Development of software tool to perform automatic analysis and documentation for large sets of measurement data

Aleksandra Wantuch
Marcin Zwierzyński

This thesis is presented as part of Degree of Master of Science in Electrical Engineering

Blekinge Institute of Technology
June 2016
ABSTRACT

A modern car’s safety and reliability is heavily influenced by the durability of its components and subsystems. These elements’ strength and durability is evaluated with the help of Computer Aided Engineering simulations, test rides on special test tracks and thorough simulations in test rigs. The test rides provide actual information about components’ wear under specific circumstances. However, reducing the testing costs requires limiting the test rides as much as possible. Therefore, CAE and test rigs simulations must be able to provide quality estimation on vehicle’s state under any given circumstances.

These three types of simulations are sources for data which can be analysed and compared. This data primarily is sensor measurements in specific, crucial parts of the car. With the limited input from actual test rides, it is necessary to ensure that on-site simulations are accurate, so that they can be later used to estimate the car’s durability. The data analysis is currently performed using a number of commercial programs as well as manually. This makes the process lengthy and difficult. However, it can be improved by incorporating the most useful analysis tools along with the extra ones which the currently used programs lack into a new program.

These tools were put together to form the scope of this master thesis. Its purpose is to deliver a Graphical User Interface based application, which will incorporate the needs of the engineers in an intuitive matter. The software is created in Python programming language together with PyQt framework for GUI implementation. The application’s main purpose is to read measurement data files, perform specific calculations on them and present the results on screen, both in the form of figures and numerical data. Additional functions of the program include creating a sequence of test events and saving the analysis results as an MS Excel file type report.

The core functionality of the application was successfully implemented. In order to fulfil certain requirements additional libraries for Python were used. The program reached its main goal, which is being a useful, additional tool for measurement data analysis for Volvo’s engineers. However, there is a great amount of extra functionalities that would also be desirable, which opens up a reasonable amount of future work to do.
ACKNOWLEDGEMENTS

This master thesis was a closure to the authors' master level course at Blekinge Institute of Technology and Gdansk University of Technology, which had proven itself to be quite challenging. Here, the authors wish to thank Dr. Josef Ström Bartunk, who always had the time for both college-related support and a kind word; who also later accepted the supervision over this thesis. He is a great teacher as well as a dear friend.

The authors also wish to thank Volvo Car Corporation for the amazing opportunity of performing a master project within the automotive industry. It was a valuable experience as well as a fruitful environment for learning. Warm thanks are due for the thesis' advisors, Erik Täcklind and Robert Höglind and also the team’s manager, Patrik Settergren for their kind welcome and continuous help. They always had the energy and willingness to answer our questions, satisfy our curiosity and let us in on Volvo as much as possible.

Finally, the authors wish to express their deep gratitude to Christer Ericsson, who was the thesis’ supervisor on Volvo’s side. We thank him for his constant and unconditional aid, support and kindness. We highly appreciate the invaluable help, friendly advice and guidance which could always be counted on.

We also wish to thank our mothers for their lifetime support and hard work towards creating amazing opportunities for us, such as this thesis project. Without their aid, we would not have gone this far.
# TABLE OF CONTENTS

1. INTRODUCTION ................................................................................................................ 9  
2. BACKGROUND ................................................................................................................ 10  
3. DESIGN AND IMPLEMENTATION ............................................................................... 11  
   3.1. REQUIREMENTS ...................................................................................................... 11  
      3.1.1. File handling ......................................................................................................... 11  
      3.1.2. Calculations ........................................................................................................... 11  
      3.1.3. Displaying ............................................................................................................. 12  
      3.1.4. Reporting ............................................................................................................... 12  
      3.1.5. Sequence creation ................................................................................................. 12  
      3.1.6. Documentation ...................................................................................................... 13  
   3.2. DESIGN ....................................................................................................................... 14  
      3.2.1. File overview ........................................................................................................ 14  
      3.2.2. Main window ........................................................................................................ 14  
      3.2.3. Analysis settings window ..................................................................................... 17  
      3.2.4. Reporting tool ....................................................................................................... 18  
      3.2.5. Sequence creation tool .......................................................................................... 19  
   3.3. IMPLEMENTATION .................................................................................................. 21  
      3.3.1. Libraries ................................................................................................................ 21  
      3.3.2. File reading ........................................................................................................... 21  
      3.3.3. Main window ........................................................................................................ 22  
      3.3.4. Displaying ............................................................................................................. 26  
      3.3.5. Threads .................................................................................................................. 27  
      3.3.6. Calculations ........................................................................................................... 28  
      3.3.7. Analysis settings ................................................................................................. 32  
      3.3.8. Reporting ............................................................................................................... 33  
      3.3.9. Sequence creation ................................................................................................. 36  
4. CONCLUSIONS ................................................................................................................. 39  
5. FUTURE WORK ................................................................................................................ 40  
REFERENCE LIST ................................................................................................................ 42
1. INTRODUCTION

This master thesis is a project conducted for Volvo Car Corporation in Gothenburg and therefore its specification and scope is determined by the company. The general aim of this thesis is to develop a Graphical User Interface based application using Python 3.4 as its programming language and PyQt 4.8 as the GUI framework. The purpose of the application is to help engineers at Volvo evaluate the durability of a car’s components through test data processing.

The program is required to be able to read large amounts of RPC III files containing multiple channels, both from test track data acquisition and Computer Aided Engineering simulations. For each sensor data in each event it should automatically calculate rainflow histogram, level cross histogram, power spectral density, basic statistics and fatigue damage. It should also enable the user to perform a number of displaying and editing activities, as described further in this paper, while operating on multiple sets of files. All of these functions are to be implemented within an intuitive and clear graphical user interface.

The scope of the work is a collection of the above mentioned requirements specified by Volvo’s engineers. These requirements can be put into 6 main groups:

- file handling – regards the files to be worked on,
- calculations – a group of functions mathematically processing the files’ data,
- displaying – the general idea of incorporating all functionalities into the graphical interface,
- reporting – a possibility to save the results of the analysis performed by the user,
- sequence creation – a tool with the purpose of running the analysis on a set of files put together to form one long measurement (a simulation of many different, consecutive events)
- documentation for the project.

Because the thesis is aimed at developing an application to help Volvo’s engineers in their everyday work, an essential part of the program is its reliability. The quality of the work is monitored throughout the entire design and implementation process, and the program is continuously tested for bugs and errors. Also, for the same reason, it is crucial to focus on the easiness of using the application and adjusting its intuitiveness to its future users.

The first two chapters of this dissertation are the Introduction and the Background. These chapters provide the general description of the thesis and its motivation. The main body of the paper is the next chapter, titled Design and Implementation. This chapter contains three large subchapters: Requirements, Design and Implementation. The Requirements subchapter presents the scope of the work, i.e. the set of all functions which the created application must include. The next subchapter, Design, contains the concept for the implementation of the required functions as well as the looks of all windows included in the application. The last subchapter, Implementation, presents the final look of all parts of the program and describes in detail how each of the parts works. The last two chapters are: Conclusions and Future work. The Conclusions chapter contains a brief summary of the entire thesis and the results obtained. The last chapter, Future work, is a set of all the functions and tools which can be implemented to improve the program, but were outside of this thesis’ scope.
2. BACKGROUND

A very important aspect of developing a car is assuring its safety. Large influence on the car’s safety lies within the quality and durability of its components. Volvo Cars has established requirements for how durable a car must be. These requirements refer both to the entire vehicle, and are broken down into requirements for separate components and sub-systems. All of the specific requirements come together to define how many kilometres a car should be driven without any major failure.

Precise durability testing requires in-depth knowledge of different processes’ influence on vehicle’s components. This knowledge is obtained by conducting test rides while collecting multiple sensors’ data, running CAE simulations and running real-life simulations in test rigs. The tests represent many different events, such as braking, driving on cobble stones etc. [1]. The resulting data is the ground for fatigue calculations and is the mean for checking the particular component’s durability.

The data obtained in the three ways is subjected to cross-comparison. The idea behind it is to obtain the information about a certain event in order to recreate this event later in a test rig at Volvo. These tests simulate this event – or set of events – to generate the wear equivalent to long-term, varying driving. The cross-comparison is currently done partially using RPC Pro software, using other software or manually. The shortcomings of RPC Pro along with the lack of certain functions make the comparison process lengthy and difficult.

To improve the process, it is necessary to design a new software tool which will contain all functions directed toward performing the analysis faster and more effectively. The general idea is to deliver a program dedicated towards the cross analysis of all tests mentioned above, from the most basic, such as file reading and calculations, to more intricate analysis and displaying tools.
3. DESIGN AND IMPLEMENTATION

3.1. REQUIREMENTS

As mentioned in the introduction, the requirements for this thesis can be divided into six main groups.

3.1.1. File handling

All of the measurement data subjected to the analysis is saved as RPC 3 file format. Therefore, the program must be able to read the content of these files by operating on the specified structure of this file format, which is described in detail in chapter 3.3.2. Every file represents an event and contains a number of channels, each representing a sensor measurement, and the program must distinguish between each channel and operate on different file lengths, i.e. the time duration of the event. There should be no limit to the amount of files being used during the analysis and it should also be possible to read more than one file at a time.

The files are provided with extensions: .tim, .rsp, .drv, .des, .rfl and thus the program must be able to read all these files.

3.1.2. Calculations

The analysis of the measurement data bases on a set of mathematical procedures performed on the files loaded into the program. The calculations apply to every channel selected by the user. The procedures include:

- **Level cross counting algorithm** – as specified in ASTM E1049 standard [2]. The input for this algorithm is the measurement data, reference point and the amount of levels, i.e. equally spaced values between data’s maximum and minimum value. The reference point and level amount should be specified by the user. The algorithm counts how many times each of the levels is crossed by the data’s time series plot. The output of the algorithm is a list of level values and a list of crosses count per each level. These two lists are then handed over to displaying functions.

- **Rainflow counting algorithm** – as specified in ASTM E1049 standard [2]. This algorithm’s only input is the measurement data. The algorithm first creates a sequence of reversals from the data using only its peaks and valleys. Then, it calculates how many times every valley-peak range occurs in the data. The output of the algorithm is a list of ranges and a list of cycles per each range. These two lists are then handed over to displaying functions and duty calculating function.

- **Duty calculation** – a measure providing a comparative value between data sets. The input to this function is the rainflow algorithm’s *Ranges* and *Cycles* output lists and a user-definable *Wöhler slope* parameter. With these, for each channel separately the following calculation is performed:

\[
Duty = \sum_{i=0}^{i=n} Ranges(i)^{\text{Wöhler slope}} \cdot Cycles(i),
\]

Where \( n \) is the length of the *Ranges* and *Cycles* lists.

- **Duty normalized** – for the set of selected files and channels, one is to be chosen as the reference for this calculation. Duty normalized is a ratio between the duty of each channel and the reference duty.

- **Power Spectral Density** – for the each file and channel their PSD should be calculated and handed over to displaying functions.
• Statistics – basic statistical information about each data series should be calculated. The information consists of: maximum value, minimum value, maximum range defined as the difference between the two, mean value and root mean square value.

3.1.3. Displaying

The entire project with all its functions must be delivered in the form of a Graphical User Interface application. Its design process must focus on easy and intuitive work with the software regardless of the amount of files currently loaded.

The first requirement is embedding four figures for every set of files: level crossing, rainflow histogram, PSD and time series based on the selected data and the previously conducted calculations. To generate the histogram from the rainflow output lists, appropriate histogram generating function must be used and number of bins passed as a parameter. It must be possible to add files/channels to the display or remove them from the figures. One of the parameters of the analysis is scaling, which enables the user to set a multiplication factor for selected channel(s) values.

It is required to implement standard figure analysis tools, such as zooming, panning, adding data flags and grid toggling. For the time series plot, it should also be possible to switch between overlaying plots and subplots. Additionally, the program must enable changing axes ranges.

Another vital function is displaying 'summed' level cross plot and rainflow histogram. It is defined as additional line in a figure which is obtained by adding together y-values of each overlaying line for its corresponding x-values. The resulting summed line is meant to help view these figures while there are many channels selected. Similar to this, for rainflow histogram there should be a possibility to add 'accumulated' rainflow, which is obtained by starting at the last point of the figure and moving backwards on the x-axis while adding together corresponding y-values. This is performed per line present on the plot. The resulting lines are easier to view than the regular rainflow histogram, and it should be possible to generate a 'summed' line from them as well.

3.1.4. Reporting

A possibility to save the performed data analysis is a necessary part of the previously described work since the conclusions drawn from it are later used for durability testing. The reporting tool should save the current state of the analysis, i.e. the figures, files and channels loaded and/or selected, previously described statistics for all loaded files and duty calculations. The user should be able to specify the reference duty files and channels for normalized duty calculation. For every file set, an MS Excel document containing all of the above should be created by the application upon user’s demand.

3.1.5. Sequence creation

The idea behind this tool is to create a simulation of a set of events within the program which will estimate their impact on the car’s components. The estimation is based on figures, statistics and duty calculations. This information is a base for the user to select the most appropriate sequence to be later recreated in a test rig. This sequence is a set of files with a repeat number for each of them. The repeating files are put together to form a long series of varying events which is meant to simulate an actual ride or a long test ride.
The sequence creation tool should therefore enable file and repetition selection. However, because the resulting data would be extremely long, time series plot and PSD are not supposed to be provided for the sequences. Calculating level crosses and rainflow from such data would take an indefinite amount of time. Instead, these two algorithms will be run for every file separately, and then added together for every repeat for all files. This is done in the same manner as ‘summed’ level cross and rainflow histograms.

It should be possible to generate multiple sequences and compare their plots as overlaying, as well as their statistics. Also, the program is not meant to be able to save the designed sequence as a new RPC 3 file. An additional assumption is that all files forming the sequence contain the same set of channels.

3.1.6. Documentation

Because the thesis is a software development work for an industrial company, certain documentation is required. This documentation consists of three forms:

- Software development plan – a limited version of plans provided by professional software developing companies. It should contain all basic information which defines the project and its requirements, as well as time schedule for the project’s milestones
- Operator’s manual – a document with the purpose of introducing a new user to the application. It should include the description of every functionality of the program and the way the analysis should be performed
- Maintenance manual – a larger document describing in detail all functions of the source code. This also includes adding appropriate comments to the source code itself. The purpose of both of these is to provide guidelines for Volvo’s engineers if they wanted to make changes to the program or fix any future errors.
3.2. DESIGN

The design process for this thesis project combines meeting the requirements with incorporating them into an intuitive GUI application. The program should be easy to use, the analysis process not difficult to learn and it should be reliable.

Data analysis is to take place in the main window of the application. There, the user should have an option to open files, select channels for display or use figure analysis tools and the results of their work should be presented. The main window shall also enable calling other required tools, such as the analysis settings tool, reporting tool and sequence creation tool.

Analysis settings tool should provide a method to display a given set of files and enable changing such parameters as: scaling, level cross reference value, level cross levels count, rainflow histogram bin count and Wöhler slope. It should be possible to change parameters for each channel in every file separately.

3.2.1. File overview

RPC 3 files are binary files with a specific structure [3]. The files have two main parts: a header and data. The header is made up of descriptors for the file, such as the number of channels, names of channels or sampling period. These descriptors are 128 bits each and come in blocks of four. This means that the header’s size is a multiple of 512, and knowing the amount of header’s descriptors it is possible to determine the binary position for the first bit of data.

Every channel’s data is divided into groups, and groups into frames. Group and frame size can be obtained from the file’s header. In the data section of the file, the groups of binary data come one after another. Thus, loading data from only one channel requires reading its each consecutive group and determining the binary distance between them.

Therefore, the method for file loading will take form of a standard file opening Windows prompt and will be designed to accept multiple files at once. To avoid excessive memory consumption, only the headers will be loaded into the memory, and the data will only be accessed upon request.

3.2.2. Main window

The main window is a part of the application where data analysis takes place. The main idea behind the analysis is to compare the figures for different sets of events and duty they generate. To enable handling multiple set of files, the main window will contain tabs. Every tab will contain different file setup, and the user will be able to perform the analysis separately for each tab, switch between them for comparison or generate a report for any file set.

The main window will contain a tool for selecting files and channels, which data should be put forward to calculations and display. There should be a toolbar for the figure analysis tools described in 3.1.3., a list of files and the statistics table. The main body of the window will contain the four figures (3.1.3.).

Also, from this window, there will be a possibility to call other tools, i.e. analysis settings, reporting tool and sequence creation tool.

The design for the main window is presented in figures 3.1., 3.2 and 3.3.
Figure 3.1 presents the window in which analysing the data takes place. The outermost box with blue background is a QLayout provided by PyQt. The type of the layout defines how its child widgets are positioned on screen. In this case, it is a QHBoxLayout, which means it is a horizontal layout. This type of layout places the widgets in one line to the right side. By default, they are all given the same amount of space in the QBoxLayout. In this case, the main layout contains only one widget – a horizontal QSplitter. Splitter is a type of container, which accepts two elements – widgets or layouts – and puts a dragging area between them. This enables resizing the contents of the splitter in relation to each other.

Inside the main splitter, on the left side there is a QTabWidget and a QVBoxLayout. QTabWidget is a tab tool, adjusted for the application to contain four canvases (time series, level crossing, PSD and rainflow) and have an option to add new tabs or remove them. Each tab is a place for a different analysis. The QVBoxLayout, similarly to the main box, places its child widgets one after another – but in this case it is placing them one under another. Inside of this layout there are two groups of widgets: Navigation and Toolbar. Navigation is a set of five buttons: four directional and a confirmation button (3.3.3) used for moving the selected cells around the channel selection table. Toolbar contains the plot analysis tools mentioned in 3.1.3, for example zooming or grid toggling. The tools are represented by checkable buttons. Both Navigation and Toolbar’s content is placed inside of QGroupBox widgets, which are a grey frame around the content with its name at the top.

On the right side of the main splitter, there is another, vertical, splitter. This splitter contains two tab widgets as its child elements which are shown in figures 3.2 and 3.3 and described below.
The first widget inside of the vertical splitter is a QTabWidget, containing two tabs: Channel selection and Legend. The Channel selection tab’s layout is a QVBoxLayout, with one child – a QTableWidget. The purpose of this table is to enable file and channel selection for plotting, or deselecting them to remove them from graphs. Each of the main tab widget’s tabs has a different channel selection table.

The Legend tab contains an inner QTabWidget with tab’s orientation set to vertical. There are four tabs here, corresponding to the four canvases in the main tab widget of the window. The legends inside each tab are related to their figures. In every of these tabs, there is a QScrollArea placed inside a QVBoxLayout. The reason for using a scroll area for the legend is that a QLayout does not provide a scrolling action for its child widgets. Instead, they become squeezed when their combined size exceeds the size of the layout. This issue is solved by using a scrolling widget with a vertical layout inside of it. As an effect, there is no size limit for the inner layout and thus its children widgets can take up as much space as needed. The legend itself is created by combining QLabels, which are text areas with the file names, and a QGridLayout under each label. A QGridLayout places its children widgets in a grid, and upon using a QLayout.addWidget() method, for this layout the method requires additional positional variables. Inside the grids, there are a number of QPushButton and QLabel pairs corresponding to the selected channels, with the button presenting the selection colour of the channel and the label showing its name.

The second widget of the vertical splitter is one more QTabWidget with two tabs: File list and Statistics. The File list tab contains one widget – a QListWidget, which contains a list of all files loaded into the currently open tab. From this list, it is possible to choose a file to be removed from the current tab’s data set (and, consequently, from the list and channel selection table). The Statistics tab contains a single QTableWidget, which purpose is to present the basics statistics of the selected channels (3.1.2) and the calculated duty values.
3.2.3. Analysis settings window

Because of the amount of analysis parameters and the necessity to enable the input for each of them for every channel in every file, it is required to embed this function in a separate window. This window can also be used to load new files, and set the analysis parameters before this file set is placed into the main window and handed over to calculations and display.

Thus, the screen must contain a mean for file loading, channel selection, a list of loaded files and the inputs for the parameters. The inputs will take a form of edit boxes, and selecting which channel does the input apply to will be done by checkboxes.

The design for the analysis settings window is presented in figure 3.4.

![Fig. 3.4. The concept of the application’s analysis settings window](image)

The outermost layout is a vertical box layout containing two QHBoxLayouts. The leftmost element inside of the first QHBoxLayout is a ‘toolbar’ group of buttons. This includes a button for file opening along with a set of navigation buttons. The middle element is a QTableWidget, which is identical to the channel selection table from the main window. The rightmost element is a QTreeWidget, which purpose is to present all loaded files, the channels they contain as well as basic information about the channels’ data, such as their unit, maximum and minimum values.

The second horizontal layout contains two elements: another QTreeWidget and a QGridLayout. The QTreeWidget contains a list of all channels selected in the QTableWidget above. It also presents the selection colour for each channel as well as their analysis parameters. Every row in this tree contains a QCheckBox, which determines which channels’ parameters are to be changed. The grid layout contains five QLabel-QTextEdit pairs for the five analysis parameters (value scaling, Wöhler slope, level crossing reference value, level crossing level amount and rainflow histogram’s bin amount). The QLabels serve as a description to the corresponding parameters, and the QLineEdits’ purpose is to enable value input for the parameters.
3.2.4. Reporting tool

The reporting tool must enable selecting the file set (tab) from which the report is to be generated and editing options. The options include selecting the reference file and channel for normalized duty calculations and disabling individual parts of the report itself, in case they are not all necessary.

Thus, the tool will take the form of a dialog window with a tab selection part and a section for the report options and a confirm button. Upon confirmation, a separate report for each analysis (tab) should be created, in the form of an MS Excel document.

The design for the reporting tool dialog window is presented in figure 3.5.

---

![Diagram of the reporting tool dialog window](image)

**Fig. 3.5. The concept of the application’s reporting tool dialog window**

The main layout of the window, a `QHBoxLayout`, contains two `QVBoxLayouts`. The first vertical box has a single child widget – a `QScrollArea`. The area provides a scrolling tool for a list of `QCheckBoxes`. Checkboxes 1-1, 1-2 etc. each represent a tab from the main window. By selecting tabs here, the user chooses which analyses are going to have a report prepared. Thanks to the scroll area, there is no limit to the number of tabs which can be added to its inner vertical layout as there is no size limit for the layout.

The second `QVBoxLayout` of the main layout contains options for the reports themselves. The topmost widget there is a `QComboBox` (DROP LIST 1 in fig. 3.5). Whenever a tab inside of the `QScrollArea` gets checked, its name is added to this drop list. The reporting options below refer to a currently selected tab from this drop list and are updated whenever the selection changes.

The first set of options is placed inside a horizontal layout below the first drop list. It contains two elements: a vertical layout and a `QTreeWidget`. The vertical layout arranges a list of checkboxes 2-1–2-5. These checkboxes represent: time series figure, level crossing histogram, PSD figure, rainflow histogram and statistics table. By deselecting one of these boxes, the user can prevent these components from being saved into the
reports in case they are not needed. The advantage from this is that generating the report itself takes less time and only includes the useful information. The tree widget displays the files loaded into the selected tab, along with the channels which had been selected for displaying within the main tabs.

The second set of options is placed inside a QGridLayout. There are two options to be edited here, which are described by two QLabels: reference duty and report name. The reference duty can be selected using two QComboBoxes (DROP LIST 2 and DROP LIST 3 in fig. 3.5) – first for selecting a file and second for selecting a channel. The duty calculated for the chosen channel in the chosen file will later be used to calculate reference duty for the set of files from the selected tab. The report name can be defined using a QLineEdit (TEXT EDIT). If no name is defined, the report is named automatically after the corresponding tab’s name.

The final widget of the window is a QPushButton. It is a confirmation button, which begins the report generation upon being clicked.

3.2.5. Sequence creation tool

This tool’s core function is loading files and giving the possibility to set a repeat count for each of these files. This will be included in a dialog window along with a list of files and channels and a confirmation button.

Upon confirmation, the appropriate calculating functions will be called. They will be slightly altered versions of the previously used algorithms, set up for their following summing procedure. In order to do the counting for the same set of levels, which means eliminating the need for interpolation, it is necessary to run the level cross counting algorithm for globally determined levels, i.e. levels obtained by finding the largest and smallest value among all channels in all files, and dividing their difference by 500. This number marks the resolution of the resulting plot and has been agreed upon with Volvo. The resulting plots and sequence will be added to the main window to a separate tab.

Another aspect to designing this tool is ensuring that the main window is compatible with the sequence tab. The fact that this tab’s content is different from the others forces had to be taken into consideration when designing the tab switching method, plotting method and other functions common to all tabs.

The design for the sequence creation window is presented in figure 3.6.

![Sequence creator](image)

**Fig. 3.6. The concept of the application’s sequence creation tool dialog window**
The main (horizontal) layout contains two inner vertical layouts. The first of these two arranges two widgets: a QPushButton and a QTableWidget. The button (BUTTON 1) calls a file opening method, which means the file opening prompt. The files selected using the prompt are added as rows to the table widget below. Within this table, there is an input tool for defining the repeat count for each file.

The second vertical layout contains a QTreeWidget and a QPushButton. The tree widget contains all loaded files as expandable top level QTreeWidgetItems, with sub-items representing channels which each of the files includes. Inside the tree, disabling selected channels is enabled. The last button (BUTTON 2) is a confirmation button, which, upon clicking, triggers the sequence generation procedure: file reading, calculations and adding the sequence to the main window.
3.3. IMPLEMENTATION

3.3.1. Libraries

Python has a wide variety of methods. However, some of the features required in the application exceed its capabilities. Therefore in order to implement these features installing additional libraries is necessary.

As mentioned in the introduction, all graphical elements in the application are implemented using PyQt 4.8 framework, primarily its QtGui and QtCore modules. These elements are called widgets. PyQt framework provides a great amount of widgets with multiple methods and signals, prepared for different activities. In addition, this framework provides a threading class which is used in the application. All components of the PyQt framework are presented and described in its online documentation [4].

As was described in 3.1.3, one of the functions of the application is displaying the required figures. To do this, matplotlib library is used. It is a commonly used plotting library, which also enables figure customization. Every figure in the application is implemented as matplotlib.pyplot.figure(). In order to embed the figures in the PyQt's graphical interface, which only accepts widgets, matplotlib provides canvases. It is a canvas that is added in to a graphical layout, and the figure is drawn on it. For a lower memory consumption and shorter plotting time matplotlib.backends.backend_qt4agg backend is used. It provides FigureCanvasQTAgg canvas widget. Also from this backend, NavigationToolbar2QT is used for performing certain figure analysis tools, described in 3.3.2.

More advanced mathematical operations are implemented with the help of NumPy library. It is a widely used python library containing a great number of methods and classes, for example: linspace, histogram, interp, fft and more basic: floor, ceil, min, max, mean, power or sum. NumPy also provides a numpy.array() array class and array editing methods (e.g. numpy.append(), numpy.delete()), which are used in the application.

To fulfil the requirements for the reporting tool, it is necessary to be able to save data to an MS Excel file format. Because Python does not provide this functionality on its own, openpyxl library was used. From openpyxl two classes were imported: Workbook and drawing.image.Image. Workbook class provides an Excel type file to add spread sheets to using workbook.spreadsheet(), and save later as a final report. The Image class enables adding image type files into the workbook.

3.3.2. File reading

Python 3.4 has a number of built-in methods for operating on binary files. As a start, file.open() method is used in order to access all the data contained in the RPC files. In every RPC file, the first three header records are fixed and they define the format in which the data is stored (e.g. ASCII, binary big endian/little endian), the number of header blocks (4 records each) and the overall number of records in the header. This information is obtained using file.read() method, which takes the amount of bytes to read as its variable. Knowing the last two values, it is possible to determine the overall length of the header and read the remaining records. The records most commonly used in the application tell the amount of channels in the file, the sampling period, every channel’s description, the number of frames, points (bytes) per frame, points per group and every channel’s units.

After reading the remaining header records, since they all are stored in groups of four, if the total number of records is not divisible by four it is required to move the reading cursor to the end of the header manually. This makes the next file.read() method start with the beginning of the data, and is done using file.seek() method. Alternatively, the empty headers can be read anyway as they are simply series of zeros. This way, the cursor is set at the end of the header without the need for file.seek().

As mentioned in 3.2.1, every channel’s data is divided into groups, which size can be determined using information from the header. Further division into frames has no influence on data reading. With the cursor at the end of the header, all content remaining is the data. However, in most cases the user will not wish to load all
channels in a file. Then, it is necessary to move the cursor over the data groups belonging to the not selected channels. This is done by combining the PTS_PER_GROUP header record and the number of sought channel, thus obtaining the variable for file.seek() method. The method for moving the cursor and reading takes all selected channels into consideration, so it only needs to go through the data once.

In the application, this whole procedure is preceded by selecting the file to load. In certain parts of the program (3.3.2, 3.3.6, 3.3.8) there is such an option, triggered either by pressing a button, pressing Ctrl+O or selecting File->Open file in the menu bar. These actions trigger a standard Windows file opening prompt, shown in figure 3.7, where the user can select one or more files to load into the program. The paths to the selected files are stored in a variable and are later needed whenever data reading is required.

![Fig. 3.7. Windows file opening prompt used in the application](image)

The entire file reading procedure is incorporated into a class, containing different methods such as header reading or data loading. In order to reduce memory consumption, a new object of this class is created upon every file reading inquiry, the data it reads saved to a variable and after the data is processed, it is removed from the program along with the RPC reading class.

3.3.3. Main window

This is the container for most activities in the software and it is the place in which the data analysis is performed. This window is shown maximised upon the start of the program. The finished implementation of it is shown in figure 3.8. The GUI elements were implemented using PyQt4 framework.
The main body of the window is a `QTabWidget`. The last tab is set up to act as an ‘add tab’ button, and the other tabs are closable. The right-top corner of the window contains the area for channel selection as well as a legend.

The tab widget contains a `QGridLayout` with four canvases. The canvases are provided by `matplotlib` library’s `FigureCanvas` widget, onto which the figures are added. The units on the time series canvas are added as an `AnchoredOffsetBox`, which makes their colours, size and direction customizable. A mouse click on any of the canvases is handled by a mouse press event. For all canvases, a double click removes the clicked canvas from the main layout, creates a new window and puts the canvas into it. This enables the user to see the figures up close, and also generate multiple windows this way and place them around the screen at will. Also for all canvases, right mouse button click triggers a `QMenu`, taking the form of a drop list widget, appearing at the mouse cursor’s location. For the time series and PSD figures, the only option in the menu is to change axes range. Selecting it opens a dialog window (fig. 3.9.) with four `QLineEdit`s – one for each axes end. For level cross and rainflow figures there is also an option to add a ‘summed’ line. Furthermore, for the rainflow histogram there is yet another action – switching to the ‘accumulated’ lines.

The main tab widget plays a crucial role in how the analysis is performed. As was mentioned, every tab has its own set of files, and the analyses on different tabs are independent. This is achieved through a
method which is triggers by `QTabWidget's tabChanged()` signal. If the user switches tabs, this method is called along with the index of the selected tab. If the index points at the last tab, it means a new tab is requested. Then, a new tab is added to the widget and to it a new layout with new canvases, a new channel selection table, file list, statistics table and legend. Before continuing, the user has an option to import a set of files from another tab or start with a blank file list. This selection appears in the place of the channel selection table and is shown in figure 3.10. Importing a set of files fills the file list with their names, sets the appropriate shape of the channel selection table and also imports the analysis’ parameters from the chosen tab. All widgets created along with the new tab are appended to their respective global lists, which store these widgets. These lists are necessary for switching between the tabs. If the index points at a tab other than the last, currently active widgets from these lists are hidden and in their place, the widgets belonging to the chosen tab shown. Clicking the red corner button of any tab sends a closing request. Upon it, widgets from the corresponding index are removed from the lists and destroyed and then the tab is closed.

![Fig. 3.10. Option to import a set of files from another tab](image)

The channel selection area includes a `QTableWidget` and navigation buttons. In the `QTableWidget`, the columns represent the loaded files and the rows represent channels for each file. To enable method calling after clicking the table in specified ways, an `EventFilter` was installed on this widget. The event filter catches a mouse button click on the table, and selects corresponding channels (left mouse button) or de-selects them (right mouse button). Selecting channels causes a call to file reading, calculating and displaying functions, and de-selecting them calls methods for removing lines from the figures. The legend and statistics table are also affected. Double-clicking a cell in the file selection table triggers a colour change dialog window, presented in figure 3.11. This enables the user to choose a colour they wish for the selected channel (and, consequently, its line in the figures, its legend index and statistics table index).

![Fig. 3.11. Colour selection dialog window](image)
The navigation buttons enable moving the channel selection in the indicated direction. After clicking one of the directional buttons, the coloured cells shift accordingly (within the boundaries of the table) and the ‘check’ button changes colour to light red, indicating the need for confirming desired selection position. Clicking the confirmation button then calls methods which perform similarly to simply de-selecting all of the channels and selecting new ones.

Fig. 3.12. Application’s main window’s functions

The legends were removed from the figures and placed on the side of the window to prevent them from covering the plots or squeezing them, which would result in the plots being hard to work with. Also, since in all of the four figures the lines colours represent the same corresponding channels, one legend for all is sufficient. The legend can be seen in figure 3.12. The coloured boxes in the legends are QPushButton set to checkable, and pressing one of them results in the corresponding figure’s line to become hidden. The figure is selected by switching vertical tabs with the figures’ names.

In the lower right corner there are: a toolbar and a tab widget allowing the user to switch between the file list and the statistics table.

The toolbar provides standard figure analysis tools, described in 3.1.3. All buttons are set to checkable, with the tool active when its button is checked. Only one button at a time can be checked, and selecting a new one unchecks the active one. Zooming, panning, returning to normal view and grid make use of matplotlib’s built-in NavigationToolbar2QT, which can be added to any canvas. Zooming tool, which uses NavigationToolbar2QT.zoom() method, has an effect for both left and right mouse button click. When it is selected, left-clicking on a figure triggers a marker box, which is the zoomed in area after button release. The right button click works in the same way, except it zooms out. Panning, which uses NavigationToolbar2QT.pan() method, also has two different functions: left click enables moving the lines around the figure and right mouse button squeezes or stretches the plot depending on mouse movement direction. Reset view (NavigationToolbar2QT.home()) , toggling grid (NavigationToolbar2QT.grid()) and adding data points become deactivated instantly after clicking the canvas to perform the function. The last tool enables switching between overlaying time series plots and subplots view. Upon activating this function, its button becomes checked as long as this mode is selected on the currently active tab. The state of this button is adjusted upon switching tabs in order to match the currently displayed time series mode. Other tools can be
used regardless of the state of this button. In order to prevent subplots from becoming too small when in high numbers, a QScrollArea is added to the main tabs layout, and the time series canvas placed into the scroll area. This alone would be insufficient, as matplotlib by default does not enlarge its canvas automatically, nor does it resize subplots to take the maximum space available if the canvas was enlarged manually. The solution to this problem is creating the subplots using `plt.subplots()` method and passing a desired plot size through the `figsize=(height, width)` parameter (dimensions are given in inches). The effect is presented in figure 3.13.

![Fig. 3.13. Time series subplots embedded in a QScrollArea](image)

The tab widget’s first tab contains a QListWidget with currently loaded file names. A method connected to this list catches a right mouse button click on the list’s element, which executes a QMenu drop list. This drop list contains one position – removing a file – and upon clicking it, the file name is removed from the list and along with the file’s path from the global path list.

The second tab contains a QTableWidget presenting the selected channels basic statistics, as described in 3.1.2., as well as duty calculations. The vertical header of the table contains a QIcon in each row, created from QPixmaps filled with the corresponding channel’s colour. Upon hovering over any of the icons, a ToolTip containing the file’s and channel’s names appears. To trigger normalized duty calculations, the reference channel must be selected. This is done by double clicking any of the cells containing calculated duty, which sets the selected cell’s background to light green by using `QTableWidgetItem.setBackground(QtGui.QColor(180,220,180))`, where the numbers are RGB values. Then, the last column is filled with the calculated normalized duties. Double clicking another cell containing a duty value automatically deselects the previous one, highlights the new one, recalculates the normalized duty and updates the values in the table.

3.3.4. Displaying

The methods described in this section are triggered upon selecting or deselecting channels from the main window’s channel selection table. To catch mouse button presses and releases on the table, an EventFilter is installed into the table using `QTableWidget.installEventFilter()`. After catching an event, this method creates a list of numbers of files and a nested list of numbers of selected channels per each file. After marking the cells in the table is complete (the mouse button is released), these two lists along with the list of colours of the selected cells are passed to the displaying methods. The colours of the selection are pre-defined for the first twenty selections, and after that a method which randomly generates the RGB values produces new colours.

The method responsible for adding new line to the figures works in two ways, depending to the view mode of the time series. If default, overlaying plots are selected, the method only adds an overlaying subplot to
the figures’ axes. If on the currently active tab the subplots function is selected, the time series graph is discarded and recreated with a larger canvas, meant to fit the additional subplots. In both cases, once the figures and axes are prepared, the plotting itself takes place in threads (3.3.5).

Removing lines from the figures requires more steps, primarily because they all take place in the same method instead of threads. The first step is to update the legend. This is achieved by looping through it and finding the buttons and labels corresponding to the de-selected channels. Additionally, if, for a given file, all channels become de-selected, this file’s label should also be removed from the legend. Then, selected lines are removed from the figures. If the subplot display mode for the time series is selected, a subplot instead of a line is removed. Another step is to update the AnchoredOffsetBox, containing all units by the time series figure (only if the overlaying time series is active). Finally, for all figures, axes are rescaled and canvases redrawn.

3.3.5. Threads

The application contains a number of functions which take noticeable amount of time. Normally, the application freezes – is not responding for the operating system – for as long as such a function is running. Such behaviour is highly undesirable, as it gives the impression of faulty and unprofessional software. Therefore, every function required in the program which might fall to this case was places in a separate thread, which enables it to run in parallel with the program, eliminating all the issues mentioned above [5].

The application contains six threads beside the main one: one for each figure, one for statistics calculation and one for generating a report. The displaying method passes files and channels to plot and colours to use to the first four threads. There, every figure is prepared separately, and calculations for them (level crossing, PSD and rainfall) are performed simultaneously. This not only prevents the program from not responding, but also shortens the overall plotting time, whereas if this whole procedure took place in the main thread then every figure would be prepared in turns.

The threads are implemented using PyQt’s QThread class. Every thread class contains an __init__() method, where the class is initiated; a run() method, where the thread’s function is programmed, and a run_it() method. The run() function is the way the threads are referred to by the rest of the program: this method accepts variables from the main thread (e.g. selected files, selected channels per files, colours, parameters), passes them to the whole class and contains start(), which automatically launches the thread.

For every thread, in the run() method there is a for loop over each file and over each channel selected. Every thread reads corresponding RPC files on its own, using the file path list it was passed from the main thread. Because the threads only read the files without changing them, there is no conflict in sharing these resources. The plotting functions prepare another line to plot during consecutive iterations.

The time series thread is the most straightforward, as its main purpose is to load the data from the files and plot it against a time list. If the user customized the scaling parameter, here the data is multiplied by the scaling value. The time list is obtained from the length of the plotted data and the DELTA_T (sampling period) header record. However, this thread does more than that. Firstly, it prepares the AnchoredOffsetBox for the figure, with all the units coloured according to their corresponding lines. This also means checking if there already is such a box (when something is plotted already and the user selected additional channels) so that the figure does not end up with two boxes on top of each other. Additionally, the plotting itself is different depending on the displaying mode (overlaying plots or separate subplots). The lines in both cases are labelled as ‘<filename>, <channel name>’. Finally, after plotting is complete, a self.emit() method is used to trigger legend generation while passing the colour list, file names list and channels names list.

The level crossing thread itself is shorter compared to the time series thread, but most of the work here is done by code in another file named Mathematics. This thread only reads the data and passes it to Mathematics.lvl() method, which performs the level crossing counting. The lvl() method besides the
measurement data also requires the reference level and level amount as its parameters. The calculation returns a nested list containing: the ranges list, which is the X axis of the figure, and crosses count per each range list, which is the Y axis. The lines here are labelled just like in the time series figure. The Y scale is set to logarithmic.

The PSD thread’s calculations are performed within the thread. After reading the data, `numpy.fft()` method from `numpy` library is used to obtain the Fourier transform of the series. The power spectrum is obtained as described in [6], plotted and the Y scale set to logarithmic.

The rainflow thread, similarly to level cross thread, calls a `rainflow()` method from the `Mathematics` file after reading data. This method returns a nested list of ranges list and cycles per each range list. Rainflow figure is supposed to take the form of a histogram. Therefore, these two lists are used as variables in `numpy.histogram()` method, which generates histogram’s bins’ edges and the values corresponding to each bin. To obtain the bin centre values, against which the Y values will be plotted, an average between every two bin edges is calculated. Then, these two are added onto the figure with a logarithmic scale. Additionally, a list of the ranges and cycles is passed to a `self.emit()` method, which triggers duty calculations. This thread includes two additional emitted signals: at the end of every iteration, increasing the value of the progress bar is triggered and at the end of the entire thread, the termination is signalled. The main thread then calls `canvas.draw()` on each of the four canvases, and if summed or accumulated level cross/rainflow had been selected, calls the responsible method to update these lines.

The final thread is the statistics thread. This thread runs in parallel with the previous ones and fills the statistics table with the required values. These calculations require a separate thread as obtaining minimum, maximum, mean and RMS value from long data series take noticeable amount of time.

The reporting thread is described in section 3.3.8.

3.3.6. Calculations

This section is implemented in the `Mathematics` file mentioned in the previous section. It contains the level cross counting algorithm, rainflow counting algorithm, duty calculations, summed and accumulated plot calculations.

The level cross counting method was briefly described in 3.1.2. In the application, this procedure is triggered by the level cross thread, which uses the results for plotting the level crossings. This algorithm is presented in figure 3.14 in the form of a flowchart, where `data` is the channel data for which the level cross counting is performed.
The flowchart branches to the sides because the crosses are treated differently for negative and non-negative data. On the non-negative side, only crosses from the ascending slopes are counted, and for the negative side – only from the descending slopes. This also results in a slightly different method for searching the nearest level for counting. In the application, in early stages of the implementation, this search was at first done iteratively over every point in the data and then the point was searched for in the level list. This resulted in a very slow calculation. What turned out to be a much better solution is bisection. This method uses a sorted list, such as the levels list, and returns an index for a given value’s position in the list. For the non-negative data, `bisect_left()` was used, and `bisect_right()` for the negative data. Both of these methods return the position of a data point in the levels list, with the difference of on which side of the searched level the position is. Once the level corresponding to the given data point is determined, a `while` loop is entered, in which all levels between the found one and the previous data point – providing they are on the correct slope direction - are added one cross. The algorithm stops once it reaches the end of data.

The rainflow algorithm was also described in 3.1.2. It provides information useful for damage fatigue analysis [7]. In the application, this procedure is triggered by the rainflow thread, which uses the results for generating and plotting the rainflow histogram. This algorithm is presented in figure 3.15 in the form of a flowchart, where data is the channel data for which the rainflow algorithm is performed.
Rainflow algorithm is performed on data series’ reversals, i.e. a list made up from its peaks and valleys. Therefore, the first step is to reduce the given data to its peaks and valleys sequence. This is done iteratively over the whole series by comparing the difference between the current and previous point and the previous difference. This comparison points at any change in slope direction, which indicates a reversal. Peaks and valleys obtained this way are appended to a new list, which then becomes the data for which the rainflow algorithm is performed.

Instead of analysing one reversal at a time and iterating until reaching the end of the list, the rainflow algorithm stores at least three reversals and terminates once there are no more points to load. At every step, the algorithm compares two ranges: X, formed from the last two reversals, and Y, formed from the one- and two before last reversals. When the algorithm starts, an S (starting) point is set to the first reversal. The conditions present in the flowchart are implemented as stated, in the form of if conditions. What simplifies the algorithm is the fact that a range which should be counted as a (half) cycle does not have to be searched for in the ranges list. The same range can be present multiple times in the resulting list, with its cycles in the corresponding positions in the resulting cycles list, because the histogram method used later to create the figure gives the same result in both ways. Thus, adding a searching would be an unnecessary time waste, especially since the ranges list is unsorted, which results in a longer searching time.

Duty calculations use the ranges and cycles lists obtained from rainflow counting described above. Upon user selecting the channels to be plotted, this method is triggered by the rainflow thread, which means that it takes place outside of the main thread as well – the thread calls this method and waits until it is finished. Using the cycles and ranges lists, the cell QTableWidgetItem from the statistics table in the main window and the Wöhler slope parameters, duty described by the formula in 3.1.2 can be calculated and placed into the table. For the calculation itself, numpy.power() and numpy.sum() were used. The reason this method is passed and edits only QTableWidgetItem from the statistics table’s cells instead of receiving the entire QTableWidget as a
parameter is the fact that this table’s vertical header contains the `QIcon` in each row. Passing such a widget to a secondary thread returns a warning in each of the iterations, which states that it is unsafe to use `QIcon` outside of the main thread.

Duty normalized is triggered by double clicking any cell with a duty value in the statistics table. Then, the cell’s (`QTableWidgetItem`) background is set to light green (RGB: 180, 220, 180) and the clicked value becomes the reference duty. The normalized duties are obtained by dividing each duty by the reference one, and placing these values in corresponding cells.

Summed plots (level cross plot, rainflow histogram) are triggered by right-clicking a level cross or rainflow canvas and checking the summed plot action from the drop menu. The length of the summed line is 500 by default, without the possibility to change. Because the lines on the figures are both plotted against different coordinates and might have different lengths, obtaining the summed line begins with finding the minimum X value and maximum X value and using `numpy.linspace()` to create the X list for the line. Then, in order to obtain the figure values for these X coordinates, `numpy.interp()` is used. After interpolating all lines against the X vector, the consecutive values are added together to create one line, which is finally added to the figure as a black line thicker than the others. The resulting line based on a level cross plot is presented in figure 3.16. When switching tabs, the presence of the summed line is checked because every tab has its own canvas, but the drop menu is common to all tabs. Therefore, whether the summed plot action in the drop menu is checked or not depends on the presence of this line in currently active canvas.

Accumulated rainflow is triggered by right-clicking a rainflow canvas and checking the accumulated rainflow action from the drop menu. This procedure is carried out for each line separately, and consists of a `for` loop with a reversed range, from the end of a line’s data to the start. At every step, to the accumulated value the current Y value is added and the accumulated value is appended to the final list, backwards. When all lines are ready, the regular rainflow lines are hidden and the accumulated ones shown in their place. This function is presented in figure 3.17. When summed rainflow becomes selected, that method sums whichever lines are currently displayed, which means that it is possible to obtain summed accumulated rainflow histogram.
3.3.7. Analysis settings

Changing the parameters of the analysis, due to its complexity, requires a separate window. In the analysis settings window, the user can load files, select channels for plotting and change their settings, i.e. scaling, Wöhler slope, level cross reference value and level amount and rainflow bin amount. The implemented window is presented in figure 3.18.

The top side of the window consists of a toolbar, channel selection area and a file overview tree. The toolbar includes a file opening button and four navigation buttons. Clicking the first file button triggers the standard Windows file opening prompt (fig. 3.4), where the user may select one or more RPC type files which are loaded into the window upon confirmation. Then the channel selection table is resized to match the file and channels amounts, and the items for file overview tree are generated.
The four navigation buttons’ purpose is identical to the navigation buttons from the main window. Clicking one of them moves the entire channel selection one position to the clicked direction. There is no confirmation button in this case as there are no figures yet.

The channel selection area is a QTableWidget, alike the tables from the main Window. Upon accepting the settings and starting the analysis, this table will be places into the main window’s channel selection section. Also alike the main window’s tables, this one also has an EventFilter installed. This method handles all single clicks on the table, with the left mouse button click selecting a channel or multiple channels, and the right mouse button click de-selecting a channel or channels. Also, it is possible to change the colour of a cell by double clicking it and using the colour selection dialog window.

The file overview tree is a QTreeWidget with the sole purpose of providing detailed information about the loaded files. When files are loaded, the units for each channel are extracted and minimum and maximum values for each channel are calculated. All this information is then placed into the low level QTreeWidgetItem.

This widget’s role is displaying, so it has no methods triggered by a mouse click.

In the lower side of the window there is the setup area. It consists of a channel selection tree and a grid with parameters input.

The file selection tree presents which channels have been selected from the table and which files do they belong to. The tree contains 7 columns: Name, Selection, Scaling, Reference, Bins, Ranges and Wöhler. The Name column is shared by top level QTreeWidgetItem with the file names, and low level channel names. The checkboxes next to the channel names determine of which channels the parameters are being changed. The checkboxes at the file names enable checking all channels belonging to this file. Selection column contains boxes coloured accordingly to the channel selection table, which makes for a simple reference. Scaling, Reference, Bins, Ranges and Wöhler include the corresponding parameters for each channel: the data’s value scaling, level cross reference level, level cross levels amount, rainflow histogram’s bins amount and duty calculation’s Wöhler slope parameter.

The parameters presented in the file selection tree can be changed by using the QLineEdit with the corresponding QLabel before it. To enable changing the parameters for only one channel, multiple channels or all channels, depending on the user’s choice, the values given to the QLineEdit are saved only to the channels which have been checked in the channel selection tree. The parameters’ values displayed in the tree are changed right away.

When all parameters are set up, clicking the START ANALYSIS button saves the parameters list, loaded file list, selected channels and creates a new tab for this analysis. This means using the channel selection table in the main window, generating the figures from the selected channels right away, filling the file list and all consequential functions.

3.3.8. Reporting

Another key feature of the program is the possibility to save the progress or results of the work the user has performed. Currently, since the work is done using multiple software tools and also manual calculation, future users of the software are accustomed to saving the results of their work in the form of MS Excel spreadsheets. Therefore, the reporting tool saves the selected analyses as a report into MS Excel files. Additionally, this report format enables the user to edit the files and possibly combine them with the results of working in any other software.

The tool must enable selecting which part of the work is to be saved and what components should be saved. To incorporate all possibilities, a dialog window presented in figure 3.19 was implemented.
On the left side there is a list of QCheckboxes, where each checkbox represents a tab in the main window. Here, the user selects which tab’s analysis should have a report generated for. Selecting multiple tabs produces this many Excel files, as the tool produces one report per tab.

On the right side there is a section with options for the reports. The topmost drop list is implemented as a QSpinBox. It is filled with the tab names which have been checked, and therefore can have their reports’ settings changed. The option area below applies to the currently selected tab in this drop list.

In the option area, a list of checkboxes of the components of the report enables turning any number of them off, in case some of them were not needed. The tree, implemented as a QTreeWidget, presents the files loaded into the selected tab, as well as channels which are present in the figures. Then, a pair of QSpinBoxes lets the user select a file and channel which will serve as a reference duty in the normalized duty calculations. The calculations regard the “Full file statistics table” option, which stands for a collection of tables, one for each loaded file, filled with the channels’ units, basic statistics, duty and duty normalized. The final widget is a QLineEdit, which provides the input for a report file’s name. There is a separate set of these widgets per each selected tab, and switching between them in the topmost drop list updates the options for the selected tab.

The final widget is the confirmation button. Upon clicking it, the report generation thread is called and the entire procedure is executed.

Report generation is a for loop which goes through all selected tabs, creating a report for each of them. The first step of this procedure is to save the figures for the current iteration’s tab. The figures, as they are seen in the main tabs, require minor changes. First of all, it is necessary to add a legend to them. Matplotlib’s default legends tend to cover the figure once they have to contain many lines. Therefore, when using `axis.legend()` method to add a legend, this method’s `bbox_to_anchor` parameter was used which enables shifting the legend’s box up by a fixed amount of pixels depending on the amount of lines present in the figure. The content of the legend is generated automatically by the library based on the figure’s lines and their previously defined labels. Additionally, in order to introduce a common size and resolution of the images in the report, regardless of the original state of the figures, resizing the figures is needed. The resolution is adjusted upon calling `figure.savefig()` method, with `dpi` as one of its parameters. This method saves the figure as an image of a chosen type (png in this case) on the computer’s hard drive. Its `dpi` parameter’s value is set to 300 for all figures, which provides a satisfactory optimum between the image quality and its file size on the drive, which has an impact on the saving time. These images are later added to the report’s second spread sheet.
Each report consists of two spread sheets in one MS Excel file: the first one contains the file information and statistics tables, and the second spread sheet contains the saved figures. The spread sheets have a fixed layout presented in figure 3.20 and 3.21, with the file name and creation date up top and one table per file. Saving any numeric or string data to an MS Excel file requires only the appropriate cell index. The spacing between each table depends on the number of channels in the latest file. To load the figure images into the second spread sheet, a temporary image variable is created using `openpyxl.drawing.image()`, with the `png` file of a figure as its first parameter and desired image size as the second. This ensures that all saved figures have the same size. The saving itself is done using `sheet.add_image()` while passing the image variable and target cell index for its top left corner as variables.

The final step is to save the workbook using `workbook.save()` and passing the user-requested file name or generating a default name based on the reported tab’s name.

It is worth noting that the figures saved in the report are the exact form as they were in the main tabs. This means that any change introduced to a figure, e.g. zooming, hiding a line or changing axes range will remain in the report.

![Fig. 3.20. The first sheet of an exemplary report generated by the application’s reporting tool](image_url)
3.3.9. Sequence creation

Sequence creation is a tool with the purpose of helping the user to evaluate a testing sequence which can later be implemented in a test rig on the company’s site, as described in 3.1.5 and 3.2.5. This process is simplified to loading multiple files and setting a repeat count to each of them. In addition, a possibility to zero out all values in selected channels is required. The sequence creation tool’s dialog window is presented in figure 3.21.

![Fig. 3.21. The implemented sequence creation tool’s window](image)

The left side of the window consists of a `QPushButton` and a `QTableWidget`. Clicking the button triggers the file opening procedure, as described in the previous chapters. The table’s row count is equal to the number of loaded files, and in each cell of the last column there is a `QSpinBox`, added to the table using `setCellWidget()`. The values of the spin boxes represent the repeat counts for their corresponding files.

In the right side of the window there is a `QTreeWidget`, `QLabel` with a `QLineEdit` and a `QPushButton`. The tree widget presents the channels for every loaded file. For each channel, a QCheckbox is placed in the
The calculations, i.e. the sequence creation procedure are executed upon clicking the CONFIRM button and consist of level cross, rainflow and duty calculations, as mentioned in 3.2.5. This procedure is handled by its separate thread and therefore outside of the main program thread, which prevents the program from not responding.

The first step in level cross counting is to search all loaded files for a global minimum and maximum per each channel. From these two values and the default bin size of 500 the list of all level values is generated using numpy.linspace(). For each loaded file and each of its channels, the level cross counting algorithm is executed. As all files have the same set of channels, the resulting sequence carries the same channels. Therefore, a separate level cross is obtained for every channel by summing together corresponding channels in all files, taking the repeat values into account. The summing itself is done as described in 3.3.5.

Calculating the sequence’s rainflow histogram follows the same principle as far as the summing goes. Then, the difference is that the bin values are different for each of the obtained histograms, which causes a need for an interpolation (3.3.5).

The statistics also have to take multiple channels into consideration. For any given channel \( X \), the statistics are obtained from all loaded files, i.e. channel \( X \)’s minimum is the lowest value between the minimum values of this channel in all files (same for maximum value and maximum range), and the lowest mean and root mean square value among all of them. Duty value for each channel is calculated from its rainflow output and added together for corresponding channels.

Once the sequence data is processed, it is added to the main window as a new tab, or to an existing tab containing sequences. The right-side area of the main window remains the same, since the sequences act as files. Therefore, they are appended to the file list, and the selection table’s row and column count corresponds to the generated sequences. The difference here is that selecting a channel in the table triggers presenting only the level cross plot and the rainflow histogram. Because the data is pre-calculated and stored in memory, the plotting is instantaneous. The resulting sequence tab is presented in figure 3.22.

Fig. 3.22. The created sequence added to the main window
Large part of the sequence tab is similar to the regular tabs (fig. 3.8 and 3.12). The extra widgets placed instead of the time series plot and the PSD plot are a \texttt{QScrollArea} and a \texttt{QTreeWidget}. The scroll Area contains a \texttt{QLabel} and a \texttt{QTableWidget} for each created sequence. The tables present the basic statistics for all channels in every sequence and they refer to a sequence described by the \texttt{QLabel} above each of them. The tree widget, similarly, presents information on all sequences created. The top level \texttt{QTreeWidgetItems} are the sequences’ names, and their sub-items are the names of the files that make up each sequence. Also in this widget, an extra column provides information on how many times was each file repeated.
4. CONCLUSIONS

The fundamental goal of this thesis project was to design a reliable software tool which would enable measurement data analysis within a GUI environment. Because all of the core requirements were met and the resulting application was proven to be correct, this goal was reached.

The correctness of the calculations and display outputs were approved by thesis supervisors at Volvo, who had also been testing the software in terms of reliability and error-freedom. The implemented figure analysis tools provide basic functions, which enable drawing valuable conclusions from the plots. However, a number of additional tools were not implemented and are included in the future work section (chapter 5).

The specification for both reporting and sequence creation tools was simplified because of the time limit for the software development. This specification was fulfilled in both cases and the two tools provide their basic functionality. Saving the analysis’ results and creating a series of events for failure evaluation are both important aspects of the durability assessment and therefore if would be beneficial if they could have been further developed.

As mentioned above, the application has been continuously tested for errors. During the development, many bugs were brought out by the supervisors. Their sources varied from simply not taking certain behaviours under consideration, through misusing the methods provides by the libraries up to misunderstanding specific requirements set by the supervisors. Some of the issues required redesigning entire functions, where the primary example would be the first version of the level cross counting algorithm. Even though all of the errors that were found so far were fixed, it is expected that in the early stages of regular use of the software some unexpected issues might occur. The authors of this thesis provided the users with the program’s source code as well as code description and a maintenance manual, which should provide enough means for any such faults to be dealt with by the users.

PyQt has proven to be a good choice for the GUI framework. It provides a great amount of commonly used widgets along with many means of customising them through inherited properties. Moreover, every widget comes with its own methods, properties and signals, which enable creating very flexible tools and applications. Lastly, it has a very thorough online documentation, which provides a clear description to all of the above as well as code examples, although using C++ language.

An important matter for the end user is the program’s performance depending on the hardware. The computation power of the computer has a direct influence on the time consumption of calculations, plotting and report generation. However, the company computers at Volvo Car Corporation are fairly modern and are sufficient for smooth performance of the program. Another matter is the end look of the graphical interface in relation to the used screen’s resolution. The size of interface elements depends on the resolution – the higher it is, the smaller the elements are. The biggest impact in this regard falls to elements sized by *matplotlib* – primarily the subplots’ size and axis labels. Therefore, setting the size of any of these elements required obtaining the screen’s resolution first and then using it to specify the sizes.
5. FUTURE WORK

The time given for a master thesis is sufficient to deliver all the core functions of the application. However, there are a number of additional options that are useful or desirable for the users. Those can be added onto the application either by the users themselves using the maintenance manual prepared for the software, or put together to form another degree project.

The first of the additional options is a group of extra plot analysis tools. The idea behind these is to help the user extract information from the figures or compare different figures more easily. The major tool in this group is marking a region of a plot. This means clicking and dragging the mouse cursor over a figure, creating a highlighted area in the figure, which indicates the marking. Marking itself has no influence on the data, analysis or calculations. Its primary purpose is to be able to switch between the channels (e.g. using the navigation buttons) while retaining the marked region, which makes it easy to compare the occurrences in the data from different channels for the same time window. This also means that if the figure includes subplots, then the marked region should be present and identical for all subplots. As a supplementary action, it would be possible to influence the analysis using the marked region. The two tools for this are: deleting and duty calculation. Deleting means removing a part of the time series data, which would affect all other plots and calculations. This comes in handy if the measurement was done for a repeating event, for example braking a number of times – this feature would enable removing idle measurements between the actual braking occurrences. The other tool, duty calculation, means providing the duty value calculated only for the selected region. This would help evaluate a specific part of the time series plot.

Another group of additional functions refers to analysing the figure’s extrema. The first function is enabling the user to specify an amount of extrema, which values would be returned. To extend this feature, it could be possible to define a minimum distance in data points between the returned extrema. Another approach to searching for extrema is defining a number of regions in a figure, and then returning the extrema for each region.

The future work can also include increasing the GUI’s customisability. Primarily, this means for the user to be able to determine which widgets should be present in a given tab and what their position on the screen should be. This enables omitting irrelevant elements of the analysis, changing the size of the figures within the tab or moving a number of lines from a figure to a new one. Additionally, the figures themselves can be made more customisable. The first step towards this is creating a user-definable amount of subplots for a selected figure, and letting the user determine the distribution of the figure’s lines between the subplots. Another step is to enable customising the figure’s axes. This can be done by specifying the amount of axes in a selected figure (or subplot) which would solve problems coming from scaling or range differences among displayed lines.

Moreover, a useful feature to the application would be a possibility to filter the loaded data. The primary filter here would be a band-pass filter, with cut-off frequencies determined by the user. The major requirement for the filter is a pass band as close to an ideal one as possible, i.e. almost a perfectly rectangular transition slope.

The extra work could include incorporating additional data formats of the RPC files. Currently, the application works on integer data. The expansion would primarily add the possibility to read files with data saved as floating point format.

An additional type of figure can also be incorporated into the application. This figure is a three-dimensional rainflow histogram. Such plot introduces and additional source of information, which is number of cycles per a given mean value. That results in three following axes: X – range, Y – mean, Z – cycle count. The 3D rainflow histogram could be switched between with the regular 2D histogram or added as a new figure to the main layout and there should be a possibility to add it to a report, if one is generated.

Lastly, it would be useful to add an option to save a selected analysis. This would mean saving the list of loaded files, selected channels and specified parameters, as well as the analysis’ tab’s layout and other
settings. This information should be saved in a file format enabling reading it back and recreating the analysis. This function is a necessity in case of more complicated, time-consuming analysis, to which the user would have a possibility of returning later.
REFERENCE LIST


