that the configuration file directives wouldn't always be correct and that the solution would create needless complexity, and I am glad that I discovered how GCC solves cross-language programming, even though I think it shouldn't be necessary to guess how the compiler names a function in the object code.

The use of an Agile development methodology didn't go as planned either, the customer collaboration promoted by Agile being a failure mainly on my part, and since I was working by myself on the project, it was easy to be unfaithful to the Agile principles.

Because I already had gone from one subject in which we were developing graphical Qt applications, it was easy to get back into the flow of working with Qt, although I still had to rely on the Qt 4.8 documentation. Fortran was an entirely new field for me, so no attempts to delve deep into existing program logic were made, for example, when trying to implement the logic from two other programs and these returning uninitialised values, I suspected that deeper down the problem could stem from data types being inconsistent, but I didn't have enough knowledge about Fortran to troubleshoot this.

I'm not entirely satisfied with how the solution turned out, and I will provide suggestions to further improvements for the solution.

## 6.2 Future work

As it stands now, users are not able to save the numerical results resulting from the routines, preventing them from analysing the numbers closer with resorting to the data entries in the QTableWidget belonging to MainWindow. Therefore, it would be desirable to add a class that not only took care of saving these results to a file in a certain format, but also load entries in the same format to the program for GraphPlotter to plot onto a graph.

GraphPlotter could be extended so it can render more than one graph at the same time in the same context, the user being able to choose different outlining styles and colours for each graph. The save to image function should also be fixed, so that depending on the image format the user chooses, the filename extension is automatically appended to the filename, and checks whether it won't save over an existing file. The folder the user saves to should also be remembered for the next time the user wants to save an image, whether for the application's lifetime or for the next time

the application is execute is up to personal preference. This should have to be accomplished by using the file dialog window native to the OS.

The dialog windows and threads should really be inheriting from abstract classes, that inherit from QWidget and QThread respectively. Although the Fortran functions have been granted extended life by this solution, it could be worth it to remake them into thread-safe functions in C++. An even better method would be to integrate a script engine, and remake the functions into scripts to be processed by the script engine.

# Appendices

## Appendix A

As written, each of the dialog windows were based on Fortran programs I had received from my supervisor.

**GaussXIntensityDialog** was based on a program called unigauss. This program uses the equation for calculating the electric field using a form similar to that for scalar Gaussian beam propagating in an isotropic medium [Equation 1]. This program was the most advanced one, with several different independent functions, and also the slowest. From this program, I extracted 17 clear input variables which are echoed in the dialog window, the class of the thread belonging to GaussXIntensityDialog having a variable for each of these. The thread is in charge of calling an interaction function in Fortran called GAUSSXAXIS() for each iteration of its for-loop, from which it extracts $|E_x|^2$ at a point of the uniaxial crystal.

**AnalyticIntensityDialog** was based on a program called zanalytic. This program uses a simplified equation for calculating the electric field when the aperture size $a$ is larger than the beam waist σ [Equation 2]. From this program, I extracted 13 clear input variables, which are echoed in the dialog window, the class of the thread belonging to AnalyticIntensityDialog having a variable for each of these. The thread is in charge of first calling an interaction function in Fortran called INITIALISEGAUSS(), before it starts calling an interaction function called ANALYTIC() for each iteration of its for-loop. Because there's a choice for the output in the dialog window, the output of from the thread is based on the choice, which is either the total intensity $E=E_x+E_z$, the intensity's x-component $E_x$, $\tilde{Z}$, or beam width σ at the point.

**NewAnalyticIntensityDialog** was based on a program called znewanalytic. This program uses a variation for the equation for calculating the electric field when the aperture size $a$ is larger than the beam waist σ [Equation 2], for when the crystal is a negative uniaxial crystal. From this program, I extracted 14 clear input variables, which are echoed in the dialog window. The class of the thread belonging to NewAnalyticIntensityDialog inherits from the class of the thread belonging to AnalyticIntensityDialog, as the variables of these two programs are equal, except for the one zanalytic doesn't have. Because it inherits AnalyticIntensityDialog, it also calls INITIALISEGAUSS(), but calls an interaction function called NANALYTIC() in its for-loop. Because there's a choice for the output in the dialog window, the output of from the thread is based on the choice, which is either the total intensity $E=E_x+E_z$, the intensity's x-component $E_x$, the curvature $R(\tilde{Z})$, or beam width σ at the point.

## Appendix B

In order to compile the application, g++, gfortran and the Qt SDK needs to be installed. To install g++, gfortran, and the Qt SDK on Debian GNU/Linux distributions, use the following commands:

1. sudo apt-get install g++
2. sudo apt-get install gfortran
3. sudo apt-get install qt-sdk

To compile the application, first extract the ZIP-compressed archive fortemvisrelease.zip:

1. Copy fortemvisrelease/nbproject/qt-Release.pro to fortemvisrelease/
2. Run “qmake -makefile qt-Release.pro” in fortemvisrelease
3. Run “make”
4. (Optional) Run “make clean”

The compiled executable will be located as fortemvisrelease/dist/Release/fortemvisrelease.
It is also possible to load the project into NetBeans IDE 8.0 for C/C++ by importing fortemvisrelease.zip.

# References

GAU1: Stamnes, Jakob J.; Velauthapillai, Dhayalan, Transmission of a two-dimensional Gaussian beam into a uniaxial crystal, 2001
GAU2: Stamnes, Jakob J.; Velauthapillai, Dhayalan, Double refraction of a Gaussian beam into a uniaxial crystal, 2012
QT: Various developers, Qt4.8 documentation http://qt-project.org/doc/qt-4.8/, ,
ASD: Martin, Robert C.; Newkirk, James W.; Koss, Robert S., Agile Software Development, 2012