Diffpack GUI
A portable and fully interactive application

Zlatko Hodzic

28th January 2004
Preface

This master thesis is written at the University of Oslo, Department of Informatics.

I would like to thank my mentors Xing Cai, Hans Petter Langtangen and Are Magnus Bruaset for their valuable input, help and encouragement during these two years it has taken me to finish the thesis.

I would also like to thank my family, friends and fellow students for their valuable time, support and patience.

Oslo, 28th January 2004
Zlatko Hodzic
Contents

1 Introduction .................................................. 1
   1.1 Background and motivation .......................... 1
   1.2 Scope and overview .................................. 2

2 Toolkits ...................................................... 5
   2.1 Diffpack .............................................. 5
   2.2 Qt .................................................. 11
   2.3 VTK ............................................... 12
   2.4 vtkQt ............................................... 13

3 Design ........................................................ 15
   3.1 Object oriented design .............................. 15
   3.2 Control ............................................. 17
   3.3 MenuTree ............................................ 18
   3.4 SimResBrowser ..................................... 20
   3.5 Visualizer .......................................... 20

4 Implementation ............................................. 23
   4.1 MenuTree ......................................... 23
   4.2 SimResBrowser ..................................... 28
   4.3 Visualizer ......................................... 29
   4.4 Control ............................................. 34

5 Using Diffpack GUI ......................................... 39
   5.1 Control ............................................. 39
   5.2 MenuTree ........................................... 40
   5.3 SimResBrowser ..................................... 41
   5.4 Visualizer ......................................... 42
   5.5 Compiling and running Diffpack GUI .............. 47

6 Conclusion and Reflection ................................. 49
   6.1 Wrapping up ........................................ 49
   6.2 Future work ......................................... 50
# List of Figures

2.1 Example of an input file ..................................... 7
2.2 Snippets from a `menutree` file .............................. 8
2.3 Two examples of a `.CASENAME.simres` file ............. 9

3.1 Mental model of the GUI application ......................... 16
3.2 Data structure of the GUI application ........................ 17

4.1 Code snippet of the header file `MenuTree.h` ............ 24
4.2 Code snippet of the header file `Submenu.h` ............ 25
4.3 Code snippet of the header file `Item.h` ................. 27
4.4 Code snippet of the header file `SimResContainer.h` ... 29
4.5 Snippet of the function involved with inserting a solution field in a `SimResContainer`, in file `SimResContainer.cpp` ... 30
4.6 Code snippet of the header file `SimResField.h` ....... 31
4.7 Functions from the `QtSimResBrowser.cpp` file that are involved in visualizing a solution field from the `SimResBrowser component` .............................................. 32
4.8 Functions from the `QtControl.cpp` file that are involved in visualizing a solution field from the `SimResBrowser component` .............................................. 33
4.9 Functions from the `Control.cpp` file that are involved in visualizing a solution field from the `SimResBrowser component` .............................................. 34
4.10 Code snippet of the header file `Visualizer.h` ......... 35
4.11 Code snippet of the header file `QtVisualizer.h` ....... 36
4.12 Code snippet of the header file `Control.h` ............ 37
4.13 Code snippet of the header file `QtControl.h` .......... 38

5.1 Control window ............................................... 39
5.2 File dialogue for simulation menu ........................... 40
5.3 MenuTree and `SimResBrowser` windows ................... 41
5.4 File dialogue for `SimResBrowser` ......................... 42
5.5 Visualization of a 2D and 3D scalar field .................. 43
5.6 Visualization of a vector field using the `hedgehog` visualization technique ........................................ 44
5.7 Windows that control the contouring of the scalar field, extracting of a part of a solution field, and cutting a slice plane through a 3D solution field ...................... 44
5.8 Isolines in a 2D scalar field left, and surface rendering of the same scalar field right ......................... 45
5.9 Widget for setting the background color .................... 46
5.10 At the left hand side is an extracted domain of the scalar field shown at the right hand side ................ 47
Chapter 1

Introduction

In this chapter I introduce my thesis and the GUI application, explain the motivation and the motives in the background. I am also defining the scope of the thesis and presenting a short view of the following chapters.

1.1 Background and motivation

Diffpack (see [1] and [2]) is a collection of the C++ libraries used to numerically solve the partial differential equations (PDEs), using the object oriented programming in the C++. Diffpack can also be regarded as a collection of the basic simulators used as the building blocks for creating the more problem specific PDE simulators. In the Diffpack version for the Unix/Linux platforms, these basic simulators are mostly run from the command prompt, unlike in the Windows version where a fully interactive graphical user interface (GUI) has existed for a while (see [1], Appendix B.2.2). Some attempts have been made to change this, and several smaller GUIs, like gui.pl (for adjusting the simulation parameters and running the simulation), simresgui (for exporting the simulation results to different visualization formats) and vtkviz (for visualizing the simulation results in VTK), have been programmed, although none of these are functioning as a unified application with GUI (for examples see [1]).

This is obviously of no advantage to the users who are accustomed to the Windows' GUI solution, in which they can test the existing simulators. The users also have a possibility of incorporating their own, self developed, problem specific simulators in the Windows' GUI solution, by using the Diffpack GUI library. This library is specifically designed for this purpose, and is included only in the Windows version of Diffpack. To test a simulator on the Unix/Linux platforms the user has to give a
prompt command in order to start a simulation. After waiting for the
evolution of a simulation to end, the user then has to transform the sim-
ulation results to a visualization format of choice. Finally, the user can
now visualize the simulation results, either by using his own visualizing
solutions or by using the already available visualizing tools. From now
on this set of operations and processes will be referred to as the execu-
tion chain of a simulation process, that is, the complete process of the
simulation and presentation of the simulation results.

The aforementioned smaller GUIs (gui.pl, simresgui, vtkviz), bypass
the command prompt and offer the user a slightly more intuitive option
of a window based interaction with a simulator. These GUIs cover all
of the aforementioned processes in the execution chain, but their main
problem is their lack of interaction with each other, as well as being
rather "old fashioned". Some of them are also offering only a few op-
tions to the user.

If the user wants to view the simulation results he has to start an
application, simresgui for example, which displays all the solution fields
from a given simulation, and export the chosen fields to a visualization
format of choice. In order to visualize these fields, the user then has
to start another, visualizing application, for example vtkviz or mayavi.
Obviously, this process is not very user friendly.

The main purpose of my thesis is therefore to make a GUI application
which would combine all these processes into one unit, and enable the
user to bypass the prompt line.

1.2 Scope and overview

Making such an application is not a trivial task. There are many differ-
ent, and sometimes complicated, aspects to consider and to handle. The
sheer complexity of the application itself grows rapidly as one adds the
additional components and options to it. This, in itself, is one of the
limitations of my thesis, as I do not have such a comprehensive know-
ledge of all the possible details involved in the greater picture which is
a "complete" or "definite" GUI application for Diffpack (if such exists at
all). My objective is therefore to combine all these different, aforemen-
tioned GUIs, and "update" all of them, while at the same time keep the
application simple and easily manageable.

During this process, in co-operation with my mentors, I had to make
the certain decisions and choices regarding the final outcome of the GUI
application. All these choices are discussed in detail in the following chapters.

I start with the explanation of the different programming packages and programs used in the GUI application, in Chapter 2. In this chapter, I am also explaining the reasons for choosing these packages and programs in detail.

Chapter 3 covers the issues regarding the build-up phase of the GUI application, explaining the details of the GUI application design, discussing the choices that have been made regarding the GUI application’s structure and comparing them to the other possible solutions.

Moving on to the implementation itself, Chapter 4 also covers the issues and the problems encountered during the production stages.

In Chapter 5, I give a simple run-through of the GUI application, using the simple examples to demonstrate the GUI application’s use and functionality.

Chapter 6 rounds off the work that has been done, reflecting on the things that could have been done better and/or differently. Some thoughts on the future work are also offered, mostly about the possible improvements and the expansion of the GUI application.
Chapter 2

Toolkits

In this chapter I would like to shed a little light on the different packages and programs used in the GUI application. The short descriptions of Diffpack, Qt and VTK are the key moments of the chapter, as well as explaining the reasons for choosing these toolkits.

2.1 Diffpack

**Diffpack**, as already explained in Chapter 1, is a collection of the C++ libraries, used to numerically solve the partial differential equations (from now on referred to as PDEs), using the object oriented programming in the C++. The PDEs are the mathematical representations of the several nature phenomena. Solving these equations offers the insight into the possible developments of these phenomena, but this process is very time demanding and difficult. Diffpack contains several different, pre-defined simulators that solve the most simple and universal PDEs. The numerical methods for solving the PDEs are used to solve the equations, while the object oriented programming of the libraries offers the user a possibility of easily upgrading and adjusting the simulators to fit the user’s own problems and equations. Diffpack also contains the other useful tools, offering the user a wider range of the solution methods for these simulators. Some of the utilization areas of these simulators are structural mechanics, water waves, heat transfer and aero dynamics, to name a few.

The latest Diffpack release, 4.0, is available on Windows and the Unix/Linux platforms. The Windows distribution has an integrated GUI application for controlling and monitoring a simulation process, and visualizing the simulated data (see [1] Appendix B.2.2), while there is no such a thing in the Unix/Linux distribution. There are some smal-
ler GUI applications (see [1], page 241 for the Menu System GUI, page 232 for simres GUI and page 238 for vtkviz), independent of each other, which allow the user to control and execute several separated parts of a simulation process.

The menu system GUI (as seen in [1], page 242, figure 3.3) is based on the MenuSystem class in Diffpack (see [1], page 536, table B.2, and page 224). This class holds the information about all the parameters in a simulator, and is used to access and set the parameter values. Understanding the MenuSystem class and its utilities is central in the development of one of the components of my GUI application. In Diffpack there are three different ways of feeding a simulator with the input data. When running a simulator from a command line, it is very common to feed the input data to the simulator by using an input file, which contains the data we want to use in the simulation. In figure 2.1 on the facing page we see an example of an input file. The parameters’ values used in a simulation are simply set by writing "set PARAMETER NAME = PARAMETER VALUE". To access some of the parameters in the lower submenus of the MenuSystem class the user has to access the submenu itself by writing "sub SUBMENU NAME". Now he can set the value of a parameter like previously explained. After the desired parameters in a submenu are set, it is necessary to exit the submenu by writing "ok". This is the first way of feeding the data to a simulator.

Another very common and convenient way of feeding a simulator with the input data is by running the Menu System GUI (see [1], page 241), which scans the simulator for all its parameters, and offers the user a possibility of changing the input data as desired (see [1], page 241). The third way is feeding the data by typing it in, in the command prompt, after the simulation execution has started. Whichever way of inputting the data the user chooses, all the values entered will be stored in the MenuSystem class of the simulator, and written to a file (.menutree) after executing the simulation. The snippets from the .menutree file can be seen in figure 2.2 on page 8. The .menutree file contains all the information about all the parameters used in a simulation. Each parameter's data is stored in a single line starting with the code word "item:”, with the character '@' separating the different types of information. For example the number between the first two '@'s is the depth level at which this parameter is located in the MenuSystem class, followed by the parameter's name, and so on. Many of these parameter data are not essential for the GUI application, as I will explain later, in

---

¹ The component in question is the MenuTree component, and is explained in detail in Chapters 3.3, 4.1 and 5.2
set time parameters = dt=0.1 t in [0,13.0]
set gridfile = PreproStdGeom | BOX_WITH_ELLIPTIC_HOLE
   a=0.25 b=0.25 c=0.25 d=0.25 deg=45 | d=2
e=ElmB4n2D [10, 20] [1, 1]
set theta = 1.0
set redefine boundary indicators = n=5 names = d1 d2 g
dudn robin 1=() 2=() 3=() 4=(2 4) 5=(1 3)
set Ts = 5.0
set A = 1
set omega = 1
set Robin u value = 1000
sub LinEqAdmFE
sub Matrix_prm
set matrix type = MatSparse
ok
sub LinEqSolver_prm
set basic method = ConjGrad
ok
sub Precond_prm
set preconditioning type = PrecRILU
set RILU relaxation parameter = 1.0
ok
sub ConvMonitorList_prm
sub Define ConvMonitor #1
set #1: convergence monitor name = CMAbsResidual
set #1: residual type = ORIGINAL_RES
set #1: convergence tolerance = 1.0e-6
ok
ok
sub SaveSimRes
set time points for plot = ALL
set field storage format = ASCII
set grid storage format = ASCII
ok
ok

Figure 2.1: Example of an input file
MenuLists@Poisson2
...
item:@1@beta@.@-@beta@.@beta*u term in equation@S@0.000
...
submenu:@2@LinEqSolver_prm@1
  @Linear solvers and associated parameters@0
MenuLists@LinEqSolver_prm
...
item:@3@shift parameter@.@-@Symmlq_shift_parameter@. @shift parameter for Symmlq iterations@R[0:2]@0.000
...

Figure 2.2: Snippets from a .menutree file

Chapter 4. The lines containing the information about submenus start with a code word "submenu:" and have pretty much the same syntax, although not as much information, as the lines containing the information about the parameters. An empty line indicates the end of the information for a submenu.  

Diffpack generates a data set of five hidden files for every simulation execution, which is Diffpack's own storage format called simres. These files are as follows:  

- **.CASENAME.simres** - A file containing the list of all the solution fields generated by a simulation execution. This file is the most important file for one of the components in my GUI application, because it contains all the necessary information I need to generate the visualizations of a solution field. This file does not contain the solution fields themselves, but only an overview of all the solution fields generated by the simulation execution. Two examples of a **.CASENAME.simres** file can be seen in figure 2.3 on the facing page. In the first column is the solution field's ordinal number in the set of the solution fields. The second column indicates the solution

### Notes

2 Some of the lines in both .menutree and input file have been edited so that they are placed within the figure bounds.

3 **CASENAME** is the name of the simulated result field given by the user under the simulation execution. If no name is given then **CASENAME** is automatically set to **SIMULATION**.

4 The component in question is the SimResBrowser component, and is explained in detail in Chapters 3.4, 4.2 and 5.3.
<table>
<thead>
<tr>
<th></th>
<th>field</th>
<th>type</th>
<th>value</th>
<th>Field</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>u</td>
<td>stationary</td>
<td>scalar</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>flux</td>
<td>stationary</td>
<td>3D-vector_1</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>flux</td>
<td>stationary</td>
<td>3D-vector_2</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>flux</td>
<td>stationary</td>
<td>3D-vector_3</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>flux_magnitude</td>
<td>stationary</td>
<td>scalar</td>
<td>FieldFE</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>field</th>
<th>type</th>
<th>value</th>
<th>Field</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>u</td>
<td></td>
<td>0</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>u</td>
<td></td>
<td>0.1</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>flux</td>
<td></td>
<td>0.1</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>flux</td>
<td></td>
<td>0.1</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>u</td>
<td></td>
<td>0.2</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>flux</td>
<td></td>
<td>0.2</td>
<td>FieldFE</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>flux</td>
<td></td>
<td>0.2</td>
<td>FieldFE</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.3: Two examples of a `.CASENAME.simres` file

field's name. The third column denotes the time level this solution field is at. In the example file on top of figure 2.3 all the solution fields are stationary, so there are no time levels. In the fourth column is the information about what kind of solution field this is. It can be either a scalar or a vector component in 2D or 3D (and in x or y and x, y or z direction respectively). The other two columns are neither important for, nor used by my GUI application, and thus not explained here.

- `.CASENAME.field` - This file contains the values of the solution fields. Its associating grid is placed in the `.CASENAME.grid` file.
- `.CASENAME.field.ix` - An index file for direct access of the values in the `.CASENAME.field` file.
- `.CASENAME.grid` - A file that contains the grid objects. The associated fields are stored in the `.CASENAME.field` file.
- `.CASENAME.grid.ix` - An index file for direct access of the grid objects in the `.CASENAME.grid` file.

Another important, and for my GUI application very central, aspect of Diffpack is its ability to generate the files containing the solution results in several different types of the visualizing formats, without having to rerun a simulator. This is done thanks to the several utilities that transform and filter the data saved in the `simres` format to the other visualiz-
ing formats. An example of a simres filter utility is the simres2vtk utility, which generates a file in the VTK\(^5\) format. Executing the following line\(^6\) in the command prompt

\[
\text{simres2vtk} \ -f \ \text{CASENAME} \ (-n \ \text{SOLUTION\_FIELD\_NAME} \ | \ -r \ \text{SOLUTION\_FIELD\_NUMBER}) \\
\quad (-s \ | \ -v) \ [-t 'TIME;' \ ] \ (-a \ | \ -b)
\]

will generate a VTK file. This file can be used in association with vtkviz to visualize the solution field. The simres2vtk options in the line above are the following:

- **-f CASENAME** - Indicates the name of the simres file set which holds the information about all the solution fields.

- **-n SOLUTION\_FIELD\_NAME** or **-r SOLUTION\_FIELD\_NUMBER** - Picks the solution field either by its name, or by its ordinal number in the simres file.

- **-s** or **-v** - Indicates whether a solution field is a scalar or a vector field.

- **-t 'TIME;'** - In case a solution field is specified by its name, and the solution is not stationary, this option picks out a distinct field called SIMULATION\_FIELD\_NAME at time TIME. Otherwise, this option can be omitted.

- **-a** or **-b** - Indicates whether the output file is in ASCII or binary format.

There are also some other options for the simres2vtk utility, but since they are not important for my GUI application I will not explain them here either. For more information about these options and the other filter utilities see Hans Petter Langtangen’s book [1], for the extensive descriptions. The simres GUI is largely based on manipulating the information from the simres format, and using the simres filters.

As none of the aforementioned smaller GUIs (menu system GUI, simresgui, vtkviz) are functioning as a unified application with GUI, my goal is to combine the ideas from all of these applications, and make a new GUI application that would cover all the tasks done by the old separated, applications. This new GUI application would also have to be as portable as possible, and its data structure should be as flexible as possible. By portable application I mean a platform independent application, since

---

\(^5\) VTK is discussed in Chapter 2.3
\(^6\) Again, the line is split in order to fit into the boundaries of this document
Diffpack itself is a multiplatform product. And by flexible data structure I reckon that the application’s data structure and source code is easily reused by the future developers. If not all of it than at least the most of it.

2.2 Qt

As one of the criteria for this new GUI application is the portability (in other words, the GUI application should function on both Windows and the Unix/Linux platforms, with no changes to the source code), I had to find the suitable GUI libraries, which provide such a flexibility. As Diffpack itself is written in the C++ programming language it feels natural to write my GUI application in the C++ as well, using the same object oriented approach, such that the choice of the GUI libraries itself is rather marginalized. The future developers should be able to neglect my particular GUI solution and start developing their own solution, using some other GUI libraries while at the same time being able to use the core of my solution, which is totally independent of all the GUI libraries. This way they would not have to develop from the scratch, but rather integrate their own solution on top of my core. This would satisfy the second important criteria which is the data structure flexibility.

Having already explained the two rather strict criteria for my GUI application, the choice of the GUI libraries itself is pretty much straightforward. There is only one GUI library that offers such portability and flexibility, and it is Qt. This product, developed by the Norwegian company Trolltech AS (see [3]), makes it possible to write the same source code for any of its supported platforms, and it runs natively on all of these. The supported Qt platforms are Windows, several Unix/Linux versions, Apple Mac and embedded Linux. Another strong advantage of Qt is that it is a commercial product, and as such has the very well developed support and documentation. It is frequently maintained and updated, and has a strong user community, discussing both the problems and the development issues. Besides, it is fairly easy to use, and it is rather intuitive.

It is also interesting to point out that several wrappers\(^7\) on top of Qt have been developed, making it possible to use Qt with both Perl and Python. None of these wrappers are officially developed, supported nor maintained by Trolltech, but by the third parties, so even though they

\(^7\)A Wrapper is an interpreted interface layer, making it possible to use the package libraries written in one programming language, while programming in a completely different programming language
could have made the development process of my GUI application much easier, I decided to disregard them and use the official C++ implementation.

The other GUI solutions that I evaluated at the early stages of the production process are Motif (see [4]) and GTK+ (see [5]). While both are, by all means, very efficient, none of them has the flexibility and the platform independence of Qt. It is also fair to point out that Diffpack already has a well functioning GUI on Windows, so why bother reinventing the wheel, and insist on making this new GUI application so portable? Both Motif and GTK+ have a wide range of the GUI widgets\(^8\), both are well documented, both have a strong user community, and, like Qt, they are both completely free provided that you are operating on the GNU public license (see [6]). Which are all quite legit reasons for choosing one of the other two GUI libraries. That is, if you want to maintain two different codes in the future, one for Windows and the other for the Unix/Linux platforms. Rather a case of improving the wheel, than reinventing it, choosing Qt offers the future developers a possibility of refining and improving the GUI application, and then porting it to Windows without any problems. This portability is where Qt wins against the two, because it enables the developers to maintain only one source code, and recompile it on the different platforms to make it work on each of them.

2.3 VTK

In order to make a well functioning GUI application I also had to evaluate the different visualization toolkits. As explained in Chapter 2.1, the solution data generated by a simulation execution can be transferred to the data files in several specific visualization formats, such as Plotmtv, Gnuplot, Matlab, VTK, etc. Unlike VTK, none of these visualization toolkits can offer such a high degree of the user interaction, combined with the platform independence.

VTK is an acronym of Visualization Tool Kit, which is a collection of the C++ libraries developed by Kitware Inc., used for image processing, 3D computer graphics and visualization (see [7]). VTK supports a wide variety of the visualization algorithms (scalar, vector, tensor, etc.) and the advanced modelling techniques (cutting, contouring, implicit modelling, etc.) As well as Diffpack and Qt, VTK is developed by strongly fol-

---

\(^8\) A GUI widget is any element you see in a GUI application. Button, check box, open file menu, etc. are all the GUI widgets. They can be simple, like a label, or more complex, like a list view, for example.
lowing the object oriented principles. It is developed on top of OpenGL (see [8]), so that all the low level graphics programming is omitted from the development process, enabling the users to concentrate on the visualization techniques instead.

VTK is an open source package, available on Windows, the most Unix/Linux platforms and Apple Mac. Unlike Qt, VTK has the several officially developed and maintained wrappers. Such wrappers for VTK exist in Java, Python and Tcl/Tk.

2.4 vtkQt

Another strong case for using VTK instead of some other, Diffpack supported, visualization package, is its interconnection with Qt. There have been several attempts to successfully link Qt and VTK through Qt’s OpenGL module. To my knowledge there are four such wrappers, and I have managed to make only one of these function without any problems. This small package, called vtkQt (see [9]) is developed by Matthias Koen- ing, then, of the Department of Simulation and Graphics at the Otto von Guericke University of Magdeburg. It is small, it is fairly simple to use, it supports the user interaction very well, and it is easily incorporated with Qt the widgets.
Chapter 3

Design

Chapter 3 covers the issues regarding the design of the GUI application, commenting the choices that have been made, as well as shortly discussing the other possible solutions.

3.1 Object oriented design

Having chosen the C++ as the programming language for my thesis, and one of my main goals being as high flexibility as possible, it is only natural that I have to follow the object oriented principles while developing the Diffpack GUI application. I have to structure the program in such a way that the later changes in the design and layout of the GUI application itself are cut to minimum. This frame of thought was determined in the very beginning, during the first consultations with my mentors, as we agreed that the GUI application should be based on a specific mental model. As it can be seen from figure 3.1 on the following page it consists of three parts. The first part is totally independent of any GUI solution, and it handles the interaction between the operating system and the GUI application. It is written in the pure C++, and it handles the interaction between the GUI application and the operating system. The second part is a specific GUI implementation of the GUI application, and should be easily replaceable in case the future developers decide to use another GUI package instead. Putting together these two parts is the last part of the model. This part represents the guidelines for any GUI solution to follow, and it is written in the pure C++. This generic GUI solution is the connecting point between the other two parts, and should be used as it is in the future development.

My GUI application had to deal with three separate tasks: running a simulation from a chosen set of the input values; displaying a list of the solution fields generated by a simulation, browsing through this list, and
choosing an arbitrary solution field to visualize; and finally, visualizing the chosen solution field. It is natural to divide these tasks in three separate parts in the data design. Controlling these three parts is being done with a fourth component, designed only, and specifically, for this purpose. These four components in my GUI application are named as follows:

- **Control** - The main control part. It handles the interaction between the operating system and the program. It also controls the rest of the components.

- **MenuTree** - A component that handles the input values for a chosen simulator, allowing the user to change the values before running the simulation.

- **SimResBrowser** - A simple component for browsing a list of the solution fields. It enables the user to choose which solution field to visualize.

- **Visualizer** - A component that visualizes a given solution field.

Before I start describing the components in detail, I would like to point out a set of simple rules I decided to follow in naming the classes:

- **Class XXX is always written in the pure C++**. It is placed at the left hand side of the GUI application model (see figure 3.1).
• Class **GUIxxx** is a generic GUI solution for its C++ counterpart\(^1\). It is always the base class of the **Qtxxx** class, and would typically be placed in the middle of the GUI application model (see figure 3.1 on the facing page).

• Class **Qtxxx** is a specific Qt solution of the generic GUI class **GUIxxx**. It is always a subclass of its generic GUI counterpart, and as such is placed at the right hand side of the GUI application model (see figure 3.1 on the preceding page).

The complete data structure can be seen in figure 3.2.

![Data structure of the GUI application](image)

**Figure 3.2: Data structure of the GUI application**

### 3.2 Control

The core component in the GUI application is the Control component. As its name suggests it controls the execution flow in the GUI application, and it handles the interaction between the GUI application and the operating system. It also launches all the other three parts of the GUI.

\(^1\)The only exception is the SimResBrowser component, as some name inconsistencies appear in this component.
application according to the user's requests. It is divided into the three parts which can be seen at figure 3.2 on the page before:

- The Control class - This class handles the interaction between the GUI application and the operating system. It also stores all the GUI application data, and is thus connected to the other three components of the GUI application.

- The GUCntrol class - An interface actually, not a class, it imposes some guiding lines for a generic GUI solution to follow. All the functions are virtual and overridden by the extension classes, which in this case is only the QtControl class. The Control and GUCntrol class are meant to be kept as they are, providing a platform on which any other GUI solution can be implemented.

- The QtControl class - This class is a GUI widget which you actually see when running the GUI application, as it handles the interaction between the user and the GUI application. The QtControl class is an extension class of the GUCntrol interface, and thus my solution of the Diffpack GUI application. This is the class which could, and should, be easily replaceable with some other class, implementing another GUI solution.

The important thing here is to take notice of the interaction between these three classes. It could be possible to implement the later two as the extensions to the Control class, and for this particular component it would make sense, as it would make sense for all the other components as well, but instead I have chosen to connect these classes with the internal pointers. This is not a two way connection, as the Control class has no pointers to, and thus no knowledge of, the GUCntrol interface and the QtControl class. It basically does not need these classes to execute its functions. The later two classes are, on the other hand, very much dependent of the Control class, and have a frequent interaction with it. This frame of thought is heavily influenced by my mentors' desire to keep the components that interact with Diffpack apart from those that provide a GUI solution, as can be seen on figure 3.1 on page 16.

3.3 MenuTree

The MenuTree component deals with running a chosen simulator. It offers the user the total control of all the parameters involved in running a simulation, and initializes the simulator execution (which, itself, is done by the Control component). The MenuTree component is largely based on the ideas incorporated in the DpMenu.py script, and it follows closely
its data structure build-up. As seen at the left hand side of figure 3.2 on page 17, it comprises of the following classes and interfaces:

- The MenuTree class - This is the main storage class of the component. It holds the information about the smaller logical units as the Submenu and Item objects.

- The GUIMenuTree class - As in the Control component GUIMenuTree is an interface, rather than a class. It imposes the guidelines for any given GUI solution of the MenuTree component.

- The QtMenuTree class - An extension class to the GUIMenuTree interface, it deals with the interaction between the user and the MenuTree component of the GUI application. As all the other classes with Qt prefix, this class can be changed according to a chosen GUI implementation, and modified to fit another developer's need and taste.

- The Submenu class - A class that holds the Item objects (which are equivalent to the parameters in a simulation) and its own Submenu objects. Implemented in the pure C++, it is basically a storage class.

- The GUISubmenu class - Another interface, imposing the guidelines for the possible GUI solutions of the Submenu class. All the functions are virtual and abstract, and thus overridden in a particular GUI solution, in this case the QtSubmenu class.

- The QtSubmenu class - This class contains the GUI widgets defined from the information in the Submenu class, and it also implements the guidelines set by the GUISubmenu interface.

- The Item class - A class that contains all the available information about a random simulation parameter. Again, a storage class, like its less refined counterparts the Submenu and MenuTree classes. These three classes follow closely the data structure generated by the DpMenu.py script.

- The GUIItem class - An interface imposing the functionality to any GUI solution of the Item class.

- The QtItem class - Extending the GUIItem interface, this class carries out the guidelines for a GUI solution of the Item class, set by the GUIItem interface.

Apart from these nine classes, I have also designed another one, called MyQComboBox, but as this is just a Qt widget designed in order to make the QtItem class more effective and user friendly, I choose not to refer to it at this stage, as it basically has no influence on the data structure of the GUI application.
3.4 SimResBrowser

After executing a simulation the user would presumably like to view the solution fields that the simulator has saved in a simres data set. This task is done by the SimResBrowser component, which offers the user a possibility of browsing the results and choosing which solution field to visualize, and thus evaluate. This component is simply structured (as can be seen at the bottom of figure 3.2 on page 17), and it contains the following four parts:

- The SimResContainer class - A storage class containing a set of the solution fields stored in the SimResField class.

- The SimResField class - A simple class that holds the certain information about a random solution field generated by the simulation.

- The GUISimResBrowser class - An interface containing and defining the guidelines for any GUI solution of the SimResBrowser component. All the functions in this interface are virtual and have to be implemented in a specified GUI solution.

- The QtsimResBrowser class - The actual implementation of the GUISimResBrowser interface, this class is controlling the interaction between the user and the SimResBrowser component of the GUI application.

3.5 Visualizer

The last component of the GUI application is the Visualizer component, seen at the top right corner of figure 3.2 on page 17. As its name suggests, it is a component that transforms a given file containing a solution field into a meaningful visualization, providing the user with the ability to evaluate the simulation results. This particular solution implies the use of VTK for visualizing the simulation results. The reasons for choosing VTK have already been explained in Chapter 2.3. This simple component is build from the following classes and interfaces:

- The Visualizer class - A class that builds the visualization pipeline, and stores the visualization data.

- The GUIvisualizer class - Like all the other parts of the GUI application with a GUI prefix, this is an interface imposing the guidelines for all the possible GUI solutions of the Visualizer component.
• The QtVisualizer class - Extending the GUILVisualizer interface, this class actually holds the final parts of the visualization pipeline, combining the Qt and VTK packages to offer the user a full interactivity with the visualized solution field.
Chapter 4

Implementation

Chapter 4 deals with the programming aspect of the thesis, explaining the choices taken during the "production" stages. All the source code explained in this chapter can be found at http://folk.uio.no/zlatkoh/DiffpackGUI/.

4.1 MenuTree

With the GUI application's data structure set up and explained, I would now like to concentrate on the actual implementation of the GUI application, beginning with the MenuTree component. Apart from being the first component that I have developed, it is also a logical starting point, as it would typically be the commencing point in a simulation process.

The first task of the component is loading a file and building the data structure. This is done in the MenuTree class, and in the pure C++. Since there are two file types, .menutree and the input file respectively, I have programmed two different functions, each handling its respective file type (see figure 4.1 on the following page). As the data in both file types is stored as text strings, I have to have a simple parser too. For this purpose I downloaded the stringtok.h file containing a template parser function, so I do not have to write the parsing procedure myself. This function is developed by Phil Edwards at www.jaj.com, and is much easier to use than the standard C function strtok(). Both functions for traversing the respective file types, build a tree-like data structure of the submenus and items, similar to the structure build by the DpMenu.py script. The main difference between these two functions is the amount of information they have to process.

This is of course due to the nature of the two respective file types. As explained in Chapter 2.1, a .menutree file contains all the information
class MenuTree
{
  public:
    bool cmd1_submenu_set;
    Submenu top_level_menu;
    Submenu init_from_cmd1_submenu;

  MenuTree();
  MenuTree(const char *filename, char file_type);
  ~MenuTree();

  Submenu buildFromInput(const char *filename);
  Submenu buildFromMenuTree(const char *filename);
}

Figure 4.1: Code snippet of the header file MenuTree.h

about all the parameters in a simulator, while an input file may contain only some of the parameters and their corresponding pre chosen values. Obviously this leads to much simpler data structures in the case of using an input file, as, for example, the Submenu object init_from_cmd1_submenu (see figure 4.1), would not exist, since there are no command line options in an input file. The stored information in the lower level objects would also be different, especially in the Item objects, as the only variables with no null pointers would be the item command, answer and level (for all the variables see figure 4.3 on page 27).

After this the QtMenuTree class takes over and, based on the information in the MenuTree class, generates a widget with the information about all the available submenus and items. Since many of the parameters can be irrelevant to the user (especially if using a .menutree file), the choice has been made to automatically open the main menu only, and hide all the other submenus. Each menu has its hide and show button to the left of the menu's name, in the QtSubmenu widget. A submenu also has a listing of the items and submenus. This is true for both Submenu and QtSubmenu classes, which hold both Item and Submenu classes, and QtItem and QtSubmenu classes, respectively. See figure 4.2 on the next page for the Submenu class example.

The Item class stores all the information about a simulation parameter (see figure 4.3 on page 27) if available, while its GUI counterpart,
class Submenu
{
    public:
    string name, description;
    int level, noi, nos;
    vector<Item> items;
    vector<Submenu> submenus;

    ...
    /*
     * Declaration of the constructor and destructor
     * functions, as well as some other internal functions
     */
    ...

    void addItem(Item i);
    void addSubmenu(Submenu s);
};

Figure 4.2: Code snippet of the header file Submenu.h
the QItem class, holds and displays only the information relevant to the user. This information is the name of the parameter, the parameter value, possible parameter value choices if available, a button for setting the value back to the value read from a file, and a button to clear the parameter value if available\(^1\). The parameter value is stored in the MyQComboBox class. This is a simple extension to the Qt class QComboBox, which is a GUI widget that combines a button and a pop-up list of the predefined choices. In case there are several possible parameter values, defined in the `.menutree` file (and this can happen only if building a QItem widget from a `.menutree` file), a list of these values is generated, and offered to the user. This list is not editable, and the user can choose only the parameter values from this list. In case no such list exists, the user is free to enter whichever parameter value he wishes\(^2\). The extension of the QComboBox class is necessary in order to set back the parameter value to the value read from the file, which is otherwise impossible with the predefined and uneditable value lists.

When initiating a simulation execution the MenuTree component generates a temporary input file first, before using this file in the actual simulation execution. In case the user loads the parameters and their values from an input file, the temporary file generated by the MenuTree component is very similar to the loaded input file, except for the parameter values which have been changed by the user. This means that in case the user does not make any changes to the parameter values, the MenuTree component, in principle, generates an unnecessary temporary file. If the user has loaded the parameters and their values from a `.menutree` file, the temporary input file generated by the MenuTree component can be considered as a simplified version of the `.menutree` file in question, containing only the parameters' name and value. This temporary input file can also be regarded as a translation from the `.menutree` file type, to the input file type, which can be easily interpreted by a simulator. The temporary file is removed after the simulation execution has finished.\(^3\) Obviously, I am here employing the first method for feeding the input to the simulator, as described in the Chapter 2.1.

The classes written in the pure C++: MenuTree, Submenu and Item, are connected to their Qt counterparts: QtMenuTree, QtSubmenu and QtItem, by the pointers, which are, in turn, inherited from, and imposed

---

\(^1\)This button appears only if the user can enter any parameter value. In other words if the valid parameter values are not predefined.

\(^2\)This may of course cause the simulation execution to abort, so the value entered must be reasonable within the context of Diffpack.

\(^3\)All the actual system calls for these three operations are executed by the Control class, but they are all called from the MenuTree component.
class Item{
    public:
        string command;
        string option;
        string short_command;
        string short_option;
        string help;
        string answer_type;
        string answer;
        int level;

    ...

    /*
    * Declaration of the constructor and destructor
    * functions, as well as some other internal functions
    */
    ...
}

Figure 4.3: Code snippet of the header file Item.h
by, the Qt's generic GUI interface counterparts; GUIMenuTree, GUISub-
menu and GUITem. As already pointed out in Chapter 3, this connection
is one way only. All the GUI classes have a pointer to their C++ counter-
parts, but not vice versa. The complete source code for the MenuTree
component can be seen at [10].

4.2 SimResBrowser

The next logical unit in the simulation execution chain is the SimRes-
Browser component. This component is implemented using only three
classes; SimResContainer, SimResField and QtSimResBrowser, and one
interface; GUISimResBrowser. As in the MenuTree component, the first
task is to load a file, in this case a .CASENAME.simres file (see Chapter 2.1
for a detailed explanation of this file), and build a data structure. This is
much simpler compared to the similar operation in the MenuTree class.
Again I use the parsing template function from the stringtok.h file to
parse the simres file, and store all the necessary information about one
solution field generated from a simulation execution in the SimResField
class (see figure 4.6 on page 31). This information is just a short, sum-
mary information about a solution field, which is required for generating
a visualization of a chosen solution field. The solution field itself is not
loaded in this component, but in the Visualizer component, and only on
demand. When a solution field read from the .CASENAME.simres file is a
vector component in direction, an imaginary solution field is generated
as a SimResField object (see figure 4.5 on page 30). This imaginary field
represents the vector in question, and it contains the information about
the vector's name, time level it is at, and the field type (vector in this
case). This is all the information the GUI application needs to generate
a visualization of this vector field. All the sets of necessary information
about a solution field are stored in the SimResContainer class (see fig-
ure 4.4 on the next page).

Now, that the data is read and stored, the QtSimResBrowser class
takes over and generates the numerous widgets; a list of all the solution
fields, check boxes for choosing which set of solution fields to view (the
solution fields can have the same name, but can be generated at a differ-
time restriction fields.

All the widgets used for the selection and restriction of the listed
solution fields are connected to the same displaying function, which is
quite likely not the most efficient solution, as it regenerates the whole
list of the displayed solution fields from the scratch every time the user

28
class SimResContainer  
{  
    public:  
    vector<SimResField*> fields;  

    ...  
    */  
    Declaration of the constructor and destructor  
    functions, as well as some other internal functions  
    */  
    ...  
}  

Figure 4.4: Code snippet of the header file SimResContainer.h

makes any change that affects the displayed list. This is though the only  
way I made it function properly, and it seemed fast enough at the time,  
so I decided to stick with it, and use it as it is.

Similar to the MenuTree component, the SimResBrowser component  
first generates a VTK file with a system call. This is done using the  
simres2vtk utility, as explained in the Chapter 2.1. The SimResBrowser  
component then starts the Visualizer component with the file in ques-

\[\text{tion.}^{4}\] The functions used in this process can be seen in figures 4.7 on  
page 32, 4.8 on page 33 and 4.9 on page 34. As just explained, the  
QtSimResBrowser class starts the thread with a call to the plotSelection(  
) function, which then calls the appropriate functions in the Control class,  
via the QWidget class.

The SimResContainer class is connected to the QtSimResBrowser class  
by a pointer, which is inherited by, and imposed to, the Qt class from its  
generic GUI counterpart; the GUISimResBrowser interface. The complete  
source code for the SimResBrowser component can be seen at [10].

---

\[\text{As in the MenuTree component the system calls are carried out by the Control  
class, while the launching of the Visualizer component is being done by the QtControl  
class. Both of these calls are initiated by the SimResBrowser component though.}\]
SimResContainer::SimResContainer(const char *filename){
    SimResField *field;
    vector<string> words;

    ...  
    /*  
       Declaration of the other help variables. The calls  
       for reading and parsing the .CASENAME.simres file.  
    */  
    ...

    if(words[3].find("D-vector_1") != string::npos){
        field = new SimResField(words[1], ",",
                                words[2], "vector");//
        fields.push_back(field);
    }
    field = new SimResField(words[1], words[0],
                            words[2], words[3]);

    ...
    /*  
       Handling of the new solution field.  
    */
    ...
}

Figure 4.5: Snippet of the function involved with inserting a solution
field in a SimResContainer, in file SimResContainer.cpp
class SimResField
{
  public:
    string name, number, time, type;
    bool stationary;

    ... /*
    * Declaration of the constructor and destructor
    * functions, as well as some other internal functions
    */
    ...
}

Figure 4.6: Code snippet of the header file SimResField.h

4.3 Visualizer

The third and the final step in any simulation process is typically the
evaluation of the results generated by a simulation execution. The most
intuitive way is by visualizing a chosen solution field. In my GUI application
this is done by the Visualizer component, using the VTK and vtkQt
packages that generate a visualization widget in Qt.

I choose to split the Visualizer component in two parts; one pro-
grammed in a pure C++ and VTK source code, and the other programmed
using the Qt and vtkQt packages. The first part is contained in the Visu-
alizer class, while the second part is in the QtVisualizer class. These two
classes are connected by a pointer in the QtVisualizer class, inherited
from the generic GUI interface GUIVisualizer.

Almost all of the visualization pipeline, with the exception of the
render window and render window interactor objects, is in the Visual-
izer class (see figure 4.10 on page 35). These two classes are part of the
vtkQt package, and as such have to be placed in the QtVisualizer class
(see figure 4.11 on page 36). This is done in order to keep apart the code
which is considered constant for any GUI solution, and the code that
varies with a chosen GUI solution. This may, of course be wrong, as I
assume that not all of the packages combining VTK and some given GUI
solution are designed in the same way, although I doubt this would lead
to any severe problems in the future development. The complete source
void QtSimResBrowser::plotSelection()
{
    generic_VTK_file = true;
    string vtkfile = generateVTKFile();
    my_control->launchVisualizer(vtkfile.c_str());
    my_control->removeFile(vtkfile.c_str());
    generic_VTK_file = false;
    return;
}

string QtSimResBrowser::generateVTKFile()
{
    string command;
    string vtkfile;

    ... /*
     Based on the information about the chosen
     solution field, two strings will be generated.
     One being the system command string used to
     generate a VTK file, and the other the name of
     the VTK file itself.
    */
    ...

    my_control->generateVTKFile(command.c_str());
    return vtkfile;
}

Figure 4.7: Functions from the QtSimResBrowser.cpp file that are involved in visualizing a solution field from the SimResBrowser component

32
void QtControl::launchVisualizer(const char *filename = 0,  
    QtVisualizer *orig_v = 0)  
{
    QString vtk_file;

    ...  
    /*
     * Testing the filename validity.
     */
    ...

    control->generateVisualizer(vtk_file.latin1());
    qtv = new QtVisualizer(control->visualizer, 0, this);

    ...
    /*
     * Additional widget calls to make it
     * visible for the user.
     */
    ...
}

void QtControl::generateVTKFile(const char *s2v_cmd)  
{
    control->generateVTKFile(s2v_cmd);
}

void QtControl::removeFile(const char *filename){
    control->removeFile(filename);
}

Figure 4.8: Functions from the QtControl.cpp file that are involved in visualizing a solution field from the SimResBrowser component
void Control::generateVisualizer(const char *filename){
    visualizer = new Visualizer(filename);
}

void Control::generateVTKFile(const char *s2v_cmd){
    string command;
    command.append(s2v_cmd);
    system(command.c_str());
}

void Control::removeFile(const char *filename){
    string command = "del ";
    command.append(filename);
    system(command.c_str());
}

Figure 4.9: Functions from the Control.cpp file that are involved in visualizing a solution field from the SimResBrowser component

code for the Visualizer component can be seen at [10].

4.4 Control

Controlling these three components is the Control component. As the rest of the GUI application, it consists of one part written in the pure C++ (the Control class), and the other part written using the Qt classes and widgets (the QtControl class). These two parts are connected in the same way as in the other three components, by a pointer in the QtControl class. This pointer is inherited by the QtControl class from its generic GUI counterpart, the GUIControl interface.

The QtControl class is dealing with all the GUI logic and the interaction between the user and the GUI application (see figure 4.13 on page 38), while the execution of the operating system dependant actions is being supervised and controlled by the Control class (see 4.12 on page 37). The Control class also holds the MenuTree, SimResContainer and Visualizer classes used by the other three components. The complete source code for the Control component can be seen at [10].
class Visualizer
{
    public:
    vtkStructuredGrid* sgrid;
    vtkUnstructuredGrid* ugrid;
    vtkStructuredGridReader* sgrid_reader;
    vtkUnstructuredGridReader* ugrid_reader;
    vtkExtractGrid* sgrid_extractor;
    vtkExtractUnstructuredGrid* ugrid_extractor;
    vtkHedgeHog* hedgehog_filter;
    vtkContourFilter* contour_filter;
    vtkOutlineFilter* outline_filter;
    vtkPlane* slice_plane;
    vtkCutter* cutter;
    vtkCamera* camera;
    vtkLight* light;
    vtkActor* actor;
    vtkActor* outline_actor;
    vtkDataSetMapper* surface_mapper;
    vtkPolyDataMapper* hedgehog_mapper;
    vtkPolyDataMapper* contour_mapper;
    vtkPolyDataMapper* outline_mapper;
    vtkPolyDataMapper* slice_mapper;
    vtkProperty* outline_property;
    vtkRenderer* renderer;

    ...

    /*
       Declaration of the constructor and destructor
       functions, as well as some other internal functions
       and variables.
    */
    ...
}

Figure 4.10: Code snippet of the header file Visualizer.h
class QtVisualizer : public QWidget, public GUIVisualizer
{
    Q_OBJECT
    public:
    vtkQtRenderWindow *renWin;
    vtkQtRenderWindowInteractor *rwi;

    ... /*
        Declaration of the constructor and destructor
        functions, as well as some other internal functions
        and variables.
    */
    ...
}

Figure 4.11: Code snippet of the header file QtVisualizer.h
class Control
{
    public:
        MenuTree *menu_tree;
        SimResContainer *src;
        Visualizer *visualizer;

    Control();
    ~Control();

    void generateMenuTree(const char *filename, char file_type);
    void generateSimResContainer(const char *filename);
    void generateVisualizer(const char *filename);
    void runSimulation(const char *sim_file,
                        const char *sim_name,
                        const char *ifile_name,
                        const char *cmdlo,
                        const char *dir);
    void generateInputFile(const char *filename,
                            const char *filedata);
    void generateVTKFile(const char *s2v_cmd);
    void removeFile(const char *filename);
}

Figure 4.12: Code snippet of the header file Control.h
class QtControl : public QWidget, public GUIControl
{
    Q_OBJECT
public:
    QtMenuTree *qtmt;
    QtSimResBrowser *srb;
    QtVisualizer *qtv;

    ...  

    /* 
     * Declaration of the constructor and destructor 
     * functions, as well as some other internal functions 
     * and variables. 
     */
    ...

public slots:
    virtual void launchMenuTree(QtMenuTree *orig_mt = 0);
    virtual void launchSimResBrowser(QtSimResBrowser
                                      *orig_srb = 0);
    virtual void launchVisualizer(const char *filename = 0,
                                  QtVisualizer *orig_v = 0);
    virtual void runSimulation(const char *s_file,
                                const char *s_name,
                                const char* ifile_name,
                                const char *cdmlo,
                                const char *dir);
    virtual void generateInputFile(const char *filename,
                                     const char *filedata);
    virtual void generateVTKFile(const char *s2v_cmd);
    virtual void removeFile(const char *filename);
    virtual void launchHelp();
    virtual void launchAboutHelp();
}
Chapter 5

Using Diffpack GUI

This chapter aims to show the practical use of the Diffpack GUI. In this chapter I demonstrate the basic operations of the GUI application in detail, and briefly enlighten the rest of the options this GUI application has on offer.

5.1 Control

![Control window]

Figure 5.1: Control window

After installing the GUI application (see Chapter 5.5), it can be started by running the DiffpackGUI executable file. The first, and central, window of the GUI application is the Control window (see figure 5.1), which opens immediately after running the aforementioned executable file. It is fairly simple and straight forward. It has four buttons, offering the choice of launching the other three components, as well as exiting the GUI application.
5.2 MenuTree

Once the "Launch simulation menu" button in the Control window has been pressed, the MenuTree window will pop up (see figure 5.3 on the facing page). This is the window where the user can supervise and modify the simulation parameters. Before this window is displayed though, the user has to load a set of parameters from a file. This file can be either an input file, which the user can write in any text editor, or a .menutree file, which is generated by the simulation execution, as explained in Chapter 2.1. The difference between these two file types is in the information contained inside the respective files. An input file can vary in size, which basically means the number of the parameters. So it is very useful in case the user wants to limit the number of the parameters to supervise, and thus remove all the unnecessary parameters. Unlike the .menutree files that offer the user all the information about all the parameters involved in the simulation, the input files provide only the most basic parameter information, like the parameter's name and value. The input files have to have .i or .init as the file extension in order to be read by the GUI application (see figure 5.2). For more information about both file types see Chapter 2.1.

![File dialogue for simulation menu](image)

Figure 5.2: File dialogue for simulation menu

Before executing a simulation the user has to chose which executable file to run, and give a casename (see Chapter 2.1 for the explanation) for the simres data set to be generated. The simulation is run by choosing the "Simulate->With chosen parameters" option from the "File" menu. It is also possible to execute a simulation using another input file, by choosing the option "Simulate->With input file" instead, and then selecting which input file to use. This option can for example be used when the user knows the parameters in an input file are set to his liking, and he
either does not want, or need, to open the file and adjust the parameters.

The user also has the option to store the changed parameters, in an input file format, for the later use, by selecting the option "Save file" from the "File" menu, or by pressing "Ctrl+s" keys. It is also possible to load another input or .menutree file.

Figure 5.3: MenuTree and SimResBrowser windows

5.3 SimResBrowser

Pressing the "Launch simulation browser" button in the Control window will eventually open the SimResBrowser window (see figure 5.3). This is a window where the user can choose which solution field to visualize by
browsing through a simple list of the solution fields. Before this list is generated the user has to select which simulation results file to view (see figure 5.4). This file is usually named according to the case name given by the user in the MenuTree window, with the .simres file extension.

The listing options offered to the user are few and simple. the user can restrict the solution fields in the browser by adjusting the time interval of the shown set of solution fields, as well as hide all the fields with a particular name. The user can only generate a visualization of one solution field at the time, either by highlighting it in the list, and pressing the "Visualize" button, or by double-clicking the desired solution field in the list.

In case the user wants to generate a VTK file containing one solution field it can be done by clicking the "Export to file" button at the bottom of the SimResBrowser window. By doing so, the user will later be able to view this file in the Visualizer component, or even use it in his own, custom visualizing program (or any visualizing program that supports the VTK format, for that matter). If the user wants to compare the results from two simulations, or just work with another set of simulation results, he can open another simres file by choosing the "Open file" option from the "File" menu.

![Figure 5.4: File dialogue for SimResBrowser](image)

5.4 Visualizer

The visualization of a chosen solution field is displayed in the Visualizer window (see figure 5.5 on the facing page). This window is launched
either from the Control window, by pushing the "Launch visualizer" button (see figure 5.1 on page 39), or from the SimResBrowser window, as previously explained. In case the Visualizer window is launched from the Control window, the user has to select which VTK file to visualize. This file can be in either 2D or 3D, and contain scalar and/or vector values. The underlying grid in which the values are contained can be either structured (see [11], page 623) or unstructured (see [11], page 625). In principle this has very little relevance for the user though.

The Visualizer window offers the user a full interaction with the visualized data, such that the user can rotate the visualization, zoom in and out, and use all the other standard VTK options. User can also load another visualization from a VTK file, which is generated beforehand.

A solution field containing only the scalar values is visualized by using the surface rendering (see [11], page 623), which means that only the surface of the solution field is shown. In 2D this is pretty straight forward and obvious, while in 3D this means that the interior of the field is not shown. The coloring of the surface is done according to the scalar values in the solution field, such that the extreme scalar values are represented with the opposing colors. In my GUI application the color lookup table is such that red represents the low extreme values, and blue represents the high extreme values. The choice of these colors is default in VTK. This visualization technique is used for both 2D and 3D scalar fields and can be seen in figure 5.5.

43
Figure 5.6: Visualization of a vector field using the *hedgehog* visualization technique

Figure 5.7: Windows that control the contouring of the scalar field, extracting of a part of a solution field, and cutting a slice plane through a 3D solution field
In case the visualized solution field contains the vector values, the default visualization technique is *hedgехog* (see [11], page 616), which represents the vector direction and magnitude with the oriented scaled lines for each vector. These lines have the origin in the point with which the vector is associated, and are oriented in the direction of the vector components \((i, j)\) for 2D case or \((i, j, k)\) for 3D case. An example of the visualization of the vector field in 2D using the *hedgехog* technique can be seen in figure 5.6 on the preceding page. Scaling the vectors is done through a small widget, launched by selecting the "Scale vectors" option from the *Visualize* menu.

![Figure 5.8: Isolines in a 2D scalar field left, and surface rendering of the same scalar field right](image)

Besides the default visualization techniques, the user can visualize a chosen solution field using the other options offered by the Visualizer component. These include visualization using the *isolines* technique for the 2D solution fields, and *isosurfaces* and *plane slice* visualization techniques for the 3D solution fields. *Isolines* and *isosurfaces* techniques are basically the same technique called *contouring* applied in the different dimensions, that is 2D and 3D respectively. This technique can be applied to the scalar fields such that it creates a line (*isoline*) or surface (*isosurface*) representing a constant scalar value across a scalar field (see [11] pages 158 through 166). If the user chooses the "Contouring (isolines/isosurfaces)" option from the "Visualization" menu a small window will pop up (see figure 5.7 on the facing page). The user can now

45
set the number of *isolines* or *isosurfaces*, and the range of the values to be displayed. An example of the *isolines* can be seen in figure 5.8 on the page before, together with the accompanying surface rendering of the same solution field.

![Select color](image)

*Figure 5.9: Widget for setting the background color*

The user can also display the *outline filter*, which is basically a rectangle (in case of a 2D solution field), or a box (in case of a 3D solution field) around the definition area of the visualization. This is done by checking the "*Show grid outline*" option in the "*Visualize*" menu. It is also possible to change the background color by selecting the "*Change background color*" option from the "*Options*" menu (see figure 5.9). This will also automatically change the color of the *outline filter*, so that it is clearly visible no matter which background color the user picks.

If the user wishes to see the interior of a 3D scalar field in some other way than by visualizing the *isosurfaces* of the 3D scalar field, he can do so by applying the plane slice technique. This is done by selecting the "*3D volume slice*" option from the "*Visualize*" menu. The user now has to define a plane which will cut through the 3D scalar field. This is done by indicating a point in the visualization domain, which is a part of the cut plane, and by specifying the plane's orientation in respect to the point, which is determined by its normal vector, given by the user. This will produce a surface rendering of a 2D cut plane through a 3D scalar field, and thus offering the user a possibility of examining the interior of the 3D scalar field.

The user can also restrict the areas of the solution field domain which
he wants to see by adjusting the domain constrains using the "Extract part of domain" option from the "Visualize" menu. In case of an underlying unstructured grid the user will be able to extract parts of the domain indicating a range of cells to be visualized, using the window shown in figure 5.7 on page 44. The order of these cells is explicitly defined in the VTK file. In case of an underlying structured grid, the user can indicate the range of cells to be extracted in x, y and z dimension. It is also possible to fit the visualized solution field in the visualization window by selecting the "Fit object to window" option from the "Visualize" menu.

5.5 Compiling and running Diffpack GUI

With the use of the GUI application explained, I would now like to explain how to compile and run the GUI application on the Linux platform. Presuming that Diffpack, Qt and VTK (has to be version 4.0) are installed and functioning properly on the system, compiling the GUI application is very easy. All the instructions qmake\(^1\) needs to produce a makefile are

---

\(^1\)Qmake is a Qt utility that generates the makefiles for the Qt applications. It takes care of all the linking and the user never has to change the makefiles himself. Instead he can make changes in the project files (like for example controk.pro) and qmake will apply these changes to the makefiles. More info about qmake can be found at the Trolltech's documentation pages [12]
stored in the file named control.pro. Makefile is then used to generate an executable file. This is all automatically done by the install.sh script, and the program itself can be started by running the DiffpackGUI executable file.
Chapter 6

Conclusion and Reflection

This chapter rounds off the work that has been done, reflecting on the things that could have been done better and/or differently. Some thoughts on the future work are also offered in this chapter, mostly about the possible improvements and the expansion of the GUI application.

6.1 Wrapping up

Although the GUI application is working properly and satisfactorily, following the completion of its design and implementation, I am well aware of its lacks and shortcomings. Some of the choices I have made throughout this two years long process have been rather dissatisfying, while the others have faired far better. I think, for example, I would not change the layout of the GUI application that much even if I had to start again today. Much of the internal logic is also quite decent in my opinion. Most importantly it works without much problems, and seems to be efficient enough to deal with the task it has been put up to. On the other hand I would think a lot about changing the data structure design, as I feel this could be improved to reflect the object oriented principles better than it does right now.

Another great choice is the selection of Qt as the GUI library for the development of the GUI part of my GUI application. After some initial hiccups it turned out to be a pretty intuitive and extremely user friendly package. The choice of VTK was more or less imposed, but that has also worked just as fine. Setting up the connection between these two packages was pretty time consuming though, as some significant changes had to be done in order to get it functioning properly. Something which was, at the time, a rather frustrating, but still, very interesting and useful learning experience.
The highest accolade I could give myself must be for the better understanding of the object oriented approach in the programming of the large applications. I can now clearly see that my thinking in the beginning stages of the project was rather naive, as I obviously lacked the experience of dealing with the "big picture". On top of that I had a chance to learn Qt well, and to slightly improve my ability to program in the C++; a programming language which is very neglected in the courses offered at the university.

6.2 Future work

As I have already pointed out there are many things that can be done better, and there is also a lot of room for the improvement and upgrade of the GUI application.

First of all changing the data structure would be one of my main priorities. Removing the unnecessary pointers connecting the GUI parts of the components with their C++ counterparts, and restructuring the GUI application such that all the GUI classes are the extensions of the equivalent C++ classes. This would lead to a trimmed down data structure, and the interaction between some classes would significantly change, which would presumably make the GUI application more effective. It would also reflect the principles of the object oriented programming far better.

In the second phase of the improvements I would add some user friendly options to the components in the GUI application. Starting with the Control component, where adding a clean up option would be a priority, and then moving on to the other components. In the MenuTree component I would add an automatic start up of the SimResBrowser window upon executing a simulation, so that the user can browse the results straight away, while the Visualizer component would get a major overhaul. Here, I would add many more visualization options, like those that are available in mayavi, for example. The SimResBrowser component would stay pretty much the same as it is right now. Another possible improvement of the MenuTree component could be to generate a simulation menu window using the MenuSystem class and its values directly from a simulator.

The third phase of the improvement would be to make the GUI application work through a network. By this I mean that the user should be able to run the GUI application's GUI components on his desktop, laptop
or PDA\textsuperscript{1}, while all the interaction with Diffpack, like running a simulator, building a visualization pipeline, and similar, could be executed on a remote system.

In case the future developers want to reuse some of the classes from my GUI application in the development of another GUI solution, they would spare the time used on designing the data structure. They would have the core of the application finished and ready to be reused, as well as that they would have available the GUI interfaces for their particular solution of a Diffpack GUI application. They could start developing the widgets and windows straight away, and probably copy much of the internal logic from my particular solution. In that respect, I would like to think that my GUI application is rather flexible, and thus satisfying one of the criteria mentioned in Chapter 2, namely the flexibility of my GUI solution.

As for the other important criteria mentioned in Chapter 2, the portability of my GUI solution, I presume it is also fulfilled. Of course I have no proof of this, except the fact that I have been careful using the platform dependent functions in C++. Qt itself should be platform independent, and all the system calls executed by the GUI application are in my opinion platform independent. I can say this because most of these calls are using the Diffpack execution syntax, which is supposed to be platform independent. The only exception is the removal of the help files, which is also programmed in such a way\textsuperscript{2} to satisfy the portability criteria.

\begin{footnotesize}
\begin{itemize}
\item \textsuperscript{1}Personal Data Assistant - a handheld computer
\item \textsuperscript{2}Files are deleted using the \textit{del} executable file, which works on both Windows and Unix/Linux
\end{itemize}
\end{footnotesize}
Bibliography
