NTNU
Norwegian University of
Science and Technology

Analysis and Synthesis of Rope Rosettes

Sigrid Brevik Andersson

Master of Science in Computer Science
Submission date: February 2016
Supervisor: Theoharis Theoharis, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Creating rose rosettes and other similar interlace patterns is an old form of creative expression. Rope rosettes is rooted in the seafaring tradition, where they were made both for decoration, to protect the sailors from slipping on the deck and to limit wear and tear of the wood. The creation of new patterns was quite time-consuming with no guarantee of success.

In recent years, Nils Kristian Rossing at Trondheim Science Center has developed a method to analyze rope rosettes and describe them mathematically. His approach is based on the Fourier analysis rooted in signal theory. With the new mathematical description, rosette patterns can be synthesized easily. Rossing's work also includes extensive work in describing rosette properties and categorizing them, as well as studies on the impact of different parameters in the rosette description.

Based on Rossing's findings, the goal of this thesis is to build a software tool to automatically analyze and synthesize rope rosettes. I name this tool Rosette Analyzer (RosAna), and it is built using the modern tools of Qt(C++) and OpenCV. The tool has three distinct parts: image analysis to extract the rosette pattern; Fourier analysis to determine the mathematical parameters; and pattern synthesis.

Sammendrag

Fletting av taurosetter og andre mønstre er en gammel kunst. Taurosetter har sitt utspring i sjømannstradisjonen. En taurosett var for det første vakker å se på, men ble også brukt for å hindre at sjømennene skled på det våte dekket og redusere slitasje. Å lage nye rosettmønstre var en tidkrevende prosess, uten noen garanti for suksess.

I nyere tid har Nils Kristin Rossing ved Trondheim Vitensenter utviklet en metode for å analysere taurosetter og beskrive dem matematisk. Han baserer sin metode på Fourieranalysen, hentet fra signalteori. Med Rossings matematiske beskrivelse er det enkelt å tegne nye rosettmønstre. Rossing har også lagt ned arbeide i å beskrive egenskapene ved taurosetter, samt kategoriseringen av disse. Han har også studert sammenhengen mellom de matematiske parametrene og taurosettens egenskaper.

Målet for masteroppgaven er å bygge et verktøy for rosettanalyse og syntese, basert på Rossings resultater. Jeg kaller verktøyet Rosette Analyzer (RosAna), og det er implementert i Qt(C++) og OpenCV. Verktøyet har tre distinkte seksjoner: bildeanalyse av en taurosett for å avdekke det underliggende mønsteret; fourieranalyse for å finne parametrene for den matematiske beskrivelsen; og mønstersyntese basert på resultatene fra fourieranalysen.

Preface

This paper is written as a part of my Master thesis conducted at the Norwegian University of Science and Technology (NTNU) in Trondheim. The time frame for the masters thesis has been 21 weeks. The master project is the continuation of my specialization project *Analyzing rope rosettes using image processing techniques* (Andersson, 2015).

I wish to thank my main supervisor Theoharis Theoharis (Department of Computer and Information Science (IDI), NTNU) and co-supervisor Nils Kristian Rossing (Trondheim Science Center). A special thank you to Nils Kristian for sharing his encyclopedic knowledge of rope rosettes and Fourier analysis code with me. I would also like to thank the Visual Computing Group for technical suggestions and ideas. An extra special thank you goes to my family for love and support.

Sigrid Andersson
Trondheim, February 5, 2016

Contents

List of Illustrations

Algorithms

Tables

Chapter 1

Introduction

Rope Rosettes and related interlace patterns can be found all over the world and in many cultures (Grünbaum & Shephard, 1992; Gerdes, 2007; Yanagisawa & Nagata, 2007).

The analysis of a Rope Rosette covers several disciplines; signal processing for mathematical analysis, trigonometry for synthesis and craftsmanship for the final production. In an effort to digitize such patterns and to simplify the process of analysis, automatic pattern extraction would be useful. The use of image processing techniques for rosette analysis was researched in my specialization project (Andersson, 2015).

Rossing has developed a tool to perform rosette analysis using MATLAB®. Currently, his process is composed of several steps (Rossing & Krifel, 2003, ch. 9.3).

1. Trace a rosette onto a piece of transparent plastic
2. Place the plastic over the screen and trace the rosette onto MATLAB®s coordinate system
3. Run the MATLAB® analyzer
4. Select number of components
5. Tweak parameters and synthesis using some other program (e.g. Winplot etc)

1.1 Task specification

Goal *Build a software application to analyze and synthesize Rope Rosettes.*

The main goal is to make a unified software tool where image tracing, Fourier analysis and pattern synthesis can be done within the same application.

Sub-goal *Automatic parameter selection.*

After a Fourier analysis in Rossing's old tool, the user is left to make decisions about some of the parameters for the mathematical description. Specifically, deciding which frequencies to choose and whether a frequency in fact is negative. Parameter selection places high demands on the user's knowledge of mathematical rosette analysis. Therefore it would be useful to obtain automatic parameter selection in the new tool.

1.2 Development tools

Three main languages are used in image processing; MATLAB®, Python and C++. The latter two combined with the open source library OpenCV. Examples of specifically geared programming languages are Halide (Ragan-Kelley et al., 2012) and Diderot (Chiw, Kindlmann, Reppy, Samuels, & Seltzer, 2012). However, using a more widespread language seemed prudent to ensure portability and simplify potential future development.

Previous work by Rossing and myself has been done in MATLAB®. While MATLAB® and its Image Processing Tool Kit (IPTK) makes image analysis simple, it suffers from drawbacks in speed and portability (Mallick, 2015). Therefore, it was decided that the new application should be developed in a new framework.

As summarized in (Mallick, 2015), and in the general opinion of the developer community, MATLAB® and Python are mostly used for prototyping, while C++ for actual application development. The choice of C++ is based on speed, portability and ease of use with OpenCV.

To make the Graphical User Interface (GUI) of the application, the choice fell on Qt, an open-source framework for cross-platform development. It natively supports C++, but has bindings to other languages as well (e.g. Python, Java)(The Qt Company, 2015a). The open source visualization library OpenGL is also built in with Qt.

In summary, the tools of choice for this project are C++(4.8) with OpenCV(2.4) and Qt(5.5), and documentation generated with Doxygen (van Heesch, 2015). In addition, Figure 3.1 and Figure 3.2 was created using Visual Paradigm (Visual Paradigm, 2016), and Figure 3.3 with GeoGebra (International GeoGebra Institute, 2016). This report was typeset with LaTeX.

1.3 Outline

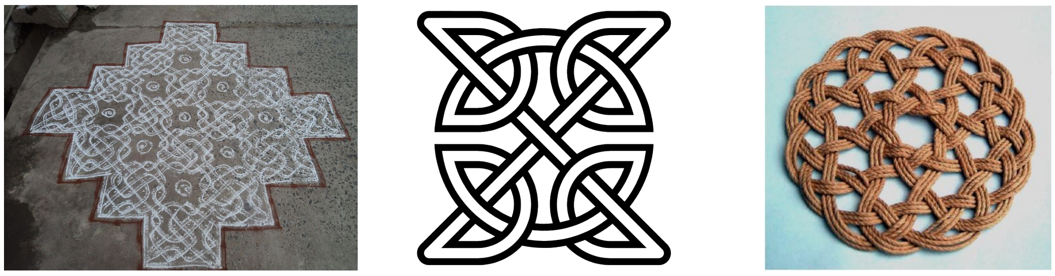
The remaining chapters are organized as follows. Chapter two gives an introduction of rope rosettes and their properties, and of the mathematical background. It also contains an overview of the current state-of-arts for interlace pattern analysis and a summary of current image processing techniques for contour extraction. The implementation details for RosAna is given in chapter three, and the completed tool is presented in chapter four. Results of rosette analysis and RosAna performance is given in chapter five with more details in Appendix B, followed by evaluation and conclusion in chapter six. Glossary, bibliography and appendices are included at the end.

Chapter 2

Background

This chapter gives a brief overview of the technicalities and mathematical foundations of rope rosettes. An overview of the field as it is today is provided, focusing on scientific work done on rope rosettes and related types of patterns. Finally, the findings in my specialization project (Andersson, 2015) is summarized.

2.1 Rope rosettes



(a) Kolam pattern, Wikipedia Commons, 2012 (b) Celtic knot, Wikipedia Commons, 2007 (c) Rattan rope rosette, Rossing, 2012

Figure 2.1: Select interlace patterns

Rope rosettes comes in many forms. They have their foundations in the seafaring tradition, and have many similarities with other traditional interlace patterns, like the Sona patterns from Angola and neighboring regions, Kolam patterns from South India and Celtic decorations, to mention a few (Rossing & Krifel, 2003, ch. 2).

2.1.1 Properties

Rope rosettes have some important properties that can be exploited in the construction of an analysis tool. Although it is possible to synthesize many different patterns, some of these are hard or impossible to realize in rope form. As the goal is to analyze physical rope rosettes, some simplifying assumptions can be made:

- Only a single run is required to trace the whole pattern. All line segments are connected in one continuous line.
- Crossings are made at an approximate 90 degree angle. Each crossing involves exactly two line segments.
- Crossings should not be too close, and ideally approximately equidistant.
- Which segment goes under and which goes over in a crossing is irrelevant. As the first crossing is defined, all other crossings will alternate along the path traced from the starting point.
- Rosettes are rotational symmetrical around the center. The symmetry is based on the number of times the sub-pattern repeats.

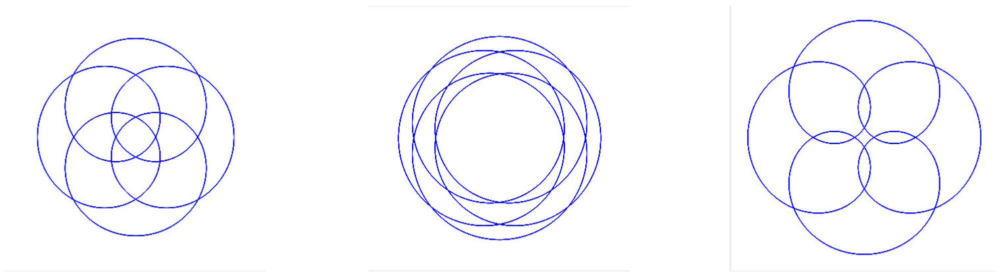
2.1.2 Mathematical foundation

As discovered by Rossing, any rope rosette can be described by two periodic functions with several components, one each for the x- and y-direction (Rossing, 2012). One can visualize the periodic functions as a series of rotating vectors, connected end to end to each other. Each vector has a length, a direction of rotation, a speed of rotation and a starting point. The visual properties of the rosette can all be determined by analysis of the mathematical description, and vice verse. These properties of the periodical functions and the rope rosettes and how they are related to each other are summarized below. The mathematical property is listed first, and the rosette property in ***bold italic font***. A more in-depth description can be found in (Rossing & Krifel, 2003, ch. 7, 9).

The mathematical description of a Rope Rosette is as follows; two periodical equations, one for x- and one for y-direction. Following the convention of Rossing and Krifel, they have the general form of Equation 2.1, where A_i is the amplitude and f_i the frequency of component i. Note that for some rosettes, the sines and cosines might be switched in some components.

$$X = f(t) = A_1 \cos f_1 t + A_2 \cos f_2 t + \dots \tag{2.1a}$$

$$Y = f(t) = A_1 \sin f_1 t + A_2 \sin f_2 t + \dots \tag{2.1b}$$



(a) Closed, $A_1 = 1, A_2 = 1.4$

(b) Ring-shaped, $A_1 = 1, A_2 = 4$

(c) Non-overlapping, $A_1 = 1, A_2 = 0.9$

Figure 2.2: Variations of the Turk's head rosette. $f_1 = 1, f_2 = 5$ Images: Sigrid Andersson

Number of components Defines the ***order*** of the rosette.

Rosettes are categorized after their order, which is the minimum number of Fourier components needed to describe the rosette. For instance, the Turk's head rosette is of order 2, and the Eye rosette of order 5. The concept of *order* is not stringently defined, and should be seen more as a guide.

Amplitude A_i Defines the ***type*** of the rosette.

A rosette is one of three main types; closed and overlapping, open and ring-shaped or non-overlapping. A rosette is closed if the bights encloses the center, which makes the center hole very small or disappear. An open, ring-shaped rosette lies as a ring around the center. A non-overlapping rosette has bights which do not enclose the center. Figure 2.2 shows the different types.

Frequency f_i Defines the general ***shape*** of the rosette. Below follows some descriptions of the specific frequency properties.

Pairwise frequency differences The ***bights*** of the rosette.

A bight is a loop on the outer edge. In this setting, a bight equals a sub-pattern, such that the frequency differences are the number of sub-pattern repetitions. See Figure 2.3.

Fundamental frequency The largest frequency, defining the rosette ***slotting***.

The slotting is the number of strands crossing an imaginary straight line from the rosette center, see Figure 2.3. Slotting is not well defined for a non-overlapping rosette.

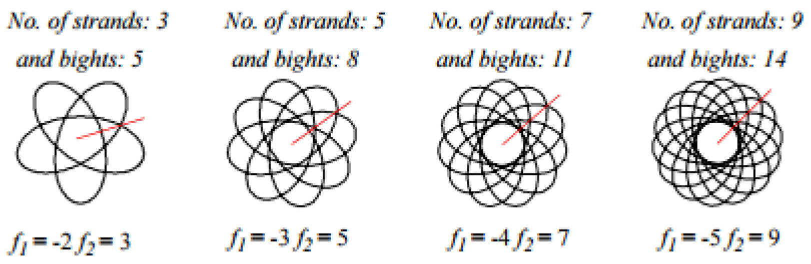


Figure 2.3: Rosettes and their properties, (Rossing, 2012).

Base frequency The smallest, positive frequency, determining the ***skipping*** of the rosette, that is how the sub-patterns are interconnected.

With a skipping factor of 1, each sub-pattern is connected to its direct neighbors. With a skipping factor of 2, each sub-pattern is connected to its second nearest neighbors, and so forth. The skipping factor is the same as the number of circuits traced around the center before reaching the starting point. Skipping is not well defined for higher order rosettes.

Due to the constraints of the task, the assumption is made that all relevant frequencies are integers and equidistant, and amplitude values in x- and y-direction are identical and positive for the same components ($A_{xi} == A_{yi}$).

2.1.3 Fourier analysis

The basis of the mathematical analysis of rope rosettes is the Fourier analysis. The formulas of interest are:

$$A_m = \frac{1}{\pi}\int_0^{2\pi} f(t)\sin mt dt \tag{2.2a}$$

$$B_m = \frac{1}{\pi}\int_0^{2\pi} f(t)\cos mt dt \tag{2.2b}$$

$$A_m \sin mt + B_m \cos mt = C_m \sin(mt + \varphi) \tag{2.3}$$

where $f(t)$ is the signal resulting from a trace of the rosette, A_m and B_m are Fourier coefficients at frequency m, $C_m = \sqrt{A_m^2 + B_m^2}$ and φ is the phase displacement. A detailed background on the mathematics in provided in (Rossing & Krifel, 2003, ch. 9).

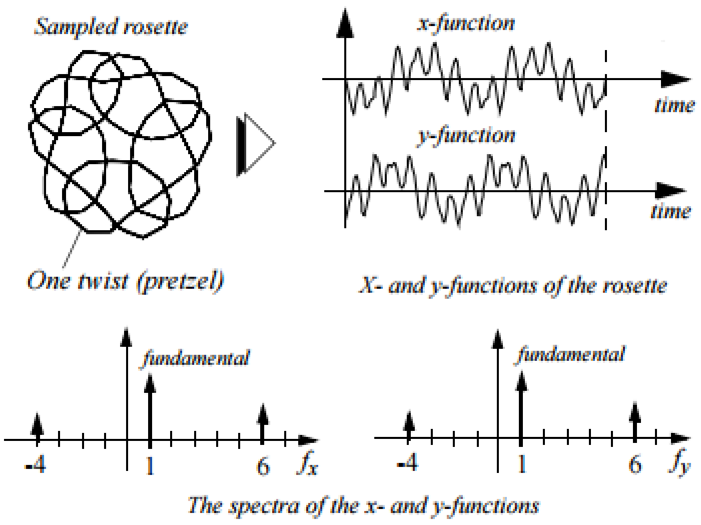


Figure 2.4: Fourier analysis, (Rossing, 2012).

Starting from a skeleton graph of a rosette, the Fourier analysis consists of the following steps:

1. Trace the pattern once, plotting the projection in the x- and y-direction, resulting in the signal $f(t)$
2. Solve Equation 2.2 for all integers within a given limit. Where the result is significantly larger than zero, the frequency/amplitude pair is one of the components of the rosette description.
3. Using the properties of trigonometry, Equation 2.1 can be written with only sine or cosine components.

2.2 Field overview

Subsection 2.2.1 summarizes the work done on analysis of rope rosettes and related interlace patterns, namely Sona, Celtic, Kolam, Nirosula, Oceania sand drawings, patterns of ancient Egypt and Mesopotamia, Lunda/Liki patterns, Cantor, mirror and meander patterns, Roman mosaic mazes and Islamic Laceria patterns. Subsection 2.2.2 summarizes the research into image processing methods to detect contours.

2.2.1 Analysis of interlace patterns

Several tools exist to generate different interlacing patterns, including a Celtic knot generator (Fung, 2007), a tool drawing a rosette with three vectors (HiB-Mediesenteret, 2004) and (Browne, 2005) which generates more advanced patterns out of simple ones, to mention some. Work has also been done in the digitalization and representation of interlace patterns (Nagata, 2007), and of classification of these patterns using a Cayley diagram (Grünbaum & Shephard, 1992). Ostromoukhov's method analyzes interlace patterns utilizing user driven input to help determine the pattern center and basic components of the pattern. Purkayasth, Dingliana, and Stalley uses a regular grid to identify crossing points in a Celtic pattern image, while Bhakar et al. extracts pattern components from textiles.

The closest tool that currently exists needs extensive user input to analyze patterns (Ostromoukhov, 1998). Several of the methods mentioned above relies on mirror symmetry. Rope rosettes contain circular symmetry, and benefits little from previous work in this area.

2.2.2 Contour detection

Contour detection is a difficult task, and has thus spawned a wide field. The many approaches has to deal with problems like noise, poorly defined edges, shadows and highlights, and detection of contours that arise from optical illusions. Pre-processing such as edge-preserving smoothing can alleviate some issues with noise and textures. However, care needs to be taken when choosing the detection method(s), as the different techniques are often tuned for a specific type of problem. Below is a short overview of the field, partly based on the findings of (Papari & Petkov, 2011) and (N. R. Pal & S. K. Pal, 1993).

Contour detection can be divided into roughly three different categories: *(i)* region-oriented methods, which extract the boundaries of closed regions; *(ii)* edge oriented methods, which directly extracts boundaries; and *(iii)* multiresolution methods, which employs some variety of scale space. Within each category, a given approach can be either local, global or hybrid. Local methods only take into account the data that can be extracted from the close neighborhood, while global methods consider much larger parts of an image. The global methods are often applied on the results of a local method.

Region oriented methods

Region oriented contour extraction can be divided into the following categories: thresholding, iterative methods, and surface techniques.

Thresholding is dividing an image into regions based most commonly on pixel values. The selected threshold can be global, multivalued or locally adaptive. The most well-known and extensively used thresholding method is Otsu's method, maximizing the class separability (Otsu, 1975). *Iterative methods* compute the regions in several iterations. Region growing and watershed transforms fall into this category, as well as the more advanced mean shift algorithm, which performs gradient ascent on the combined spatial-range feature space (Comaniciu & Meer, 2002). Relaxation labeling uses probabilities to assign labels to pixels (Kittler & Illingworth, 1985) but often gets stuck at local minima, while methods based on Markov Random Fields (MRF) are stochastic in nature and more likely to find global maxima. The usage of Neural Networks (NN) is another prominent methodology that can take into account both shape and texture, extract contours, cluster pixels and more. Egmont-Petersen, de Ridder, and Handels provides a review of MRF methods (Egmont-Petersen et al., 2002). *Surface based segmentation* is based on the idea of surface primitives: peak, pit, ridge, saddle ridge, valley, saddle valley, flat and minimal (Besl & Jain, 1986). These can be connected through region growing, spanning trees, clustering and differential geometry etc.

For multichannel images, all channels can be treated equally, or one or more channel can be segmented using different techniques.

Edge oriented methods

Local edge-oriented techniques include differential operators, statistical methods, energy and phase congruency analysis, morphological detectors and combinations of these. Of the *differential operators*, the Laplacian looks for zero-crossings to determine edges, the Sobel, Prewitt, Roberts and Beaudet operators looks for pixels matching a defined neighborhood, while the more advanced Canny operator provides a more optimized filter as well as some ground rules for edge detection (Canny, 1986). Some methods also try fitting polynomials or splines (Chen, 1995) to the edge, often by minimizing the mean squared error of the distance. The differential operators are all sensitive to noise. The *statistical methods* are more robust, but have a higher computational load. Many approaches compare two samples from different parts and directions in an image and computes the dissimilarity between them. Other statistical methods evaluate the gradient distribution around each pixel, analyzing co-variance or co-occurrence matrices, or

Eigenvalues and Eigenvector pairs. Analysis of *phase congruency or local energy* includes Gabor filters, Gaussian derivatives and difference of Gaussians (Kovesi, 1999). They are based on the Fourier transform and extract salient points where the decomposed Fourier components' phases are in sync. *Morphological detectors* uses ideas from morphology, namely the morphological gradient and extensions of it (Trahanias & Venetsanopoulos, 1996). These are similar both to linear filters and to statistical analysis, and have a trade-of between fast computation and robustness to noise and outliers. Several attempts have been made to unify two or more of the techniques above, using different methods from machine learning (Martin, Fowlkes, & Malik, 2004).

Global edge-oriented approaches can be grouped into three classes: *(i)* contour saliency; *(ii)* pixel grouping; and *(iii)* active contours. To compute *contour saliency*, the context around the detected points is taken into consideration. Points that are co-linear are enhanced while all other points are suppressed (Li & Gilbert, 2002). Popular approaches includes local averages, tensor voting and probabilistic methods like relaxation labeling. *Pixel grouping* is guided by the Gestalt principles (Wagemans et al., 2012), which are modeled on human perception of proximity, continuation, closure and symmetry. These often take into account the orientation of the detected edges and tries to connect them, as in applications for curvilinear extraction (Raghupathy & Parks, 2004). These methods provide more useful results for object recognition, but the computational demands are higher. *Active contours*, or snakes, are curves drawn around an object before an energy function is minimized (Kass, Witkin, & Terzopoulos, 1988). The basic snake can be extended to account for low contrast, noise and prior knowledge.

2.3 Previous work

In the preparation of this thesis, I conducted a specialization project (Andersson, 2015). The focus was the investigation of different techniques for contour detection. The techniques tested were Canny edge detection (Canny, 1986), Otsu's method (Otsu, 1975), marker-based watershed transform (MathWorks, n.d.), relaxation labeling (Kittler & Illingworth, 1985), active contours (Kass et al., 1988) and curvi-linear extraction (Raghupathy & Parks, 2004).

While the most methods were fairly successful at simple images, they all performed very badly on images closer to real photos. Test results are shown below for two images in Figure 2.5 and Figure 2.6. It is clear that no method yielded good results on the real rosette photograph. Canny, Otsu and relaxation labeling yielded fair results on the simple example. A detailed discussion can be found in (Andersson, 2015, ch 4).

As an implication of the results below, the focus on this thesis has been on building a complete tool with manual tracing, rather than automatic rosette pattern extraction.

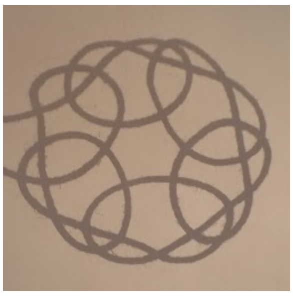
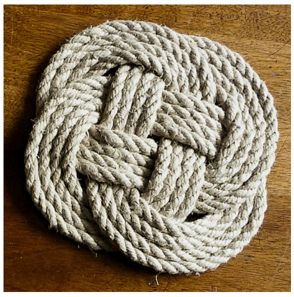
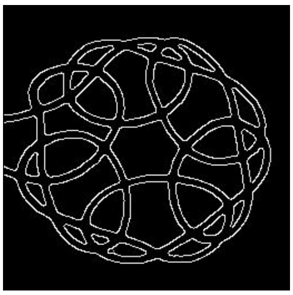
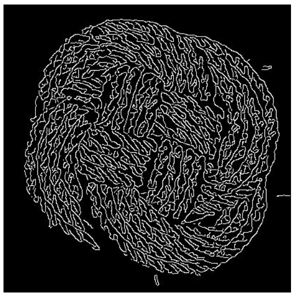
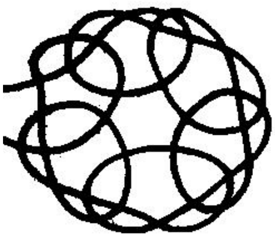
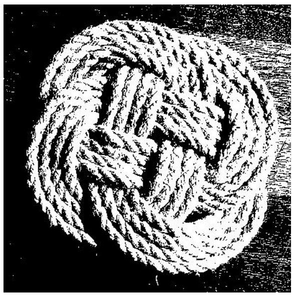
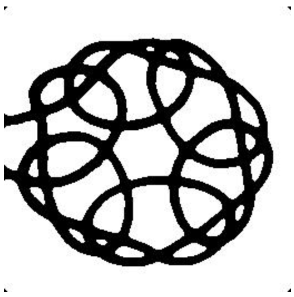
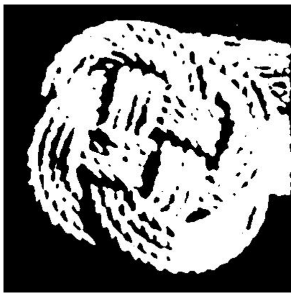
Method	**Simple rosette**[1]	**Turk's head photo**[2]
Original		
Canny		
Otsu		
Relaxation		

Figure 2.5: Results of rosette contour extraction, simple methods (Andersson, 2015).
Images: [1](Andersson, 2015), [2](Wikipedia Commons, 2013)

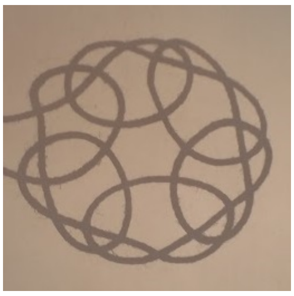
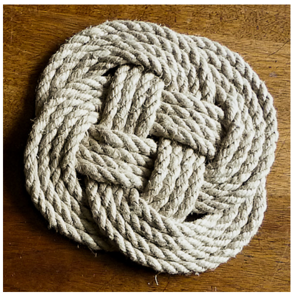
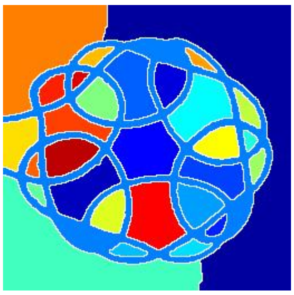
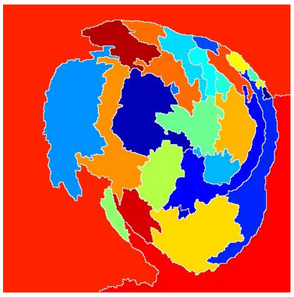
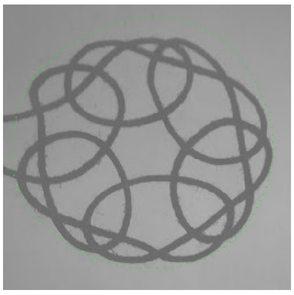
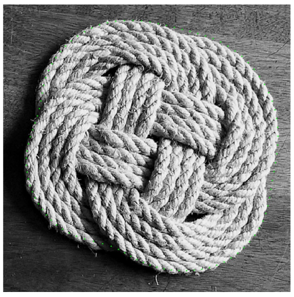
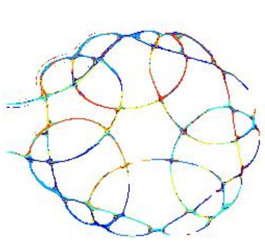
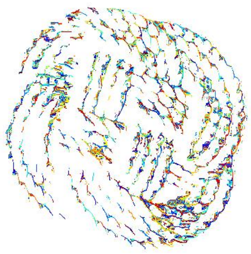
Method	**Simple rosette**[1]	**Turk's head photo**[2]
Original		
Watershed		
Active contours		
Curvi-linear		

Figure 2.6: Results of rosette contour extraction, advanced methods (Andersson, 2015)
Images: [1](Andersson, 2015), [2](Wikipedia Commons, 2013)

Chapter 3

Implementation of RosAna

This chapter documents the implementation and architecture of a new analyzing tool, named RosAna.

3.1 System description

RosAna is comprised of three main parts: *i)* rosette pattern tracing; *ii)* Fourier analysis; and *iii)* pattern synthesis. In addition, there are some files solely for Graphical User Interface (GUI) and some helper files. Lastly, two files were downloaded to provide plots and a link between OpenCV and Qt. Table A.1 lists the corresponding files. More detail is provided in Section 3.2.

3.1.1 Rosette pattern tracing

As shown in my specialization project (Andersson, 2015), automatic rosette pattern tracing is very difficult. The results are summarized in Section 2.3. None of the investigated methods would yield acceptable results in a real world setting. Therefore the decision was made to implement a simple hand trace only, using mouse clicks. The trace is saved as a list of coordinate points in `xData` and `yData`. When the trace is complete, the results are passed on for Fourier analysis.

3.1.2 Fourier analysis

The Fourier analysis is heavily based on Rossing's previous version, implemented in MATLAB$^{®}$. This subsection documents the contents of Rossing's algorithm, and Section 3.3 describes the steps taken to automate parameter selection.

Algorithm 3.1 Equidistant sample point calculation

```
Require: $x, y, z$ in xData, yData, zData
Ensure: $j \ngtr numberOfSamplePoints$
 1: $step =$ tracelength/NFFT
 2: $j \quad = 1$
 3: for $i = 0$ to NFFT-1 do
 4:     if $i * step > z_j$ then                    ▷ Keep track of old sample points
 5:         $j = j + 1$
 6:     end if
 7:     if $z_j - z_{j-1} == 0$ then               ▷ 0 distance between sample points
 8:         $j = j + 1$
 9:     else                                        ▷ Interpolate new points
10:         $xNew_i = \frac{(x_j - x_{j-1})}{(z_j - z_{j-1})} * (i * step - z_{j-1}) + x_{j-1}$
11:         $yNew_i = \frac{(y_j - y_{j-1})}{(z_j - z_{j-1})} * (i * step - z_{j-1}) + y_{j-1}$
12:     end if
13: end for
14: return List of x- and y-coordinates (xNew and yNew)
```

Data adjustments

In preparation for the Fourier transform, `xData` and `yData` are subject to a number of adjustments. First the coordinates are normalized to the range [0, 1]. All x-coordinates are scaled by a factor of 1.337 to get the correct relation between the axes. Then an estimate of the trace length is calculated using a rearranged Pythagoras formula (Equation 3.1), which results are saved in variable `zData`.

$$z_{i+1} = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} + z_i \tag{3.1}$$

where x_i, y_i, z_i are the i-th entries of `xData`, `yData` and `zData` respectively.

The range of all coordinates are then adjusted to be centred around 0 by subtracting the mean. Between all sample points, intermediate equidistant points are calculated as by Algorithm 3.1, where x_j, y_j, z_j are the j-th entries of `xData`, `yData` and `zData` respectively. The variable `tracelength` corresponds to the total length of the pattern trace, and variable `NFFT` corresponds to the number of data points for the Fourier analysis. This concludes the preparation phase. `xNew` and `yNew` are lists of coordinates to be used in the Fourier analysis.

Algorithm 3.2 Fourier analysis (X-values)

```
Require: xNew
1: for all points $x_p$ in xNew do
2:     FFT        = Fourier($x_p$)                ▷ Format: FFT = $a + bi$
3:     amplitudes $\uplus$ $abs$(FFT)/$\sqrt{NFFT}$      ▷ $abs$(FFT) = $\sqrt{a^2 + b^2}$
4:     phases     $\uplus$ $angle$(FFT)                  ▷ $angle$(FFT) = $\tan^{-}1(b, a)$
5: end for
6: return amplitudes sorted in decreasing order, keep the order in indexes
7: return phases
```

Fourier analysis

The Fourier analysis consists of several steps, where all steps are performed in both x- and y-direction (using `xNew` and `yNew`). The x- and y-direction calculations are independent of each other. The steps for `xNew` are shown in Algorithm 3.2, where `FFT` is the result of the Fourier transform, `amplitudes` and `phases` are the magnitude and phase angles of the Fourier transform.

3.1.3 Pattern synthesis

Using the values found in the Fourier analysis (see Table 3.1), a synthesized pattern can now be constructed, with the Rosette pattern description given by Equation 2.1, repeated below for convenience. Equation 2.1 is the standard equation form, however the trigonometric functions may be changed between sines and cosines. The parameters A_i and f_i are the amplitude and frequency of the i-th row of the final two columns in Table 3.1. The plot is drawn using `qcustomplot`.

$$
\begin{aligned}
X &= f(t_x) = A_1 \cos(f_1 t) + A_2 \cos(f_2 t) + A_3 \cos(f_3 t) + \dots \\
Y &= f(t_y) = A_1 \sin(f_1 t) + A_2 \sin(f_2 t) + A_3 \sin(f_3 t) + \dots
\end{aligned}
\tag{Equation 2.1 revisited}
$$

3.2 Software architecture

An overview of the RosAna is shown in Figure 3.1. See also Table A.1 for an overview of project files.

MainWindow Main class of the RosAna. It initializes the other classes, holds the main window GUI and sets up the model and signal/slots. User actions are initiated from MainWindow.

ProcessWidget GUI for image display and tracing. Image is displayed using `cvimagewidget`.[1] Has the capability of recording a rosette pattern trace, sending the results to `FourierAnalysis` when complete. Trace data can be saved to file.

FourierAnalysis Performs the Fourier analysis on sampled data. Displays the results to user using `FourierDisplayWidget` and creates a new model to be displayed in `DrawWidget`.

DrawWidget Draws the rosette pattern plot determined by the model, using `qcustomplot`.[2] Also holds the GUI of `VariableAdjustWidget`. Rosette pattern plot can be saved to file.

VariableAdjustWidget Provides a GUI for the user to view and edit parameters for the Rosette pattern description. Pattern parameters can be saved to file.

LoadExampleDialog The GUI for the user to load an example rosette pattern to display in `DrawWidget`. Loads the data read by `ReadExampleFile`, sending them to MainWindow to build a model.

ReadExampleFile Reads a file of example rosette pattern parameters. The file is on the format name, A_i, f_i, $\ldots$.

HelpDialogWidget Provides the user with some rosette theory, usage guide for RosAna, and author and licence information.

[1] Mottalli, 2012.
[2] Eichhammer, 2015.

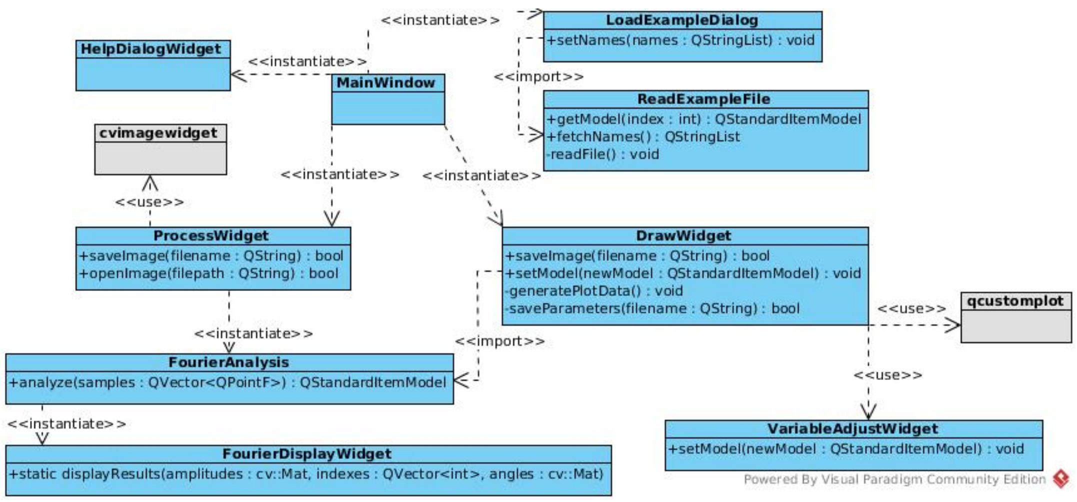

LoadExampleDialog
+setNames(names : QStringList) : void
HelpDialogWidget
MainWindow
ReadExampleFile
+getModel(index : int) : QStandardItemModel
+fetchNames() : QStringList
-readFile() : void
cvimagewidget
ProcessWidget
+saveImage(filename : QString) : bool
+openImage(filepath : QString) : bool
DrawWidget
+saveImage(filename : QString) : bool
+setModel(newModel : QStandardItemModel) : void
-generatePlotData() : void
-saveParameters(filename : QString) : bool
qcustomplot
FourierAnalysis
+analyze(samples : QVector<QPointF>) : QStandardItemModel
FourierDisplayWidget
+static displayResults(amplitudes : cv::Mat, indexes : QVector<int>, angles : cv::Mat)
VariableAdjustWidget
+setModel(newModel : QStandardItemModel) : void
<<instantiate>>
<<import>>
<<use>>
Powered By Visual Paradigm Community Edition

Figure 3.1: RosAna: Class diagram showing the most important relations and functions. Note that gray classes are external

3.2.1 The MVC pattern

Many classes need to access or write the values of a Rosette pattern description. To facilitate this, the implementation uses the Model-View-Controller (MVC) pattern. Pattern parameters are stored in a **M**odel, specifically a *QStandardItem-Model*. `DrawWidget` is a pure **V**iew class, only displaying data from the model. The **C**ontrollers are `FourierAnalysis` and `MainWindow`, both of which creates new models based on either Fourier analysis results or the values loaded from the example file. Lastly, `VariableAdjustWidget` can both **V**iew and **C**ontrol the model data.

3.2.2 Class interaction

In Qt, class interaction is handled through a system of *signals* and *slots*. All signals are public, and can be compared to events. Slots can be public or private, and is analogous to a listener. Figure 3.2 shows the most important signal-slot interactions. In addition to those shown, there are more private slots that are used internally in the classes, particularly in `MainWindow`. Class interaction due to Model-View-Controller is not shown in Figure 3.2.

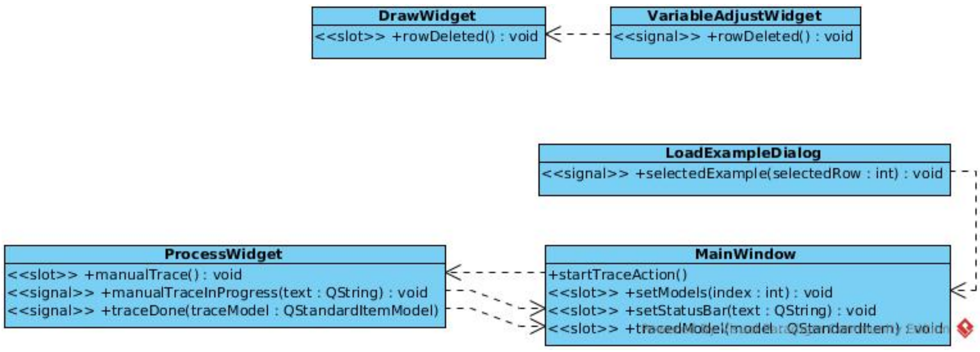

DrawWidget
<<slot>> +rowDeleted() : void
VariableAdjustWidget
<<signal>> +rowDeleted() : void
LoadExampleDialog
<<signal>> +selectedExample(selectedRow : int) : void
ProcessWidget
<<slot>> +manualTrace() : void
<<signal>> +manualTraceInProgress(text : QString) : void
<<signal>> +traceDone(traceModel : QStandardItemModel)
MainWindow
+startTraceAction()
<<slot>> +setModels(index : int) : void
<<slot>> +setStatusBar(text : QString) : void
<<slot>> +tracedModel(model : QStandardItem) : void

Figure 3.2: RosAna: Signal-slot diagram showing class interaction

3.3 Automating parameter selection

In Rossing's old Fourier analyser, some parameter selection and tweaking was done by hand. This section focuses on the selection of the number of components and the determination of which vectors (if any) had an opposite rotation. This is represented by negative frequencies.

3.3.1 Parameter selection

The frequency/amplitude pairs (f_i, A_i) of the x- and y-direction are sorted based on decreasing amplitude values. Each frequency/amplitude pair is compared to the following criteria until a pair fails.

1. $f_{xi} == f_{yi}$
2. $A_{xi} \geq$ `threshold` && $A_{yi} \geq$ `threshold` (set to 0.5 in the current implementation).

The final amplitude value is the average of x- and y-direction. Only the frequency/amplitude pairs that meet the criteria above are used for the rosette description. The remaining amplitudes are set to 0. Table 3.1 shows values obtained when analysing a twisted rosette.

X		**Y**		**Final**	
$\mathbf{f_{xi}}$	$\mathbf{A_{xi}}$	$\mathbf{f_{yi}}$	$\mathbf{A_{yi}}$	$\mathbf{f_i}$	$\mathbf{A_i}$
1	6.64	1	6.27	1	6.46
7	5.44	7	5.35	7	5.39
5	1.66	5	1.34	5	1.50
13	0.76	13	0.68	13	0.72
6	0.46	11	0.32	0	0
19	0.30	0	0.30	0	0
...	...	...	...	...	...

Table 3.1: The most significant parameter values for a twisted rosette, including the resulting frequency and amplitude values. f_i and A_i denote frequency and amplitude, respectively.

3.3.2 Negative frequencies

A Rope Rosette pattern description can be visualized using a series of rotating vectors connected end to end (Rossing, 2012), with amplitude as vector length and frequency as vector rotation. In general, frequencies are positive, denoting a counter-clockwise (CCW) rotation. However, clockwise (CW) rotation is sometimes used, represented by using a negative frequency.

Due to the nature of magnitude computation, all frequencies show up as positive. By examining the phase values, the rotation direction can be determined. Each phase is placed in a quadrant of the unity circle (Figure 3.3). If the quadrant Xi is CCW from the quadrant of Yi, the frequency is positive and the vector rotation is CCW. Otherwise, the frequency is negative with CW rotation. Table 3.2 summarizes the steps mentioned above for the twisted rosette.

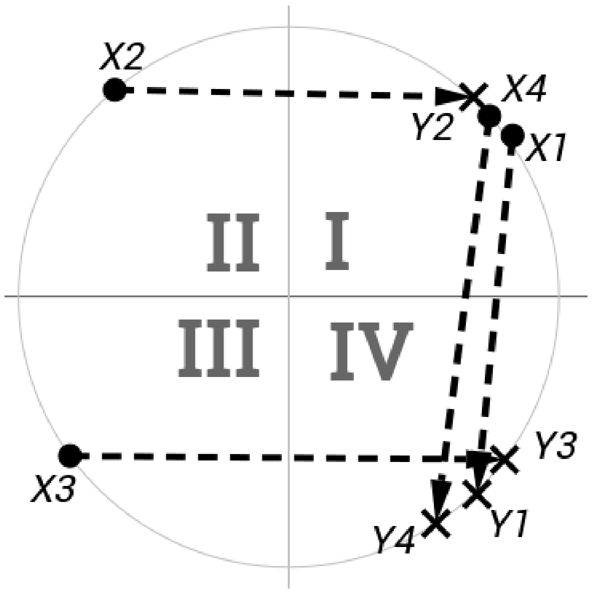


Figure 3.3: Phase angles and quadrants. The values are taken from Table 3.2 where *Xi* and *Yi* refers to the i-th row. Xs are marked by • and Ys are marked with **x**. Corresponding pairs are linked with --➝.

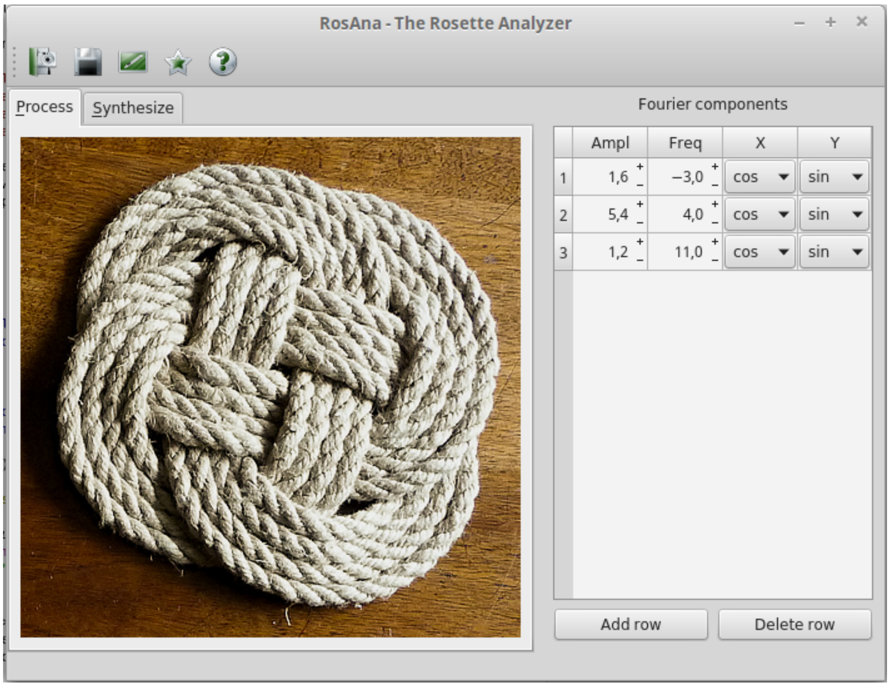
	x-phase	y-phase	x-quadrant	y-quadrant	Frequency
1	44.24	313.54	1	4	1 (CCW)
2	130.32	41.56	2	1	7 (CCW)
3	216.67	323.64	3	4	-5 (CW)
4	42.44	303.32	1	4	13 (CCW)

Table 3.2: Phase values and their responding quadrants of a twisted rosette. Column 5 shows the changes to the frequencies.

Chapter 4

RosAna installation and user guide

The tool is named RosAna, from **Ros**ette **Ana**lyzer. It comes preloaded with one example image, a four bight Turk's head, courtesy of Wikipedia Commons (2013). Figure 4.1 shows the start screen.



	Ampl	Freq	X	Y
1	1,6	−3,0	cos	sin
2	5,4	4,0	cos	sin
3	1,2	11,0	cos	sin

Figure 4.1: RosAna start screen

4.1 Installation

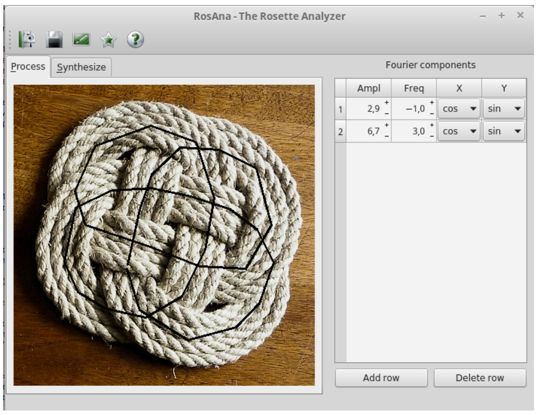
Running RosAna and further development requires the installations of Qt and OpenCV. See http://doc.qt.io/qt-5/supported-platforms.html for further information pertaining to the specific development operation system.

4.2 Usage

Currently, there are two modes available; *Process* and *Synthesize*. Pattern tracing is done in Process mode, as shown in Figure 4.2a. This includes initiation of the Fourier analysis, where the results are shown as in Figure 4.2b. Pattern synthesis is done in Synthesis mode, see Figure 4.2c, where also example rosettes can be loaded (Figure 4.2d). The following actions are available in RosAna:

📷	**Open image**	`Ctrl+o`	RosAna currently accepts images on the formats .xpm, .jpg, and .png.
💾	**Save image**	`Ctrl+s`	Saves the current trace in Process mode, or the plot and optionally the Rosette pattern description parameters if in Synthesize mode.
✏️	**Begin trace**	`Ctrl+t`	Initiate user led trace of the current rosette image. Each mouse click's position is recorded. This option is inactive in Synthesis mode.
	End trace	`Enter`	End the current rosette trace and initiate Fourier analysis. Has no effect if no trace is in progress. The Fourier analysis results are displayed in its own window and RosAna is set in synthesize mode.
★	**Load example**	`Ctrl+l`	Load example rosette in synthesize mode. Currently there are six example rosettes available; 7-bight Turk's Head, 6-bight Twist, 6-bight Double Bow, 6-bight Eye, and 8- and 10-bight Alternating Eye.

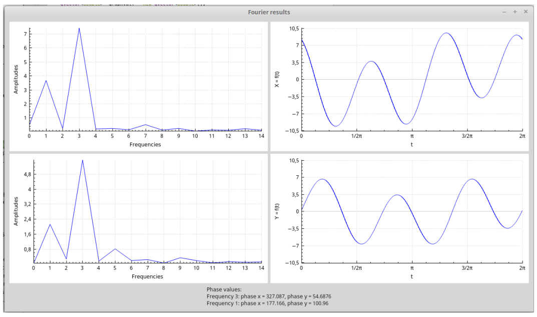
Help & about	`F1`	Open the Help and About dialog. The dialog includes basic rosette theory, this usage guide and credit and licensing information.


RosAna - The Rosette Analyzer

Fourier components

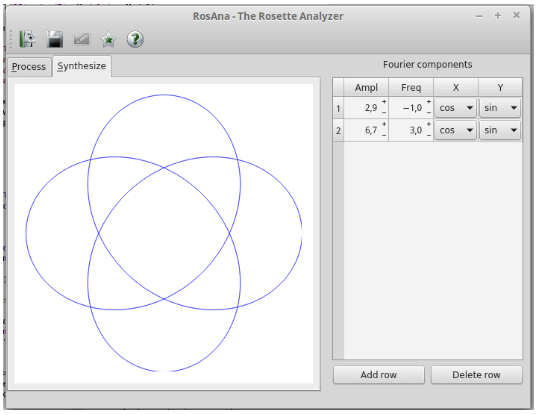
	Ampl	Freq	X	Y
1	2,9	−1,0	cos	sin
2	6,7	3,0	cos	sin

(a) RosAna, tracing a rosette



Phase values:
Frequency 3: phase x = 327.087, phase y = 54.6876
Frequency 1: phase x = 177.166, phase y = 100.96

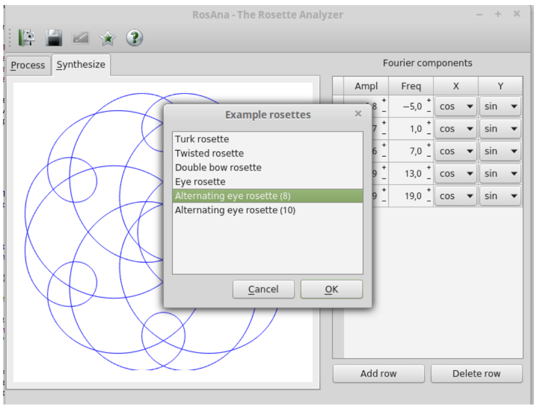
(b) RosAna, Fourier analysis result


RosAna - The Rosette Analyzer

Fourier components

	Ampl	Freq	X	Y
1	2,9	−1,0	cos	sin
2	6,7	3,0	cos	sin

(c) RosAna, synthesize mode


RosAna - The Rosette Analyzer

Fourier components

Ampl	Freq	X	Y
8	−5,0	cos	sin
7	1,0	cos	sin
6	7,0	cos	sin
9	13,0	cos	sin
9	19,0	cos	sin

Example rosettes

Turk rosette
Twisted rosette
Double bow rosette
Eye rosette
Alternating eye rosette (8)
Alternating eye rosette (10)

(d) RosAna, load example rosettes

Figure 4.2: RosAna screenshots

In addition, the following shortcuts are available:

Alt+p	Switch to Process mode
Alt+s	Switch to Synthesize mode
Ctrl+m	Maximize window
Ctrl+Shift+m	Normal size window
Esc	Escape dialog (if applicable)
Alt+F4	Exit application

4.2.1 RosAna input and output formats

RosAna reads input from a file of example rosettes, and write output on request to two .txt files. By following the formats as described below, new example rosettes can be added to `exampleValues.txt`. Note that spaces, commas and line breaks are important. $\ldots$ and $\vdots$ are purely for visualization.

Example rosettes

name, a1, f1, a2, f2, …
name, a1, f1, a2, f2, …
$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$

where *name* is the name of the rosette, and *ai, fi* the Rosette pattern description parameters.

Saved pattern trace

x1, y1
x2, y2
$\vdots, \quad \vdots$

where *xi, yi* are the normalized coordinates for each trace point.

Saved parameters a1, f1, a2, f2, …

where *ai, fi* are the Rosette pattern description parameters.

Chapter 5

Results

To test the performance of RosAna, two types of tests were executed. Rosette analysis was run on select rope rosettes, the results are in Section 5.1. Some benchmarks were also computed, namely machine resource use and Fourier analysis runtime (see Section 5.2).

5.1 Rosette analysis

The performance of RosAna was tested using several different rosettes. Figure 5.1 shows the results for four different rosettes; Turk's head, Closed Twist, Open Twist and Ratan. The second column displays the plot results after analysis. Column three displays the plots after small parameter adjustment. Nearly all adjustments were either *a)* negative frequencies; *b)* negative amplitudes; or *c)* adding more parameters. Option *c)* consisted of adding parameters that were left out in the parameter selection. The added values were based on the Fourier analysis results as seen in Figure 4.2b. The Fourier results and parameters from Figure 5.1 can be found in Appendix B.

[1]Wikipedia Commons, 2013.
[2]Wikipedia Commons, 2008.
[3]Andersson, 2015.
[4]Rossing, 2012.

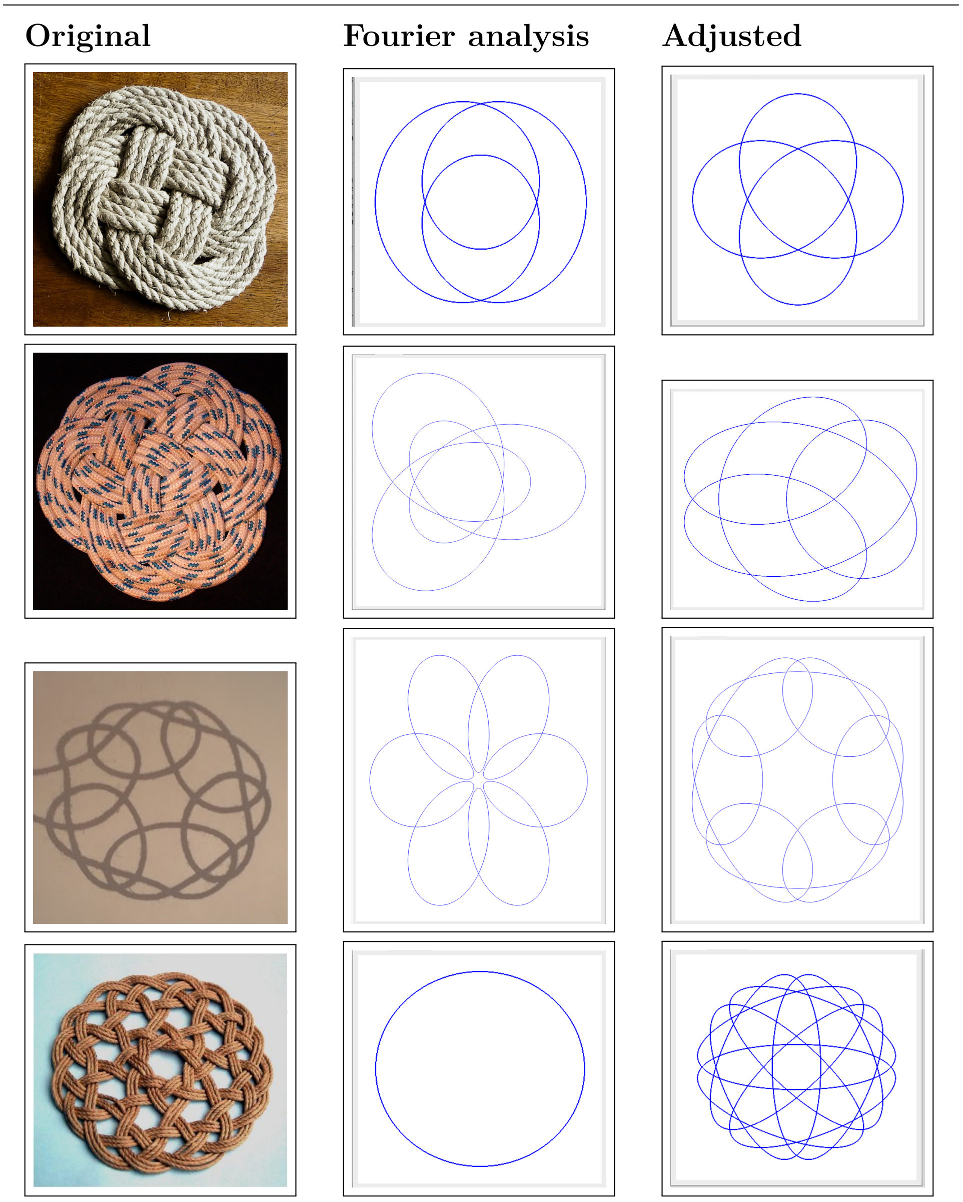


Figure 5.1: Analysis of selected rope rosettes.
Top to bottom: Turk's head[1], Closed Twist[2], Open Twist[3] and Ratan[4].
Parameter values are in Appendix B

5.2 Benchmarks

Testing the performance of RosAna was done in two areas; machine resource and runtime for the Fourier analysis.

All test were performed on a Lenovo E550 ThinkPad running Linux Mint 17.2 Rafaela. The machine sports an Intel® Core™ i7-5500U CPU 2.40GHz processor and has 8 GB of RAM.

5.2.1 Machine resource usage

Resource statistics were gathered every second using `ps aux` in the terminal.

Statistics were recorded during a typical run of RosAna; opening a new image, image tracing, Fourier analysis, pattern synthesis and saving the trace, plot and parameters to file.

	CPU	RAM
Average	1,31%	1,05%
Peak	3,9%	2,9%

Table 5.1: Machine resource usage

5.2.2 Fourier analysis runtime

The most computationally heavy section of RosAna is the Fourier analysis. To test the performance, Fourier analysis was tested with 3, 10, 50, 100, 250, 500, 1000, 1500 and 2000 sample points. For each set of sample points, 100 runs were made, without displaying graphics. The results can be found in Table 5.2 and Figure 5.2.

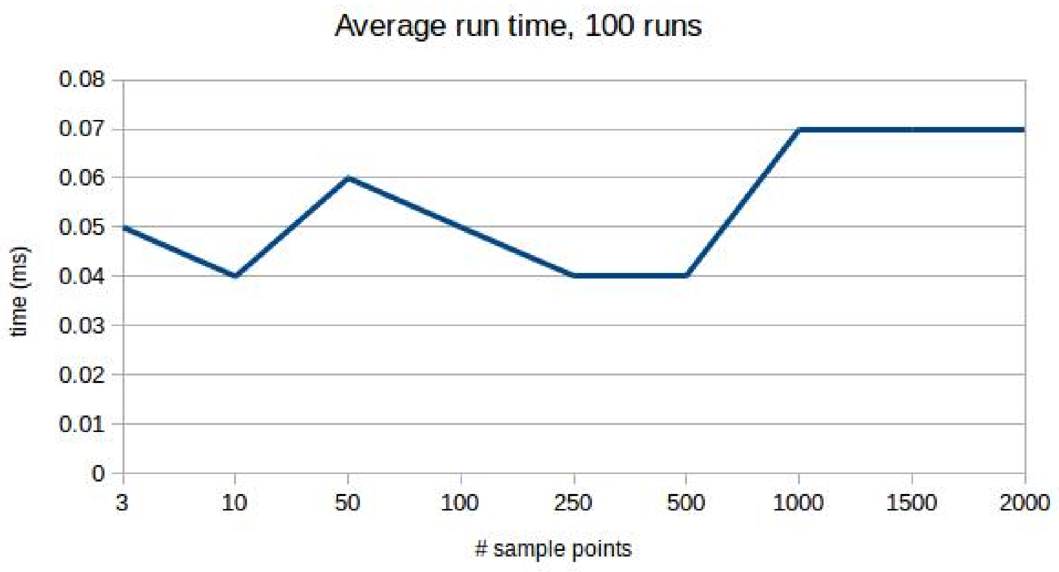


Figure 5.2: Plot of average run times, see Table 5.2

Average runtime (100 runs)	
# points	**Time**
3	0,05 ms
10	0,04 ms
50	0,06 ms
100	0,05 ms
250	0,04 ms
500	0,04 ms
1000	0,07 ms
1500	0,07 ms
2000	0,07 ms

Table 5.2: Average run time is ms

Chapter 6

Evaluation and conclusion

RosAna works fairly well. While simple, manual pattern tracing is robust as to image quality and rosette complexity. The Fourier analysis successfully shows which Rosette pattern description parameters are important. The discrepancy between the original and the synthesized rosettes are caused when automatic parameter selection fails. After studying the values of Section B.1, three causes for failure can be determined. It is to be noted that the adjustments made to correct the parameters were fairly simple.

Frequency selection does not extract all the important frequencies. Studying the results in Section B.2, frequency selection fails when the Fourier results are different in x- and y-direction. This is likely due to non-perfect rosettes, as a rosette made of rope is not likely to be perfectly symmetrical. This could be mitigated by allowing more user input in the analysis process.

Detection of negative frequencies does work some times, as with the Closed Twist rosette. The method of Subsection 3.3.2 was developed from values gained from Rossing's Fourier analyzer. It is therefore possible that RosAna's values are different and the method needs some calibration.

The final failure point is the identification of negative amplitudes. Currently, there is no good way of identifying the negative amplitudes. More work is needed in this area.

The most prominent point of failure was identification of negative amplitudes. This was expected, as no such method has been implemented. Disregarding errors due to negative frequencies, three of the rosettes were synthesized correctly. The Ratan rosette had additional errors due to key frequencies not being selected.

When it comes to performance, the results from RosAna are good. Machine resource usage remains low, even on peak demand. Average runtime is extremely low even for a significant number of sample points, running at max 0,07 ms. In fact, average run times are most likely inaccurate, due to limitations in the timer accuracy. However, for normal usage of RosAna, the number of sample points is not likely to exceed 200, nor is it needed to make the analysis more accurate.

RosAna has some limitations. All possible rosettes are not describable in RosAna, as it does not allow for different A_i values in x- and y-direction. Example rosettes cannot deviate from the standard description of Equation 2.1, specifically the trigonometric functions. Rosette pattern extraction is limited to hand-trace only.

6.1 Contributions

In this thesis, I have presented an integrated tool to analyze and synthesize rope rosettes. To my knowledge, no other such tool exists. Rope rosette description and theory has been gathered in this thesis based on Rossing's previous work (Rossing & Krifel, 2003; Rossing, 2012). The procedure of his Fourier analysis has been documented. The Fourier analysis is not specific for rope rosettes, and can be applied to any problem with some form of signal processing. Some extensions to the mathematical theory has been made, although they are not perfect.

6.2 Future work

RosAna has several areas for improvement and added features. Some points are highly mathematical in nature.

Automatic pattern extraction As of today, RosAna does not extract rosette patterns automatically. The vision is for RosAna to do pattern extraction automatically, using image processing techniques. Possible difficulties include dealing with texture, shadows and poor image quality.

Rosette description mathematics More work is needed on Rosette pattern description parameter selection, specifically in identifying negative amplitudes and making frequency selection more robust. Currently, there is no mathematical theory on how to identify a negative amplitude.

3D model Synthesizing the rosette pattern as a 3D model, possibly including texture. This also include determining the crossing points of the rosette to properly offset the ropes of the model.

Simple user interface Enable the user to select properties like rosette type, slotting, skipping and the number of bights and calculate Rosette pattern description parameters from that.

Pattern instructions A new feature for users to realize the rosette with rope. These instructions may include: *i)* indicating on the pattern if the rope goes over or under on each crossing; and *ii*) determination of rope length and rope thickness based on the size of the rosette and the number of runs.

3D printer Integrating the tool with 3D printing software. Printing rope rosettes have already been accomplished by Regueiro (Regueiro, 2012).

Glossary

Abbreviations

CCW counter-clockwise.

CPU Central Processing Unit.

CW clockwise.

GB GigaBytes.

GHz GigaHertz.

GUI Graphical User Interface.

IDI Department of Computer and Information Science.

IPTK Image Processing Tool Kit.

MVC Model-View-Controller.

NTNU Norwegian University of Science and Technology.

RAM Random Access Memory.

RosAna Rosette Analyzer.

Nomenclature

Model-View-Controller Design pattern in software architecture.

Rope Rosette A type of circular mat made of rope laid out in a specific pattern.

Rope Rosette Analyzer The software tool developed by Andersson as a Masters thesis. This document contains documentation of the tool.

Rosette pattern description the parameters A_i and f_i the describes a rosette pattern according to Equation 2.1.

Software

C++ Programming language ~ Standard C++ Foundation (2016).

Doxygen A tool for generating documentation from annotated sources ~ van Heesch (2015).

GeoGebra mathematical visualization tool ~ International GeoGebra Institute (2016).

Image Processing Tool Kit Image processing algorithms for MATLAB®.

LaTeX typesetting software ~ Lamport (1994).

MATLAB® Scientific programming language and editor ~ Mathworks (2016).

OpenCV Programming library for image processing and computer vision Originally made for C++ but can also be used with Python. ~ itseez (2016).

Python Programming language ~ Python Software Foundation (2016).

Qt GUI framework for cross-platform application development ~ The Qt Company (2015b).

Visual Paradigm tool to draw UML figures ~ Visual Paradigm (2016).

Winplot A Windows program to plot graphs ~ Parris (2012).

Symbols

A_i Amplitude of the i-th component.

A_{xi} Amplitude of the i-th component in the x-direction.

A_{yi} Amplitude of the i-th component in the y-direction.

$\uplus$ Add to set.

f_i Frequency of the i-th component.

f_{xi} Frequency of the i-th component in the x-direction.

f_{yi} Frequency of the i-th component in the y-direction.

Bibliography

References

Andersson, S. (2015). *Analyzing rope rosettes using image processing techniques* (specialisation thesis, Norwegian University of Science and Technology). Under the supervision of Theoharis Theoharis, IDI, NTNU.

Besl, P. J. & Jain, R. C. (1986). Invariant surface characteristics for 3d object recognition in range images. *Computer vision, graphics, and image processing*, *33*(1), 33–80.

Bhakar, S., Dudek, C. K., Muise, S., Sharman, L., Hortop, E., & Szabo, F. (2004). Textiles, patterns and technology: digital tools for the geometric analysis of cloth and culture. *Textile: The Journal of Cloth and Culture*, *2*(3), 308–327.

Browne, C. (2005). Cantor knots. *Computers & Graphics*, *29*(6), 998–1003.

Canny, J. (1986). A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6), 679–698.

Chen, G. (1995). Edge detection by regularized cubic b-spline fitting. *Systems, Man and Cybernetics, IEEE Transactions on*, *25*(4), 636–643.

Chiw, C., Kindlmann, G., Reppy, J., Samuels, L., & Seltzer, N. (2012). Diderot: a parallel DSL for image analysis and visualization. In *Acm sigplan notices* (Vol. 47, *6*, pp. 111–120). ACM.

Comaniciu, D. & Meer, P. (2002). Mean shift: a robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *24*(5), 603–619.

Egmont-Petersen, M., de Ridder, D., & Handels, H. (2002). Image processing with neural networks—a review. *Pattern recognition*, *35*(10), 2279–2301.

Fung, K. (2007). Celtic knot generator. *BSc (Hons) thesis, University of Bath.*

Gerdes, P. (2007). African sona, mirror curves and lunda-designs. *Forma*, *22*(1), 129–131.

Gonzalez, R. C. (2009). *Digital image processing.* Pearson Education India.

Grünbaum, B. & Shephard, G. (1992). Interlace patterns in islamic and moorish art. *Leonardo*, 331–339.

Ji, L. & Yan, H. (2002). Robust topology-adaptive snakes for image segmentation. *Image and Vision Computing*, *20*(2), 147–164.

Kass, M., Witkin, A., & Terzopoulos, D. (1988). Snakes: active contour models. *International journal of computer vision*, *1*(4), 321–331.

Kittler, J. & Illingworth, J. (1985). Relaxation labelling algorithms—a review. *Image and Vision Computing*, *3*(4), 206–216.

Kovesi, P. (1999). Image features from phase congruency. *Videre: Journal of computer vision research*, *1*(3), 1–26.

Li, W. & Gilbert, C. D. (2002). Global contour saliency and local colinear interactions. *Journal of neurophysiology*, *88*(5), 2846–2856.

Mallick, S. (2015). OpenCV(C++ vs Python) vs MATLAB for Computer Vision. Retrieved January 11, 2016, from http://www.learnopencv.com/opencv-c-vs-python-vs-matlab-for-computer-vision/

Martin, D. R., Fowlkes, C. C., & Malik, J. (2004). Learning to detect natural image boundaries using local brightness, color, and texture cues. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *26*(5), 530–549.

Nagata, S. (2007). Digitalization and analysis of traditional cycle patterns in the world, and their contemporary applications. *Forma*, *22*(1), 119–126.

Ostromoukhov, V. (1998). Mathematical tools for computer-generated ornamental patterns. *Electronic Publishing, Artistic Imaging, and Digital Typography*, 193–223.

Otsu, N. (1975). A threshold selection method from gray-level histograms. *Automatica*, *11*(285-296), 23–27.

Pal, N. R. & Pal, S. K. (1993). A review on image segmentation techniques. *Pattern recognition*, *26*(9), 1277–1294.

Papari, G. & Petkov, N. (2011). Edge and line oriented contour detection: state of the art. *Image and Vision Computing*, *29*(2), 79–103.

Purkayasth, S., Dingliana, J., & Stalley, R. (2014). A new approach to analysis of interlace in the book of kells.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S. P., & Durand, F. (2012). Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. 31*(4), 32. Retrieved from http://people.csail.mit.edu/jrk/halide12/

Raghupathy, K. & Parks, T. W. (2004). Improved curve tracing in images. In *Icassp*.

Regueiro, M. D. (2012). Mathematical art galleries - manuel diaz regueiro. Retrieved January 25, 2016, from http://gallery.bridgesmathart.org/exhibitions/2012-bridges-conference/mdregueiro

Rossing, N. K. (2012). *How to make rope mats and rosettes.*

Rossing, N. K. & Krifel, C. (2003). *Matematisk beskrivelse av taumatter* (2nd ed.). NTNU-trykk.

Sargin, M. E., Altinok, A., Rose, K., & Manjunath, B. (2007). Tracing curvilinear structures in live cell images. In *Image processing, 2007. icip 2007. ieee international conference on* (Vol. 6, pp. VI–285). IEEE.

Steger, C. (1998). An unbiased detector of curvilinear structures. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *20*(2), 113–125.

The Qt Company. (2015a). LanguageBindings. Retrieved January 11, 2016, from http://wiki.qt.io/Category:LanguageBindings

Trahanias, P. E. & Venetsanopoulos, A. N. (1996). Vector order statistics operators as color edge detectors. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, *26*(1), 135–143.

Wagemans, J., Elder, J. H., Kubovy, M., Palmer, S. E., Peterson, M. A., Singh, M., & von der Heydt, R. (2012). A century of gestalt psychology in visual perception: i. perceptual grouping and figure–ground organization. *Psychological bulletin*, *138*(6), 1172.

Wang, L., Assadi, A. H., & Spalding, E. (2008). Tracing branched curvilinear structures with a novel adaptive local pca algorithm. In *Ipcv* (pp. 557–563).

Yanagisawa, K. & Nagata, S. (2007). Fundamental study on design system of kolam pattern. *Forma*, *22*(1), 31–46.

Software

Eichhammer, E. (2015). QCustomPlot. Retrieved January 13, 2016, from http://www.qcustomplot.com/

HiB-Mediesenteret. (2004). Rosettverksted. Retrieved June 29, 2015, from http://www.matemania.no/matemania_m/verksted_rosett2/index.html

International GeoGebra Institute. (2016). GeoGebra - Dynamic Mathematics for Everyone. Retrieved January 20, 2016, from https://app.geogebra.org/

itseez. (2016). OpenCV. Retrieved January 11, 2016, from http://opencv.org/

Lamport, L. (1994). *LaTeX*. Addison-wesley.

MathWorks. (n.d.). Marker-controlled watershed segmentation. Retrieved June 29, 2015, from http://se.mathworks.com/help/images/examples/marker-controlled-watershed-segmentation.html

Mathworks. (2016). MATLAB - The Language of Technical Computing. Retrieved January 11, 2016, from http://se.mathworks.com/products/matlab/

Mottalli, M. (2012). Integrating OpenCV in Qt GUI applications. Retrieved January 13, 2016, from http://develnoter.blogspot.no/2012/05/integrating-opencv-in-qt-gui.html

Parris, R. (2012). Winplot for Windows 95/98/ME/2K/XP/7. Retrieved January 13, 2016, from http://math.exeter.edu/rparris/winplot.html

Python Software Foundation. (2016). Python. Retrieved January 11, 2016, from https://www.python.org/

Standard C++ Foundation. (2016). Standard C++. Retrieved January 11, 2016, from https://isocpp.org/

The Qt Company. (2015b). Qt. Retrieved January 11, 2016, from http://www.qt.io/

van Heesch, D. (2015). Doxygen. Retrieved February 4, 2016, from http://www.stack.nl/~dimitri/doxygen/

Visual Paradigm. (2016). Visual Paradigm - Free UML Software Design Tool. Retrieved January 20, 2016, from http://www.visual-paradigm.com/solution/freeumldesigntool/

Images

Stardock Design. (2009). WinCustomize - MyColors"Think Green". Retrieved January 27, 2016, from http://www.wincustomize.com/explore/screenshots/24848/

Wikipedia Commons. (2007). Image of celtic knot. Retrieved June 29, 2015, from https://commons.wikimedia.org/wiki/Category:Celtic_knots?uselang=nb#/media/File:Celtic-knot-insquare-transp.png

Wikipedia Commons. (2008). Fancy work platting. Retrieved July 1, 2015, from https://commons.wikimedia.org/wiki/File:Fancy-Work-Platting-6x4_Original.JPG

Wikipedia Commons. (2012). Image of kolam pattern. Retrieved June 29, 2015, from https://commons.wikimedia.org/wiki/File:Kolam-Attur_town-2012-Salem-Tamil_Nadu-11.JPG

Wikipedia Commons. (2013). Bonnet turc. Retrieved July 1, 2015, from https://commons.wikimedia.org/wiki/File:Bonnet_Turc_3x4_%C3%A0_plat.jpg

Appendix A: RosAna file list

Function	**File** (.cpp & .h)	**Responsibility**
Pattern tracing	`processwidget`	Record pattern tracing
Fourier analysis	`fourieranalysis`	Analyse trace and display results
Pattern synthesis	`drawwidget`	Display rosette patterns
GUI	`fourierdisplaywidget`	Display Fourier analysis results
	`helpdialogwidget`	Display help and about information
	`loadexampledialog`	Select example rosette
	`mainwindow`	GUI and setup of the main application window
	`variableadjustwidget`	Adjust rosette parameters
Helper code	`delegate`	GUI settings for `variableadjustwidget`
	`main`	Main application loop
	`readexamplefile`	Read examples from file
External code	`cvimagewidget`	Provide link between OpenCV and Qt (2012)
	`qcustomplot`	Store and display plots (2015)
Extra files	`master.*`	Configuration files from Qt
Resources	`resources.qrc`	Qt resource file
	`exampleValues.txt`	Example rosettes with parameters
	`turk.jpg`	Example rosette image (2013)
	`icons/`	Custom icons (2009)

Table A.1: RosAna file list

Appendix B: Rosette analysis results

B.1 Parameter adjustment

Rosette	**Fourier results**		**Adjusted**	
	A_i	f_i	A_i	f_i
Turk's Head	2,8	1	2,8	1
	6,6	3	6,6	-3
Closed Twist	2,7	-2	2,7	-2
	2,7	1	2,7	1
	6,8	4	-6,8	4
Open Twist	1,9	-5	1,9	-5
	6,8	1	6,8	1
	0,7	13	-0,7	13
	4,8	7	-4,8	7
Ratan	7,4	7	3,9	-7
			-1,4	1
			-1,2	9

Table B.1: Parameter results of Fourier analysis for selected rosettes. See also Figure 5.1

B.2 Fourier plots

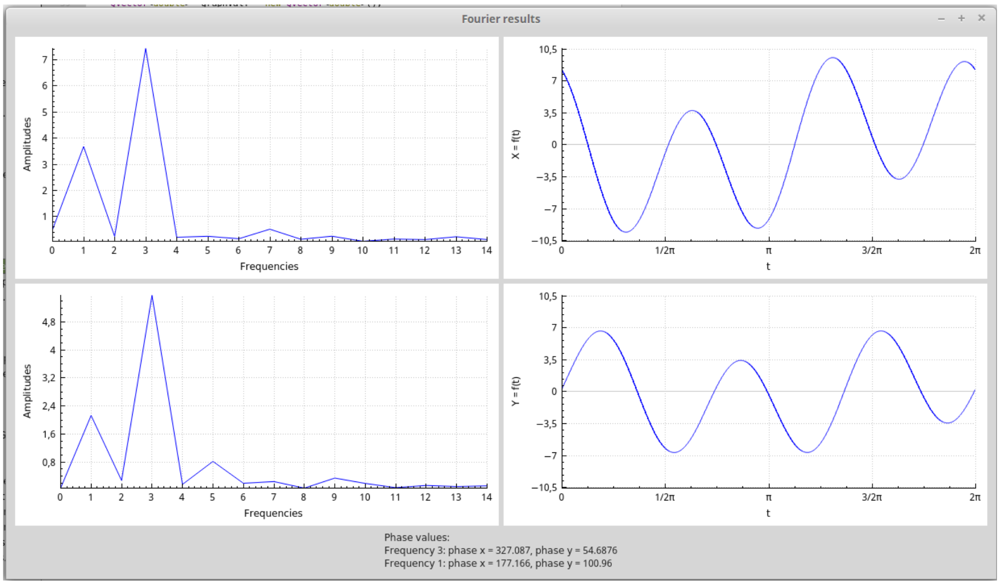
Below follows the Fourier plots from analysis of selected Rope Rosettes. Their layout is as following:

Top-Left: Fourier result, x-direction.

Bottom-Left: Fourier result, y-direction.

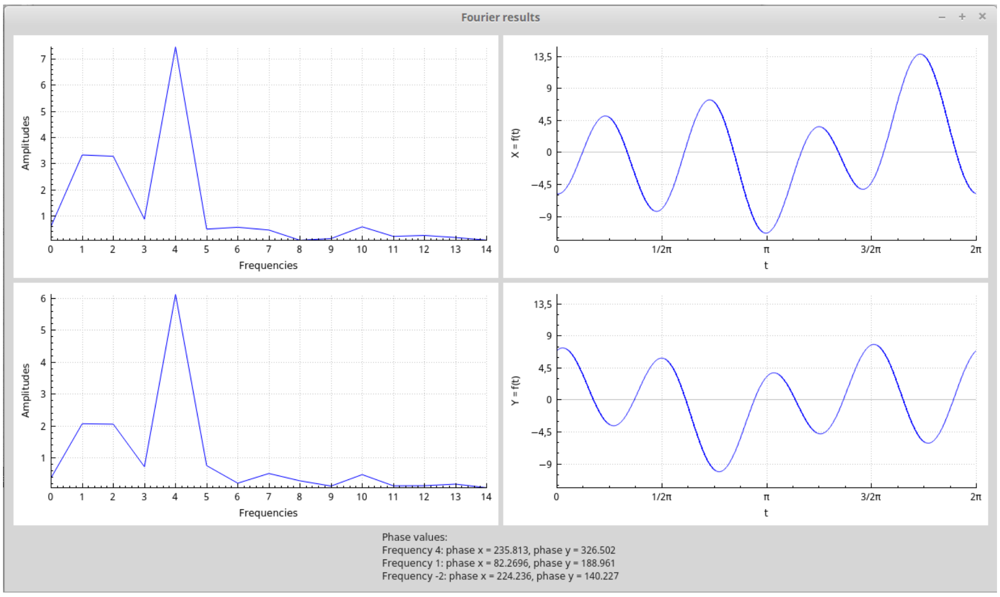
Top-Right: Rosette pattern description Equation 2.1a, x-direction.

Bottom-Right: Rosette pattern description Equation 2.1b, y-direction.



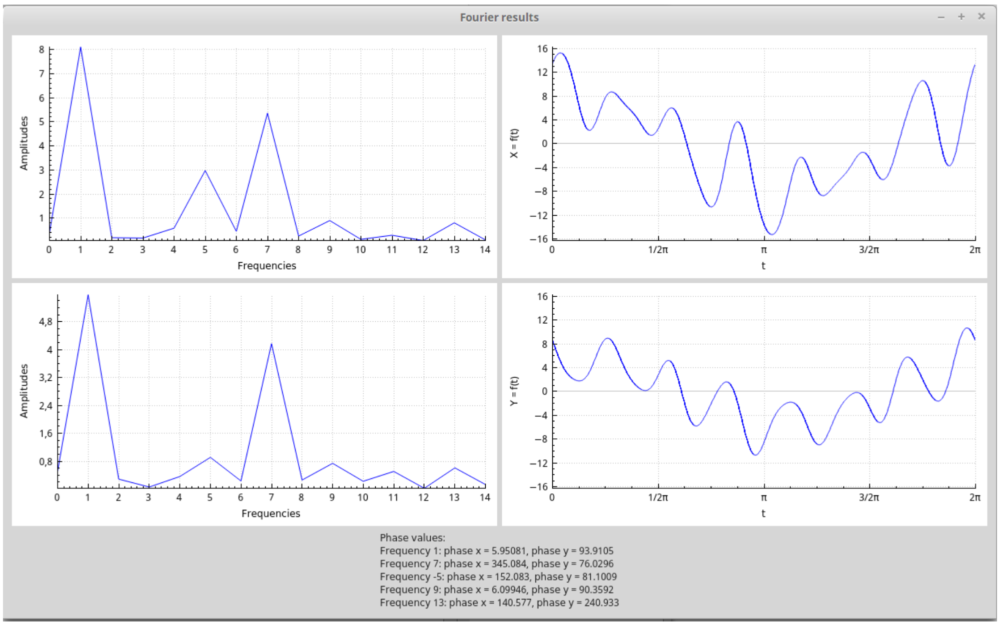
Phase values:
Frequency 3: phase x = 327.087, phase y = 54.6876
Frequency 1: phase x = 177.166, phase y = 100.96

Figure B.1: Fourier analysis results for the Turk's Head rosette



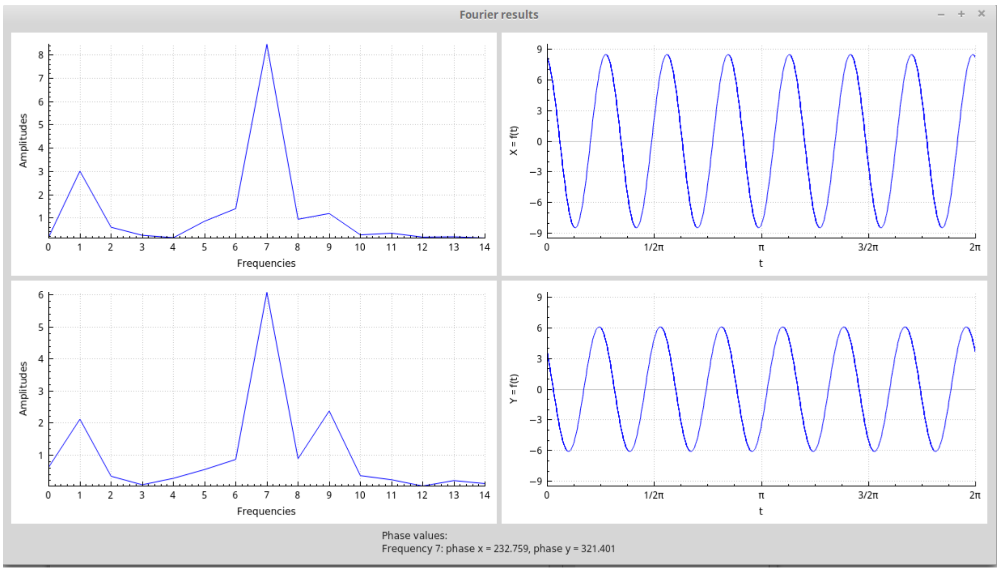
Phase values:
Frequency 4: phase x = 235.813, phase y = 326.502
Frequency 1: phase x = 82.2696, phase y = 188.961
Frequency -2: phase x = 224.236, phase y = 140.227

Figure B.2: Fourier analysis results for the Closed Twist rosette



Phase values:
Frequency 1: phase x = 5.95081, phase y = 93.9105
Frequency 7: phase x = 345.084, phase y = 76.0296
Frequency -5: phase x = 152.083, phase y = 81.1009
Frequency 9: phase x = 6.09946, phase y = 90.3592
Frequency 13: phase x = 140.577, phase y = 240.933

Figure B.3: Fourier analysis results for the Open Twist rosette



Phase values:
Frequency 7: phase x = 232.759, phase y = 321.401

Figure B.4: Fourier analysis results for the Ratan rosette

Appendix C: RosAna documentation

1 Class Documentation

1.1 DrawWidget Class Reference

The DrawWidget class plots a rosette based on the parameters found in its model.

Public Slots

- void itemChanged (**QStandardItem** ∗item)
  Is executed when an item has changed in the model.
- void rowDeleted ()
  Is executed when a row is deleted.

Public Member Functions

- DrawWidget (**QWidget** ∗**parent**=0)
  Constructs the DrawWidget. Sets up customPlot, main graph and layout.
- ∼DrawWidget ()
  Destroys DrawWidget.
- void setModel (**QStandardItemModel** ∗newModel)
  Sets a new model.
- bool saveImage (**QString** filename)
  Saves current plot as image.

Private Member Functions

- void generatePlotData ()
  Generates plot data from model to display with customPlot.
- bool saveParameters (**QString** filename)
  Saves the current parameters to file.

Private Attributes

- QCPCurve ∗ graph
- QCustomPlot ∗ customPlot
- **QStandardItemModel** ∗ model
- **QVector**< double > xdata
- **QVector**< double > ydata
- **QVector**< double > zdata

1.1.1 Detailed Description

The DrawWidget class plots a rosette based on the parameters found in its model.

It inherits QWdiget and makes use of QCustomPlot.

1.1.2 Constructor & Destructor Documentation

1.1.2.1 DrawWidget::DrawWidget (QWidget ∗ *parent* = 0) [explicit]

Constructs the DrawWidget. Sets up *customPlot*, main *graph* and layout.

Parameters

parent	The parent of the widget.

1.1.2.2 DrawWidget::∼DrawWidget ( )

Destroys DrawWidget.

Deletes *customPlot*, *graph* and *model*.

1.1.3 Member Function Documentation

1.1.3.1 void DrawWidget::itemChanged (QStandardItem ∗ *item*) [slot]

Is executed when an item has changed in the model.

Triggers a recalculation of plot data with generatePlotData().

Parameters

item	The changed item, unused.

1.1.3.2 void DrawWidget::rowDeleted ( ) [slot]

Is executed when a row is deleted.

Triggers a recalculation of plot data with generatePlotData().

1.1.3.3 bool DrawWidget::saveImage (QString *filename*)

Saves current plot as image.

Parameters

filename	Name of the saved file.

Returns

`True` if the save is successfull; otherwise `False`.

1.1.3.4 bool DrawWidget::saveParameters (QString *filename*) [private]

Saves the current parameters to file.

Parameters

filename	Name of the saved file.

Returns

`True` if the save is successfull; otherwise `False`.

1.1.3.5 void DrawWidget::setModel (QStandardItemModel ∗ *newModel*)

Sets a new model.

Parameters

newModel	The new model to display.

1.1.4 Member Data Documentation

1.1.4.1 QCustomPlot$*$ DrawWidget::customPlot `[private]`

Displays the rosette plot.

1.1.4.2 QCPCurve$*$ DrawWidget::graph `[private]`

Holds data points for the rosette plot.

1.1.4.3 QStandardItemModel$*$ DrawWidget::model `[private]`

The model containing rosette desccription parameters.

1.1.4.4 QVector$<$double$>$ DrawWidget::xdata `[private]`

List of x-coordinates for the rosette plot.

1.1.4.5 QVector$<$double$>$ DrawWidget::ydata `[private]`

List of y-coordinates for the rosette plot.

1.1.4.6 QVector$<$double$>$ DrawWidget::zdata `[private]`

List of time-coordinates for the rosette plot.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/drawwidget.h
- /home/sigrid/Code/drawwidget.cpp

1.2 FourierAnalysis Class Reference

The FourierAnalysis class handles data adjustment and Fourier analysis of trace data.

Static Public Member Functions

- static **QStandardItemModel** $*$ analyze (**QVector**$<$ **QPointF** $>$ $*$trace)

  Initiates analysis of a sampled and normalized trace.

Static Private Member Functions

- static cv::Mat adjustSamplePoints (**QVector**$<$ **QPointF** $>$ $*$normalizedSamples)

  Adjusts the sample points.
- static void meanAdjust (**QVector**$<$ **QPointF** $>$ $*$samples)

  Adjusts samples by subtracting their mean from all points.
- static **QStandardItemModel** $*$ fourier (cv::Mat samples)

  Performs Fourier analysis on the sample points.
- static **QStandardItemModel** $*$ buildModel (cv::Mat values, **QVector**$<$ int $>$ indexes)

  Builds a model of the detected parameters.
- static **QVector**$<$ int $>$ adjustValues (**QVector**$<$ int $>$ indexes, cv::Mat angles)

  FourierAnalysis::adjustValues.

1.2.1 Detailed Description

The FourierAnalysis class handles data adjustment and Fourier analysis of trace data.

1.2.2 Member Function Documentation

1.2.2.1 cv::Mat FourierAnalysis::adjustSamplePoints (QVector< QPointF > ∗ *normalizedSamples*) `[static]`, `[private]`

Adjusts the sample points.

Parameters

normalized↩ Samples	The sample points.

Returns

adjusted sample points in a cv::Mat.

1.2.2.2 QStandardItemModel ∗ FourierAnalysis::analyze (QVector< QPointF > ∗ *trace*) `[static]`

Initiates analysis of a sampled and normalized trace.

Parameters

trace	New sample points.

Returns

a new model with the results of the Fourier analysis.

1.2.2.3 QStandardItemModel ∗ FourierAnalysis::buildModel (cv::Mat *values,* QVector< int > *indexes*) `[static]`, `[private]`

Builds a model of the detected parameters.

Parameters

values	Amplitude values.
indexes	The significant indexes.

Returns

a new model.

1.2.2.4 QStandardItemModel ∗ FourierAnalysis::fourier (cv::Mat *samples*) `[static]`, `[private]`

Performs Fourier analysis on the sample points.

Parameters

samples	Adjusted sample points.

Returns

a model of the detected paramters.

1.2.2.5 void FourierAnalysis::meanAdjust (QVector< QPointF > ∗ *samples*) `[static]`, `[private]`

Adjusts samples by subtracting their mean from all points.

Parameters

samples	Sample points to be adjusted.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/fourieranalysis.h
- /home/sigrid/Code/fourieranalysis.cpp

1.3 FourierDisplayWidget Class Reference

The FourierDisplayWidget class displays results from FourierAnalysis.

Static Public Member Functions

- static void displayResults (cv::Mat amplitudes, **QVector**$<$ int $>$ indexes, cv::Mat angles)

  Displays the results from FourierAnalysis.

1.3.1 Detailed Description

The FourierDisplayWidget class displays results from FourierAnalysis.

1.3.2 Member Function Documentation

1.3.2.1 void FourierDisplayWidget::displayResults (cv::Mat *amplitudes*, QVector$<$ int $>$ *indexes*, cv::Mat *angles*) `[static]`

Displays the results from FourierAnalysis.

Parameters

amplitudes	Amplitudes from the Fourier analysis.
indexes	Indexes of significant components.
angles	Angles from the Fourier analysis.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/fourierdisplaywidget.h
- /home/sigrid/Code/fourierdisplaywidget.cpp

1.4 HelpDialogWidget Class Reference

The HelpDialogWidget class displays a dialog which containts a user guide for this tool, some rosette theory, and author and licence information.

Public Member Functions

- HelpDialogWidget (**QWidget** $*$**parent**=0)

  Constructs HelpDialogWidget.

1.4.1 Detailed Description

The HelpDialogWidget class displays a dialog which containts a user guide for this tool, some rosette theory, and author and licence information.

1.4.2 Constructor & Destructor Documentation

1.4.2.1 HelpDialogWidget::HelpDialogWidget (QWidget ∗ *parent* = 0) [explicit]

Constructs HelpDialogWidget.

Parameters

parent	The parent widget.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/helpdialogwidget.h
- /home/sigrid/Code/helpdialogwidget.cpp

1.5 LoadExampleDialog Class Reference

The LoadExampleDialog class displays the available example rope rosettes.

Signals

- void selectedExample (int selectedRow)

Public Member Functions

- LoadExampleDialog (**QStringList** names, **QWidget** ∗**parent**=0)

  Constructs LoadExampleDialog.
- ∼LoadExampleDialog ()

  Detroys LoadExampleDialog.

Private Slots

- void accept () Q_DECL_OVERRIDE

  Triggered when an example is selected and LoadExampleDialog closed.

Private Attributes

- **QListWidget** ∗ nameList

1.5.1 Detailed Description

The LoadExampleDialog class displays the available example rope rosettes.

It inherits **QDialog**. Examples are loaded through ReadExampleFile. When an example is selected, the model for plot and paramters is changed to model of the selected example rosette.

1.5.2 Constructor & Destructor Documentation

1.5.2.1 LoadExampleDialog::LoadExampleDialog (QStringList *names,* QWidget ∗ *parent* = 0) [explicit]

Constructs LoadExampleDialog.

Sets up the LoadExampleDialog GUI and connects signals for user interaction.

Parameters

names	A list of named example rosettes.
parent	The parent of LoadExampleDialog.

1.5.3 Member Function Documentation

1.5.3.1 void LoadExampleDialog::accept ( ) [private], [slot]

Triggered when an example is selected and LoadExampleDialog closed.

Gets the *index* of the selected example rosette. Emits signal *selectedExample(index)*.

1.5.3.2 LoadExampleDialog::selectedExample (int *selectedRow*) [signal]

This signal is emitted when an example has been selected.

Parameters

selectedRow	The index of the selected row.

1.5.4 Member Data Documentation

1.5.4.1 QListWidget∗ LoadExampleDialog::nameList [private]

< list of names for available example rosettes

The documentation for this class was generated from the following files:

- /home/sigrid/Code/loadexampledialog.h
- /home/sigrid/Code/loadexampledialog.cpp

1.6 MainWindow Class Reference

The MainWindow class is the main window of RosAna.

Public Member Functions

- MainWindow (**QWidget** ∗**parent**=0)

  Contructs MainWindow. Sets up Gui and interactions.
- ∼MainWindow ()

  Destroys MainWindow. Deletes exampleValues.

Private Slots

- void helpDialog ()

  Executes HelpDialogWidget.
- void loadDialog ()

  Executes LoadExampleDialog.
- void setModels (int index)

  Gets the selected model from exampleValues and sets the new model for drawGraphWidget and variableAdjustment↩ Widget.
- void tracedModel (**QStandardItemModel** ∗model)

  Is executed when a new model has been generated from Fourier analysis.
- void setStatusBar (**QString** text)

Is executed when a signal is triggered with a message for the status bar.
- void clearStatusBar ()

  Is executed when a signal is triggered to clear the status bar.
- void openImage ()

  Is executed when openImageAction is triggered. Opens an open file dialog.
- void saveImage ()

  Is executed when saveImageAction is executed. Opens a save file dialog.
- void updateActionBar (int index)

  Updates the action bar icons. Is executed when the active tab is changed.

Private Member Functions

- void setupToolbar ()

  Sets up the main toolbar, adds actions and shortcuts.
- void setShortcuts ()

  Adds extra shortcuts to MainWindow.

Private Attributes

- DrawWidget ∗ drawGraphWidget
- HelpDialogWidget ∗ helpWidget
- LoadExampleDialog ∗ loadExampleDialog
- ProcessWidget ∗ processImage
- ReadExampleFile ∗ exampleValues
- VariableAdjustWidget ∗ variableAdjustmentWidget

1.6.1 Detailed Description

The MainWindow class is the main window of RosAna.

Setup of Gui, interactions and connections is done in this class.

QTabWidget:

- ProcessWidget *processImage*.
- DrawWidget *drawGraphWidget*.

QDockWidget:

- VariableAdjustWidget *variableAdjustmentWidget*.

Dialogs:

- LoadExampleDialog *loadExampleDialog*.
- HelpDialogWidget *helpWidget*.

Helpers:

- ReadExampleFile *exampleValues*.

1.6.2 Constructor & Destructor Documentation

1.6.2.1 MainWindow::MainWindow (**QWidget** ∗ *parent* = `0`) `[explicit]`

Contructs MainWindow. Sets up Gui and interactions.

Parameters

parent	The parent widget.

1.6.3 Member Function Documentation

1.6.3.1 void MainWindow::saveImage ( ) `[private]`, `[slot]`

Is executed when saveImageAction is executed. Opens a save file dialog.

Detects the current mode and forwards the save trigger to either ProcessWidget or DrawWidget

1.6.3.2 void MainWindow::setModels (int *index*) `[private]`, `[slot]`

Gets the selected model from *exampleValues* and sets the new model for *drawGraphWidget* and *variable*↩ *AdjustmentWidget*.

Parameters

index	The selected index

1.6.3.3 void MainWindow::setStatusBar (QString *text*) `[private]`, `[slot]`

Is executed when a signal is triggered with a message for the status bar.

Parameters

text	The text to display.

1.6.3.4 void MainWindow::tracedModel (QStandardItemModel ∗ *model*) `[private]`, `[slot]`

Is executed when a new model has been generated from Fourier analysis.

Sets the new model for DrawWidget and VariableAdjustWidget. Sets the current mode to synthesize.

Parameters

model	The model resulting from Fourier analysis.

1.6.3.5 void MainWindow::updateActionBar (int *index*) `[private]`, `[slot]`

Updates the action bar icons. Is executed when the active tab is changed.

Parameters

index	The current tab index.

1.6.4 Member Data Documentation

1.6.4.1 DrawWidget∗ MainWindow::drawGraphWidget `[private]`

The widget for synthesis mode.

1.6.4.2 ReadExampleFile∗ MainWindow::exampleValues `[private]`

Class for reading example file.

1.6.4.3 HelpDialogWidget∗ MainWindow::helpWidget `[private]`

Dialog for user guide and about.

1.6.4.4 LoadExampleDialog∗ MainWindow::loadExampleDialog [private]

Dialog for selecting which example rosette to load.

1.6.4.5 ProcessWidget∗ MainWindow::processImage [private]

The widget for process mode.

1.6.4.6 VariableAdjustWidget∗ MainWindow::variableAdjustmentWidget [private]

Widget for editing of rosette parameters.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/mainwindow.h
- /home/sigrid/Code/mainwindow.cpp

1.7 ProcessWidget Class Reference

The ProcessWidget class handles image display and pattern tracing.

Public Slots

- void manualTrace ()

  Initiates manual trace.

Signals

- void manualTraceInProgress (**QString** text)

  This signal is emitted once a trace has been initiated.
- void traceDone (**QStandardItemModel** ∗traceModel)

  This signal is emitted when the trace is completed.

Public Member Functions

- ProcessWidget (**QWidget** ∗**parent**=0)

  Constructs ProcessWidget.
- ∼ProcessWidget ()

  Destroys ProcessWidget.
- bool saveTrace (**QString** filepath)

  Saves a normalized pattern trace to disk.
- bool openImage (const **QString** &filepath)

  Loads an image from disk.

Private Member Functions

- cv::Point convertPoint (**QPoint** old)

  Converts between Qt::Qpoint and cv::Point.
- void drawLine (cv::Point start, cv::Point end)

  Draws a line on traceImg.
- void keyPressEvent (**QKeyEvent** ∗ev)

  Captures key press events.
- void mousePressEvent (**QMouseEvent** ∗ev)

Records mouse clicks if a trace is active.

- **QVector**< **QPointF** > ∗ normalize (**QVector**< **QPoint** > ∗samples)

  Normalizes trace points.

1.7.1 Detailed Description

The ProcessWidget class handles image display and pattern tracing.

1.7.2 Constructor & Destructor Documentation

1.7.2.1 ProcessWidget::ProcessWidget (**QWidget** ∗ *parent* = 0) `[explicit]`

Constructs ProcessWidget.

Reads an example file and sets up gui elements.

Parameters

parent	The parent widget.

1.7.2.2 ProcessWidget::∼ProcessWidget ( )

Destroys ProcessWidget.

Releases resources held by *img* and *traceImg*.

1.7.3 Member Function Documentation

1.7.3.1 Point ProcessWidget::convertPoint (**QPoint** *old*) `[private]`

Converts between Qt::Qpoint and cv::Point.

Parameters

old	**QPoint** to be converted.

Returns

a cv::Point.

1.7.3.2 void ProcessWidget::drawLine (cv::Point *start*, cv::Point *end*) `[private]`

Draws a line on *traceImg*.

Parameters

start	Start point.
end	End point.

1.7.3.3 void ProcessWidget::keyPressEvent (**QKeyEvent** ∗ *ev*) `[private]`, `[virtual]`

Captures key press events.

Using the Enter key triggers a stop to the current trace. The trace is normalized and passed on to FourierAnalysis↩ ::analyze(QVector<QPointF>)

Parameters

ev	Key press event.

Reimplemented from **QWidget**.

1.7.3.4 void ProcessWidget::manualTraceInProgress (QString *text*) `[signal]`

This signal is emitted once a trace has been initiated.

Parameters

text	The message

1.7.3.5 void ProcessWidget::mousePressEvent (QMouseEvent ∗ *ev*) `[private]`, `[virtual]`

Records mouse clicks if a trace is active.

Parameters

ev	the mouse event.

Reimplemented from **QWidget**.

1.7.3.6 QVector< QPointF > ∗ ProcessWidget::normalize (QVector< QPoint > ∗ *samples*) `[private]`

Normalizes trace points.

Parameters

samples	The trace to be normalized.

Returns

a normalized trace on the range [0,1].

1.7.3.7 bool ProcessWidget::openImage (const QString & *filepath*)

Loads an image from disk.

Parameters

filepath	The image file path.

Returns

`True` if the load is successful; otherwise `False`.

1.7.3.8 bool ProcessWidget::saveTrace (QString *filepath*)

Saves a normalized pattern trace to disk.

Format: x1, y1, \n x2, y2, \n etc.

Parameters

filepath	The image file path.

Returns

`True` if the save is successful; otherwise `False`.

1.7.3.9 void ProcessWidget::traceDone (QStandardItemModel ∗ *traceModel*) `[signal]`

This signal is emitted when the trace is completed.

Parameters

traceModel	The model resulting from the Fourier Analysis.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/processwidget.h
- /home/sigrid/Code/processwidget.cpp

1.8 ReadExampleFile Class Reference

The ReadExampleFile class reads example rosettes from file.

Public Member Functions

- ReadExampleFile ()

  Constructs an instance of ReadExampleFile and calls readFile().
- ∼ReadExampleFile ()

  Destroys ReadExampleFile and deleted values.
- **QStringList** fetchNames ()

  returns the names of example rosettes.
- **QStandardItemModel** ∗ getModel (int index)

  Returns a model of an example rosette.

Private Member Functions

- void readFile ()

  reads the file of example rosettes.

Private Attributes

- **QStringList** nameList
- **QVector**< **QStringList** > ∗ values

1.8.1 Detailed Description

The ReadExampleFile class reads example rosettes from file.

Stores the values in a model such that other classes can get the model

1.8.2 Member Function Documentation

1.8.2.1 QStringList ReadExampleFile::fetchNames ( )

returns the names of example rosettes.

Returns

a list of example rosette names.

1.8.2.2 QStandardItemModel ∗ ReadExampleFile::getModel (int *index*)

Returns a model of an example rosette.

Parameters

index	of which rosette example to load.

Returns

a new model containing the parameters of the example rosette.

1.8.2.3 void ReadExampleFile::readFile ( ) `[private]`

reads the file of example rosettes.

The examples are on the format "name, a1, f1, a2, f2, ... Values are stored in *nameList* and *values*.

1.8.3 Member Data Documentation

1.8.3.1 QStringList ReadExampleFile::nameList `[private]`

A list of paramter values for example rosettes.

1.8.3.2 QVector<QStringList>∗ ReadExampleFile::values `[private]`

A list of example rosette names.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/readexamplefile.h
- /home/sigrid/Code/readexamplefile.cpp

1.9 VariableAdjustWidget Class Reference

The VariableAdjustWidget class holds the GUI for rosette parameters.

Signals

- void rowDeleted ()

Public Member Functions

- VariableAdjustWidget (**QWidget** ∗**parent**=0)

  Constructs VariableAdjustWidget.
- ∼VariableAdjustWidget ()

  Destroys VariableAdjustWidget, deletes viewDelegate.
- void setModel (**QStandardItemModel** ∗model)

  Sets a new model for parametersTableView and mapper.

Private Slots

- void addRow ()

  Is executed when addRowButton is clicked. Adds a new row to the internal model.
- void deleteRow ()

  Is executed when deleteRowButton is clicked. Gets the selected index from parametersTableView and deletes the corresponding row in its internal model. Emits signal rowDeleted()

Private Attributes

- VariableEditDelegate ∗ viewDelegate
- **QDataWidgetMapper** ∗ mapper
- **QPushButton** ∗ addRowButton
- **QPushButton** ∗ deleteRowButton
- **QTableView** ∗ parametersTableView

1.9.1 Detailed Description

The VariableAdjustWidget class holds the GUI for rosette parameters.

The VariableAdjustWidget class inherits **QWidget**. Parameters are set from a model shared with DrawWidget.

1.9.2 Constructor & Destructor Documentation

1.9.2.1 VariableAdjustWidget::VariableAdjustWidget (**QWidget** ∗ *parent* = 0) [explicit]

Constructs VariableAdjustWidget.

Initialize all GUI elements:

- **QTableView** parametersTableView
- VariableEditDelegate viewDelegate
- **QDataWidgetMapper** mapper
- **QPushButton** addRowButton
- **QPushButton** deleteRowButton Connect signals and set layout

Parameters

parent	The parent of VariableAdjustWidget.

1.9.3 Member Function Documentation

1.9.3.1 VariableAdjustWidget::rowDeleted ( ) [signal]

This signal is emitted when a row has been deleted from the model.

1.9.3.2 void VariableAdjustWidget::setModel (**QStandardItemModel** ∗ *newModel*)

Sets a new model for *parametersTableView* and *mapper*.

Parameters

newModel	The new model to display

1.9.4 Member Data Documentation

1.9.4.1 **QPushButton**∗ VariableAdjustWidget::addRowButton [private]

Adds a row to *paramteresTableView* and the model.

1.9.4.2 **QPushButton**∗ VariableAdjustWidget::deleteRowButton [private]

Deleted a row from *paramtersTableView* and the model.

1.9.4.3 QDataWidgetMapper∗ VariableAdjustWidget::mapper [private]

Maps model values to strings.

1.9.4.4 QTableView∗ VariableAdjustWidget::parametersTableView [private]

Displays the model parameters as a editable table.

1.9.4.5 VariableEditDelegate∗ VariableAdjustWidget::viewDelegate [private]

Provides an interface between *parametersTableview* and the model.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/variableadjustwidget.h
- /home/sigrid/Code/variableadjustwidget.cpp

1.10 VariableEditDelegate Class Reference

Handles display of model data for VariableAdjustWidget.

Public Member Functions

- **QWidget** ∗ createEditor (**QWidget** ∗**parent**, const **QStyleOptionViewItem** &option, const **QModelIndex** &index) const Q_DECL_OVERRIDE

  Creates the editor based on index.
- void setEditorData (**QWidget** ∗editor, const **QModelIndex** &index) const Q_DECL_OVERRIDE

  Sets editor data from model. Triggered when the model is changed.
- void setModelData (**QWidget** ∗editor, **QAbstractItemModel** ∗model, const **QModelIndex** &index) const Q↩_DECL_OVERRIDE

  Sets model data from editor. Triggered when an editor is changed.
- void updateEditorGeometry (**QWidget** ∗editor, const **QStyleOptionViewItem** &option, const **QModelIndex** &index) const Q_DECL_OVERRIDE

  Handles editor styling.

1.10.1 Detailed Description

Handles display of model data for VariableAdjustWidget.

This class is heavily based on the example `SpinBoxDelegate` provided by the **Qt** framework.

1.10.2 Member Function Documentation

1.10.2.1 QWidget ∗ VariableEditDelegate::createEditor (QWidget ∗ *parent*, const QStyleOptionViewItem & *option*, const QModelIndex & *index*) const [virtual]

Creates the editor based on index.

Parameters

parent	The parent widget.
index	The index of the current element.

option	Editor option (unused).

Returns

the apropriate editor.

Reimplemented from **QStyledItemDelegate**.

1.10.2.2 void VariableEditDelegate::setEditorData (**QWidget** ∗ *editor,* const **QModelIndex** & *index*) const `[virtual]`

Sets editor data from model. Triggered when the model is changed.

Parameters

editor	The editor to provide with data.
index	The index of the data.

Reimplemented from **QStyledItemDelegate**.

1.10.2.3 void VariableEditDelegate::setModelData (**QWidget** ∗ *editor,* **QAbstractItemModel** ∗ *model,* const **QModelIndex** & *index*) const `[virtual]`

Sets model data from editor. Triggered when an editor is changed.

Parameters

editor	The editor source.
model	The model to edit.
index	The index to edit.

Reimplemented from **QStyledItemDelegate**.

1.10.2.4 void VariableEditDelegate::updateEditorGeometry (**QWidget** ∗ *editor,* const **QStyleOptionViewItem** & *option,* const **QModelIndex** & *index*) const `[virtual]`

Handles editor styling.

Parameters

editor	The editor to style.
option	Styling option.
index	The data index (unused).

Reimplemented from **QStyledItemDelegate**.

The documentation for this class was generated from the following files:

- /home/sigrid/Code/delegate.h
- /home/sigrid/Code/delegate.cpp