MIRIAM MARX

NUMERICAL METHODS FOR DEFLECTOMETRY

NUMERICAL METHODS FOR DEFLECTOMETRY

MIRIAM MARX

Master Thesis

Chair of Digital Image Processing
Computer Science
University of Passau

Professor: Prof. Dr. Tomas Sauer
October 2017

ABSTRACT

Deflectometry is attracting increasing interest during the fabrication of cars. In the course of this, it is used to detect automatically if there is any kind of error on a reflexive surface, e.g., in the car finish or on glass. To improve the accuracy, it is important to solve the problem of backside reflection. The backside reflection occurs on glass, when only parts of the incoming rays are reflected right on top of the surface. The remaining rays can enter the surface and are reflected on it's back side. The camera, which monitors the reflected rays, can not differentiate between the different kinds of reflection, because they overlay each other. To provide a precise result of the deflectometry it is crucial to determine which part belongs to the front- or backside reflection.

This master thesis focuses on solving the problem of backside reflection and provides two different approaches. The first approach creates a system of equations using exactly three measurements.

As opposed to this, the second approach is based on Prony's method and allows an arbitrary number of samples, but at least four samples are required. This arbitrary number of samples is especially a big advantage if the data contains noise. In this case, the number of samples is increased to provide a more accurate result.

In summary, we were able to solve the problem of backside reflection using Prony's method. If no noise is present both approaches perform almost the same and are able to decompose the measurements sufficiently. However, if the measurements contain noise the second approach outperforms the first one and provides a much more accurate decomposition. The first approach is not able to handle noise sufficiently.

ACKNOWLEDGMENTS

I would like to thank my friends, family and especially my boyfriend for supporting me during the whole time of my master thesis and for reading the thesis. Furthermore, I would like to thank Prof. Dr. Tomas Sauer, Dr. Alexander Zimmermann and Florian Waschbichler for their professional advises.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

PMD Phase Measuring Deflectometry

ESPRIT Estimation of Signal Parameters via rotation invariance Techniques

SVD Singular Value Decomposition

TV Total Variation

GPU Graphical Processing Unit

Part I

OVERVIEW

1 INTRODUCTION

When we purchase a brand-new car we expect it to be delivered without any defects. Beside functional abilities, this also applies for its appearance. For example, there should not be a scratch in the finish or glass, which is used for the windows or spotlights. However, during the fabrication of cars many steps are automated using different machines, but even those can make mistakes. To determine if a machine made a mistake regarding the appearance of a car specialized employees have been used to inspect the car for any optical error up to now. However, humans are not able to provide a constant high standard of error detection, because their performance differs due to personal factors like concentration. Hence, it is necessary to provide an error detection system which can guarantee an overall high accuracy.

A commonly used method for automated detection of geometrical errors on reflexive surfaces is **deflectometry**. Here, a visual signal is emitted from a screen and reflected from the surface. Those reflections are recorded by cameras and used to locate the error [6]. Hence, this method attracts increasing interest for the fabrication of cars.

However, glass mostly consists of a reflexive surface, but is also permeable for light rays. Usually, if a light ray is reflected on a reflexive surface, the reflection only occurs directly on top of a surface. Hence, we call this kind of reflection **frontside reflection**. However, this is not the only type of reflection which can occur on glass. It is also possible for rays to enter the surface and be reflected on its backside. In this case, it is not possible to differentiate between the different reflections because the rays, which are emitted with different angles, overlay each other. That problem is known as **backside reflection**. This overlay of multiple angles can compromise the result of deflectometry. Hence, it is crucial for a productive usage to develop a method for splitting up these two different kinds of reflections.

1.1 TASK OF THIS MASTER THESIS

Until now, it is not possible to differentiate between the different reflections from the front- and backside. Hence, this master thesis aims to find a solution for backside reflection by splitting up the overlapping reflections algorithmically into its front- and backside. In order to do so, it is crucial to apply the developed algorithms to different

kinds of input data to simulate real-world conditions, e.g., by adding different levels of noise.

Finally, we want to determine, whether it is possible to use our proposed procedure for productive usage during the fabrication of cars.

1.2 STRUCTURE OF THIS MASTER THESIS

This master thesis is divided into four main parts. The first part gives an overview on the topic and introduces some theoretical background regarding deflectometry. The second part consists of the first approach which was applied to solve the backside reflection. Its theory was mainly developed by Alexander Zimmermann from FORWISS [10]. The approach is based on the idea to create a system of equations for a fixed set of three measurements. This system provides a solution for our problem. However, exactly three measurements limits the accuracy, because we believe more measurements provide better results especially when the data is noisy. This leads to part three describing the second approach. The second approach takes usage of an arbitrary number of measurements to improve the accuracy explicitly if noise is contained. This approach is based on Prony's method and aims to reconstruct the front- and backside reflection from the overlapping measurements. The last part evaluates the different results and provides a final statement regarding improvements.

2

THEORETICAL BACKGROUND

This chapter will introduce some theoretical background regarding deflectometry.

2.1 DEFLECTOMETRY

Deflectometry is a common method to determine geometrical errors on reflecting surfaces. It basically determines the gradient of the surface which is to be inspected. In doing so, the monitor in Figure 1 shows a predefined pattern. This pattern will be reflected at the surface and the distorted pattern will be recorded by the camera. With different decoding mechanisms it is possible to create a mapping $\hat{I}$ between a certain pixel on the monitor and the camera. This information can be used to describe the reflecting surface [6].

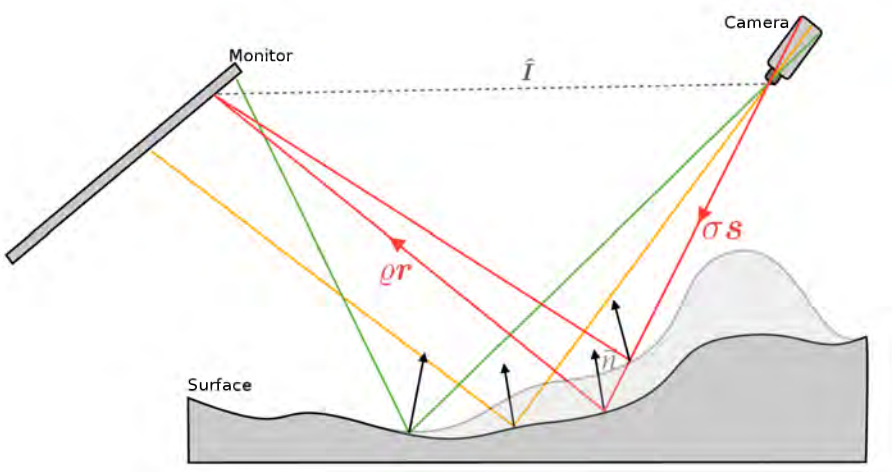

Figure 1: Scheme of Deflectometry[6].

2.2 PHASE MEASURING DEFLECTOMETRY

In the following, we will have a look at the Phase Measuring Deflectometry (PMD) which is a specialization of the deflectometry. Here, a screen emits sinusoidal fringe pattern which will be reflected by the surface. If there is any variation in the gradient of the surface, the pattern, which will be monitored by the camera, will be distorted. These distortions can be determined using different phase-shift algorithms. Figure 2 shows a scheme of this method [2].

The PMD is also applied in our special use case. Hence, we will only talk about the PMD in the following.

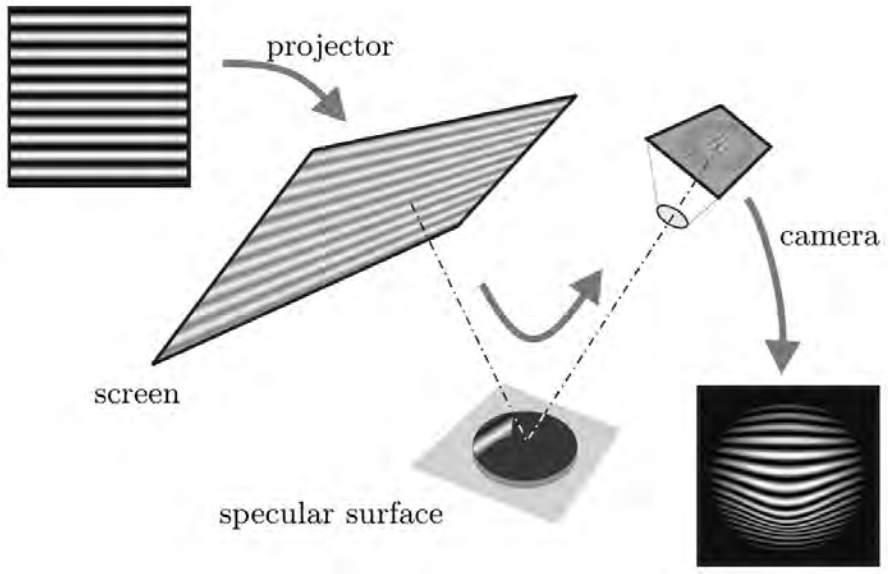

Figure 2: Scheme of PMD [2].

Part II

FIRST APPROACH USING A SYSTEM OF EQUATIONS

3

MATHEMATICAL BACKGROUND

This chapter will introduce the mathematical background, which is used to solve the problem of backside reflection by creating a system of equations. The proposed procedure was mainly developed by Alexander Zimmermann from FORWISS [10].

All angles, which are handled in the following, are elements of the torus $\mathbb{T} = \mathbb{R}/2\pi\mathbb{Z}$.

3.1 INFORMATION FROM THE RECORDING

During deflectometry a sinusoidal fringe pattern is displayed on a monitor. This monitor is moved over the reflecting car finish to monitor the reflection by a camera. The monitoring contains only the amplitudes c_j and phases γ_j for $j \in \{1,2,3\}$. j was chosen to be in $\{1,2,3\}$, because three measurements allow to create a system of equations of degree 2[1], which is *relatively easy* to solve. One measurement consists of one phase and one amplitude coming from one specific wavelength. In order to provide multiple measurements the wavelengths are altered.

In general, for $n \in \mathbb{N}$ measurements of the same spot with exactly one backside reflection, the following relation holds:

$$c_j \cdot \sin(x + \gamma_j) = a \cdot \sin(x + \alpha_j) + b \cdot \sin(x + \beta_j) \tag{1}$$

with

$$c_j = \sqrt{a^2 + b^2 + 2 \cdot a \cdot b \cdot \cos(\alpha_j - \beta_j)} \tag{2}$$

and $a, b, c_j, \alpha_j, \beta_j, \gamma_j \in \mathbb{R}$.

Hence, it is required to

1. calculate a, b and $\xi_j = \alpha_j - \beta_j$ from formula (2) and
2. to calculate the values of α_j and β_j from formula (1).

The sinusoidal functions, which will be used for our measurements, only differ in their wavelengths λ_j by

$$\frac{\lambda_1}{\lambda_j} = j \quad \textbf{for } j = 2,3. \tag{3}$$

1 see formula (11)

These wavelengths are used to calculate

$$\alpha_j = \frac{2\pi j f_x}{\lambda_1} \quad \textbf{or} \quad \alpha_j = \frac{2\pi f_x}{\lambda_j} \tag{4}$$

and

$$\beta_j = \frac{2\pi j f_y}{\lambda_1} \quad \textbf{or} \quad \beta_j = \frac{2\pi f_y}{\lambda_j} \tag{5}$$

with f_x and f_y are function values in the x or y coordinate. The calculation of these function values is explained in section 4.1.

Moreover, formula (4) and (5) fulfill

$$\alpha_j \equiv j \cdot \alpha_1 \mod 2\pi \quad \textbf{and} \quad \beta_j \equiv j \cdot \beta_1 \mod 2\pi \quad \textbf{with } j = 2, 3. \tag{6}$$

From this, a relation for $\xi_j = \alpha_j - \beta_j$ can be derived:

$$\xi_j \equiv i \cdot \xi_1 \mod 2\pi \quad \textbf{with } j = 2, 3. \tag{7}$$

Using formula (7) and the double/triple angle formulae, we get:

$$\cos(2\xi) = 2 \cdot \cos^2(\xi) - 1 \tag{8}$$

and

$$\cos(3\xi) = 4 \cdot \cos^3(\xi) - 3 \cdot \cos(\xi). \tag{9}$$

3.2 CALCULATION OF a, b AND ξ

Before we can determine α and β, we have to determine a, b and ξ first.

Therefore, we will use

$$\cos(\xi_j) = t_j \tag{10}$$

and create the system of equations with formula (2), (8) and (9):

$$\begin{aligned} c_1^2 &= a^2 + b^2 + 2\,a\,b\,t_1 \\ c_2^2 &= a^2 + b^2 + 2\,a\,b\,(2\,t_1^2 - 1) \\ c_3^2 &= a^2 + b^2 + 2\,a\,b\,(4\,t_1^3 - 3\,t_1) \end{aligned} \tag{11}$$

to determine the three unknowns a, b and t_1.

If $c_1^2 \neq c_2^2$ and $t_1 \neq -\frac{1}{2}$ and $t_1 \neq 1$ then the difference of the third and second equation can be divided by the difference of the

second and first equation. In this case, we get a quadratic equation of the unknown t_1:

$$\begin{aligned}
& & \frac{c_3^2 - c_2^2}{c_2^2 - c_1^2} &= \frac{4t_1^3 - 2t_1^2 + 3t_1 + 1}{2t_1^2 - t_1 - 1} \\
& \Leftrightarrow & \frac{c_3^2 - c_2^2}{c_2^2 - c_1^2} &= \frac{(4\,t_1^2 + 2\,t_1 - 1)\cdot(t_1 - 1)}{(2\,t_1 + 1)\cdot(t - 1)} \\
& \Leftrightarrow & \frac{c_3^2 - c_2^2}{c_2^2 - c_1^2}\,(2\,t_1 + 1) &= 4\,t_1^2 + 2\,t_1 - 1 \qquad (12) \\
\tilde{c} = \frac{c_3^2 - c_2^2}{c_2^2 - c_1^2} & \Leftrightarrow & 2\cdot\tilde{c}\cdot t_1 + \tilde{c} &= 4\,t_1^2 + 2\,t_1 - 1 \\
& & 4t_1^2 + 2t_1(1 - \tilde{c}) - (1 + \tilde{c}) &= 0. \qquad (13)
\end{aligned}$$

Solving this quadratic equation leads to two possible solutions for t_1, depending on a and b. If

$$a \geqslant b \geqslant 0 \qquad (14)$$

then the correct t_1 is found.

If $c_1^2 = c_2^2$ then $2\,a\,b\,(2\,t_1^2 - t_1 - 1) = 0$. This leads to $b = 0$ or $t_1 = 1$ or $t_1 = -\frac{1}{2}$. If $b = 0$ or $t_1 = 1$ then $c_1^2 = c_3^2$, which means there is no backside reflection or the front- and backside reflections are identical.

If $t_1 = -\frac{1}{2}$ then there is a reduced quadratic equation:

$$c_1^2 = a^2 + b^2 - a\,b \qquad c_3^2 = a^2 + b^2 + 2\,a\,b = (a + b)^2, \qquad (15)$$

which can be again solved by condition (14).

In general, it is only possible to determine the cosine $t_1 = \cos(\xi_1)$. Hence, the solution is not unique, which means beside (a, b, ξ_1) (the solution of the original deflectometric problem) also $(a, b, -\xi_1)$, $(b, a, \pm\xi_1)$, $(-a, -b, \pm\xi_1)$ and $(-b, -a, \pm\xi_1)$ are possible solutions.

It is not possible to erase this ambiguousness by an increasing number of measurements, because the equations will always depend on $\cos(\xi_1)$.

3.3 DETERMINATION OF α AND β

Until now, we determined $t_1 = \cos(\xi_1)$, a and b using the amplitudes c_1, c_2 and c_3. The next step, consists of using the phases γ_1, γ_2 and γ_3 to calculate possible solutions for α and β.

3.3.1 *Determining a formula for γ*

Now, we want to directly link the provided γ to α and β to derive a formula which determines α and β.

The theorem of addition for the sine function

$$\sin(x+y) = \sin(x)\cos(y) + \cos(x)\sin(y) \tag{16}$$

and the representation in polar-coordinates

$$\binom{u}{v} = \sqrt{u^2+v^2} \cdot \begin{pmatrix} \cos(\arctan(\frac{v}{u})) \\ \sin(\arctan(\frac{v}{u})) \end{pmatrix} \tag{17}$$

leads to

$$\begin{aligned} u\cdot\sin(x)+v\cdot\cos(x) &\overset{(17)}{=} \sqrt{u^2+v^2}\cdot\cos\left(\arctan\left(\frac{v}{u}\right)\right)\cdot\sin(x) \\ &\quad +\sqrt{u^2+v^2}\cdot\sin\left(\arctan\left(\frac{v}{u}\right)\right)\cdot\cos(x) \\ &= \sqrt{u^2+v^2}(\cos\left(\arctan\left(\frac{v}{u}\right)\right)\cdot\sin(x) \\ &\quad +\sin\left(\arctan\left(\frac{v}{u}\right)\right)\cdot\cos(x)) \\ &= \sqrt{u^2+v^2}\cdot\sin\left(x+\arctan\left(\frac{v}{u}\right)\right). \end{aligned}$$

With this, we get

$$\begin{aligned} &a\cdot\sin(x+\alpha)+b\cdot\sin(x+\beta) = \\ &\overset{(16)}{=} a\cdot(\sin(x)\cdot\cos(\alpha)+\cos(x)\cdot\sin(\alpha)) \\ &\quad +b\,(\sin(x)\cdot\cos(\beta)+\cos(x)\cdot\sin(\beta)) \\ &= (a\cdot\cos(\alpha)+b\cdot\cos(\beta))\cdot\sin(x) \\ &\quad +(a\cdot\sin(\alpha)+b\cdot\sin(\beta))\cdot\cos(x) \\ &\overset{(17)}{=} \sqrt{(a\cdot\cos(\alpha)+b\cdot\cos(\beta))^2+(a\cdot\sin(\alpha)+b\cdot\sin(\beta))^2} \\ &\quad \cdot\sin\left(x+\arctan\left(\frac{a\cdot\sin(\alpha)+b\cdot\sin(\beta)}{a\cdot\cos(\alpha)+b\cdot\cos(\beta)}\right)\right) \\ &= \sqrt{a^2+b^2+2\cdot a\cdot b\cdot\cos(\alpha-\beta)} \\ &\quad \cdot\sin\left(x+\arctan\left(\frac{a\cdot\sin(\alpha)+b\cdot\sin(\beta)}{a\cdot\cos(\alpha)+b\cdot\cos(\beta)}\right)\right) \end{aligned} \tag{18}$$

Formula (1) and (2) make it now possible to conclude

$$\gamma_j = \arctan\left(\frac{a\cdot\sin(\alpha_j)+b\cdot\sin(\beta_j)}{a\cdot\cos(\alpha_j)+b\cdot\cos(\beta_j)}\right) \tag{19}$$

$$\begin{aligned} \Leftrightarrow \tan(\gamma_j) &= \frac{a\cdot\sin(\alpha_j)+b\cdot\sin(\beta_j)}{a\cdot\cos(\alpha_j)+b\cdot\cos(\beta_j)} \\ \Leftrightarrow \frac{\sin(\gamma_j)}{\cos(\gamma_j)} &= \frac{a\cdot\sin(\alpha_j)+b\cdot\sin(\beta_j)}{a\cdot\cos(\alpha_j)+b\cdot\cos(\beta_j)}. \end{aligned} \tag{20}$$

Now, we have derived a direct correlation between γ and α and β, which will be used in the following.

3.3.2 *Calculation of possible αs and βs*

With the previously determined formula (19) for γ and its reformulation (20) it is possible to create the following system of equations

$$\begin{aligned} f_j \cdot \sin(\gamma_j) &= a \cdot \sin(\alpha_j) + b \cdot \sin(\beta_j) \\ f_j \cdot \cos(\gamma_j) &= a \cdot \cos(\alpha_j) + b \cdot \cos(\beta_j) \\ \xi_j &= \alpha_j - \beta_j. \end{aligned} \tag{21}$$

This system consists of the three unknowns α_j, β_j and a factor f_j. To solve this system the addition theorems of sine and cosine are used in the first step.

$$\begin{aligned} f_j \cdot \cos(\gamma_j) &\overset{\beta_j=\alpha_j-\xi_j}{=} (a + b \cdot \cos(\xi_j)) \cdot \cos(\alpha_j) \\ &\quad + b \cdot \sin(\xi_j) \cdot \sin(\alpha_j) \\ f_j \cdot \sin(\gamma_j) &\overset{\beta_j=\alpha_j-\xi_j}{=} -b \cdot \sin(\xi_j) \cdot \cos(\alpha_j) \\ &\quad + (a + b \cdot \cos(\xi_j)) \cdot \sin(\alpha_j) \\ &\iff \\ f_j \cdot \begin{pmatrix} \cos(\gamma_j) \\ \sin(\gamma_j) \end{pmatrix} &= \begin{pmatrix} a+b\cos(\xi_j) & b\sin(\xi_j) \\ -b\sin(\xi_j) & a+b\cos(\xi_j) \end{pmatrix} \begin{pmatrix} \cos(\alpha_j) \\ \sin(\alpha_j) \end{pmatrix} \\ &\overset{t_j=\cos(\xi_j)}{\iff} \\ f_j \cdot \begin{pmatrix} \cos(\gamma_j) \\ \sin(\gamma_j) \end{pmatrix} &= \underbrace{\begin{pmatrix} a+b\,t_j & \pm b\sqrt{1-t_j^2} \\ \mp b\sqrt{1-t_j^2} & a+b\,t_j \end{pmatrix}}_{M_{t_j}} \begin{pmatrix} \cos(\alpha_j) \\ \sin(\beta_j) \end{pmatrix} \\ &\iff \\ \frac{1}{f_j} \cdot \begin{pmatrix} \cos(\alpha_j) \\ \sin(\alpha_j) \end{pmatrix} &= M_{t_j}^{-1} \begin{pmatrix} \cos(\gamma_j) \\ \sin(\gamma_j) \end{pmatrix} \\ &= \frac{1}{\det(M_{t_j})} \cdot \underbrace{M_{t_j}^T \cdot \begin{pmatrix} \cos(\gamma_j) \\ \sin(\gamma_j) \end{pmatrix}}_{z} \\ &\iff \\ \alpha_j &= \arctan\left(\frac{z_2}{z_1}\right) \\ \textbf{with} \quad M_{t_j}^{-1} &= \frac{M_{t_j}^T}{\det(M_{t_j})} \quad \textbf{and} \quad M_{t_j}^T = M_{-t_j}. \end{aligned} \tag{22, 23}$$

Now, this formula gives an explicit definition of α depending on γ, a, b and ξ.

3.3.3 *Choosing the correct αs and βs*

Until now, we determined a formula for α, which only depends on $t = \cos(\xi)$. However, it has to be taken into account that cosine is an even function, which means $\cos(\xi) = \cos(-\xi)$. Hence there are always two possible solutions for every ξ:

a) $\xi_1 = \pm\arccos(t_1)$ with $t_1 = t$ $\quad \begin{pmatrix}\cos(\xi_1)\\ \sin(\xi_1)\end{pmatrix} = \begin{pmatrix} t_1 \\ \pm\sqrt{1-t_1^2}\end{pmatrix}$

b) $\xi_2 = \pm\arccos(t_2)$ with $t_2 = 2\,t^2 - 1$ $\quad \begin{pmatrix}\cos(\xi_2)\\ \sin(\xi_2)\end{pmatrix} = \begin{pmatrix} t_2 \\ \pm\sqrt{1-t_2^2}\end{pmatrix}$

c) $\xi_3 = \pm\arccos(t_3)$ with $t_3 = 4\,t^3 - 3\,t$ $\quad \begin{pmatrix}\cos(\xi_3)\\ \sin(\xi_3)\end{pmatrix} = \begin{pmatrix} t_3 \\ \pm\sqrt{1-t_3^2}\end{pmatrix}$.

We choose

$$M_{t_j} = \begin{pmatrix} a+b\,t_j & b\sqrt{1-t_j^2} \\ -b\sqrt{1-t_j^2} & a+b\,t_j \end{pmatrix} \textbf{ for } i = 1,2,3. \tag{24}$$

The ambiguity in the sign of $b\sqrt{1-t_1^2}$ in formula (22) is covered by M_{t_j} in formula (27) and by its transposition in formula (25).

Now, it is possible to calculate two candidates for each α and β using

$$\alpha_j^{(1)} = \arctan(z_2, z_1) \quad \text{with} \quad z = M_{t_j}^T \begin{pmatrix}\cos(\gamma_j)\\ \sin(\gamma_j)\end{pmatrix} \tag{25}$$

$$\beta_j^{(1)} = \alpha_j^{(1)} - \arctan\left(\sqrt{1-t_j^2}, t_j\right) \tag{26}$$

$$\alpha_j^{(2)} = \arctan(z_2, z_1) \quad \text{with} \quad z = M_{t_j} \begin{pmatrix}\cos(\gamma_j)\\ \sin(\gamma_j)\end{pmatrix} \tag{27}$$

$$\beta_j^{(2)} = \alpha_j^{(2)} + \arctan\left(\sqrt{1-t_j^2}, t_j\right). \tag{28}$$

This leads to 8 possible solutions for α and β:

1. $\alpha = (\alpha_1^1, \alpha_2^1, \alpha_3^1)$, $\quad \beta = (\beta_1^1, \beta_2^1, \beta_3^1)$
2. $\alpha = (\alpha_1^1, \alpha_2^1, \alpha_3^2)$, $\quad \beta = (\beta_1^1, \beta_2^1, \beta_3^2)$
3. $\alpha = (\alpha_1^1, \alpha_2^2, \alpha_3^1)$, $\quad \beta = (\beta_1^1, \beta_2^2, \beta_3^1)$
4. $\alpha = (\alpha_1^1, \alpha_2^2, \alpha_3^2)$, $\quad \beta = (\beta_1^1, \beta_2^2, \beta_3^2)$
5. $\alpha = (\alpha_1^2, \alpha_2^1, \alpha_3^1)$, $\quad \beta = (\beta_1^2, \beta_2^1, \beta_3^1)$
6. $\alpha = (\alpha_1^2, \alpha_2^1, \alpha_3^2)$, $\quad \beta = (\beta_1^2, \beta_2^1, \beta_3^2)$
7. $\alpha = (\alpha_1^2, \alpha_2^2, \alpha_3^1)$, $\quad \beta = (\beta_1^2, \beta_2^2, \beta_3^1)$
8. $\alpha = (\alpha_1^2, \alpha_2^2, \alpha_3^2)$, $\quad \beta = (\beta_1^2, \beta_2^2, \beta_3^2)$.

To determine which of these solutions is the correct one, formula (6) will be used. The correct solution fulfills the relation in formula (6) best.

4

IMPLEMENTATION

The following sections describe the C++ code which is used to implement the theory of the previous chapter. In doing so, also special libraries provided by FORWISS have been used.

4.1 GENERATION OF TEST IMAGES

In the beginning, we wanted to test our algorithm on *perfect* test images to verify the correctness of the algorithm. Hence, this chapter describes the generation of these images.

Right now, we provide two kinds of test-images - linear and quadratic ones, which means that images are computed using functions of degree one or two. The source code of this procedure can be found in *CExampleFuncQuadratic.h* and *CExampleFuncQuadratic.cpp*. The linear functions are used to simulate use cases, where no error appears on the surface. The quadratic functions are used to simulate errors in the surface. These functions are used the following way:

- Every generation requires two functions (one function handles the x-part and the other the y-part). The constructor of these function is shown in Listing 1 and requires:
  - A constant value a, which will not be modified.
  - Three values `coeff00`,`coeff01` and `coeff02` as coefficients to calculate the function values

$$f = v_2 \cdot x^2 + v_1 \cdot x + v_0$$

- The type of the two functions is defined using template parameters. Thereby, it is easier to enhance the image generation by additional functions.

Listing 1: Constructor of the example function.

```
CExampleFuncQuadratic(double a,double coeff02, double coeff01,
    double coeff00);
```

Now, the entire image-generation looks like the following and is implemented in *CImageGenerator.h* and *CImageGenerator.cpp*:

- Input: an image size, a maximum wavelength ω_{max}, and noise values for the phase and amplitude (but they can be zero as well, if the noise is not wanted)

- In general, three different images will be computed (for the maximum wavelength ω_{max}, $\omega_{max}/2$ and $\omega_{max}/3$).
- For each pixel the following values are calculated:
  - a = constant input value of the example functions depending on x
  - b = constant input value of the example functions depending on y
  - $\alpha = \frac{2 \cdot \pi}{\omega} \cdot$ `exampleFuncX` (using the linear or quadratic function)
  - $\beta = \frac{2 \cdot \pi}{\omega} \cdot$ `exampleFuncY` (using the linear or quadratic function)
  - Temporary values for further calculations:

    $$\begin{aligned} tmpX &= a \cdot \cos(\alpha) + b \cdot \cos(\beta) \\ tmpY &= a \cdot \sin(\alpha) + b \cdot \sin(\beta) \\ tmpAmplitude &= \texttt{hypot}(tmpX, tmpY) \end{aligned}$$

    With `hypot(a,b)` $= \sqrt{a^2 + b^2}$.
  - $amplitude = tmpAmplitude + noiseForAmplitude$ (the noise can be 0)
  - $phase = shiftAngleToPi(atan2(tmpY, tmpX) + noiseForPhase)$ (the noise can be 0 and for `shiftAngleToPi` compare chapter 4.2.1)
- Output:
  - Backup values for $\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3$.
  - Phase and amplitude for each wavelength.

  The phases and amplitudes represent the measured data and are used as only input to the algorithm, which reconstructs the two angles α and β. The Backup values for α_{1-3} and β_{1-3} represent the ground truth and are only used to verify the accuracy of the proposed reconstruction algorithm.

4.2 FURTHER FUNCTIONALITY

This chapter summarizes further functionality that is used for basic calculations or to validate the calculation results.

4.2.1 *The class of safe mathematical functions*

During the calculations, we had to handle numerical inaccuracies. Therefore, some own implementations of common mathematical functions have been provided to handle these edge cases:

- $safeAcos(x)$: Problems occur, if x is slightly bigger than 1 or slightly smaller than -1. In these cases we return $acos(1)$ or $acos(-1)$.
- $safeSqrt(x)$: Problems occur, if x is pretty close to zero, but slightly negative. In these cases we return 0.
- $shiftAngleToPi(x)$: This method shifts an input angle x to the torus $[-\pi, \pi)$.
- $modulo2Pi(x)$: In general it calculates the minimum of $\{r, 2\pi - r\}$ with r is the rest of x divided by 2π.
- $moduloPi(x)$: It does the same as $modulo2Pi(x)$, but instead 2π it uses just π.

The source code of this functionality is provided in *CSafeMathFunctions.h* and *CSafeMathFunctions.cpp*.

4.2.2 *The class of error exports*

To validate the source code, regarding functionality and correctness, we have created the class `CErrorExport` to compare images which have been calculated during the backside reflection against the previously generated images. If the images differ in some points, the class adds this error-case to a csv-file, containing every possible intermediate result to track the source of the error. Therefore, the instance requires knowledge about the backup images, which can be provided by three different constructors:

- The empty constructor `CErrorExport::CErrorExport()` disables an error export.
- The constructor in Listing 2 requires the following parameters:
  - `std::string path` specifies the export path for the finally generated error file. This path is also used to load the backup images of the αs and βs automatically.
  - `double limit` specifies an approximate value for zero, e.g., if the difference of two values is smaller than this `limit` these two values will be considered as equal.
  - All images of the phase and amplitude to provide all intermediate values.
- The constructor in Listing 3 uses the same parameters as the previous constructor. Additionally, it uses all images of the αs and βs directly, which is required if they are not stored in the same directory defined by `path`.

Listing 2: Constructor with phases and amplitudes as only parameters.

```
CErrorExport::CErrorExport(std::string path, double limit,
gray_image_c<double>* phase1,
gray_image_c<double>* phase2,
gray_image_c<double>* phase3,
gray_image_c<double>* amplitude1,
gray_image_c<double>* amplitude2,
gray_image_c<double>* amplitude3)
```

Listing 3: Constructor with all backup images as parameters.

```
CErrorExport::CErrorExport(std::string path, double limit,
gray_image_c<double>* alpha1,
gray_image_c<double>* alpha2,
gray_image_c<double>* alpha3,
gray_image_c<double>* beta1,
gray_image_c<double>* beta2,
gray_image_c<double>* beta3,
gray_image_c<double>* phase1,
gray_image_c<double>* phase2,
gray_image_c<double>* phase3,
gray_image_c<double>* amplitude1,
gray_image_c<double>* amplitude2,
gray_image_c<double>* amplitude3)
```

Afterwards, in every iteration over the pixels, the error file is filled by calling the method `addError(...)` from Listing 4. This method checks if an error occurs and in this case adds the error to the output. The method requires the following parameters:

- An `enum ErrorType` to specify the type of errors, which will be examined. The following error types are currently available:
  - `ALL_ERRORS`: every error (all errors in αs, βs and ts)
  - `ALL_ERRORS_A_B`: all errors in αs, βs
  - `ALL_ERRORS_A`: all errors in αs
  - `ALL_ERRORS_B`: all errors in βs
  - `ALL_ERRORS_A1_B1`: all errors in α_1 and β_1
  - `ALL_ERRORS_A2_B2`: all errors in α_2 and β_2
  - `ALL_ERRORS_A3_B3`: all errors in α_3 and β_3
  - `ALL_ERRORS_T`: all errors in ts
  - `EVERYLINE`: every pixel (no matter if there is an error or not)
- A `bool furtherCondition` to specify further conditions, which have to be fulfilled. Otherwise the error will not be exported. This can be handy if only special pixel ranges in the image are of further interest.

- `int i,int j` to specify the current pixel.
- `double a,double b` to specify the values of a and b.
- All possible values of α_1 `double alpha11,double alpha12` and the chosen α_1 `double alpha1`. The same applies to all remaining values of α and β.
- All values of t and ξ.

Listing 4: Type signatur of the method for adding errors to the error export.

```
void CErrorExport::addError(ErrorType errorType, bool
    furtherCondition,
int i,int j,double a, double b,
double alpha11,double alpha12,double alpha1,
double alpha21,double alpha22,double alpha2,
double alpha31,double alpha32,double alpha3,
double beta11,double beta12,double beta1,
double beta21,double beta22,double beta2,
double beta31,double beta32,double beta3,
double t,double t2,double t3,
double ksi2,double ksi3)
```

4.2.3 *The class of image handling*

The class `CImageHandling` was created to provide general methods to import and export ".big" image files.

4.3 BACKSIDE REFLECTION BY USING TEST IMAGES

This chapter describes the applied algorithm to split up the measured angle ξ into the two angles α and β. The source code can be found in *backside.h* and *backside.cpp*.

The following calculations are done for every pixel and take the amplitudes c_1, c_2, c_3 and phases $\gamma_1, \gamma_2, \gamma_3$ as input parameters. There are three versions of the amplitude and phase, because of the three different wavelength from the image generation process.

Right now only calculations for linear example functions are taken into account.

The calculations rely on the following relations and requirements: For every $j = 2, 3$:

$$\xi_j \equiv j \cdot \xi_1 \bmod 2\pi \tag{29}$$

and

$$\cos(\xi_1) = t_1. \tag{30}$$

Based on the calculation of the amplitudes

$$\begin{aligned} c^2 &= \mathrm{hypot}(\mathrm{tmpX}, \mathrm{tmpY}) \\ &= (a \cdot \cos(\alpha) + b \cdot \cos(\beta))^2 + (a \cdot \sin(\alpha) + b \cdot \sin(\beta))^2 \\ &= a^2 \cdot \cos^2(\alpha) + 2ab \cdot \cos(\alpha) \cdot \cos(\beta) + b^2 \cdot \cos(\beta) \\ &\quad + a^2 \cdot \sin^2(\alpha) + 2ab \cdot \sin(\alpha) \cdot \sin(\beta) + b^2 \cdot \sin(\beta) \\ &= a^2 + b^2 + 2ab \cdot \cos(\alpha - \beta) \\ &= a^2 + b^2 + 2ab \cdot \cos(\xi) \\ &= a^2 + b^2 + 2ab \cdot t \end{aligned}$$

and the behaviour of the t_is (compare formula (40) and (41)), the following system of equations has to be solved:

$$\begin{aligned} c_1^2 &= a^2 + b^2 + 2\,a\,b\,t_1 \\ c_2^2 &= a^2 + b^2 + 2\,a\,b\,(2\,t_1^2 - 1) \\ c_3^2 &= a^2 + b^2 + 2\,a\,b\,(4\,t_1^3 - 3\,t_1). \end{aligned} \tag{31}$$

Where a, b are the constant values in the example functions.

The first step is to solve system (31). Therefore we set

$$c := \frac{c_3^2 - c_2^2}{c_2^2 - c_1^2} \tag{32}$$

and want to solve the quadratic equation

$$4t_1^2 + 2t_1(1 - c) - (1 + c) = 0$$

from formula (13). In general, it is possible to solve this equation using the abc-formula. However, this can be numerical unstable, e.g., if we are subtracting two floating point numbers which are almost equal. Hence, we use Vieta's theorem, which leads us to the relation

$$\begin{aligned} ax^2 + bx + c &= x^2 + \frac{b}{a}x + \frac{c}{a} \\ \Leftrightarrow (x - x_1)(x - x_2) &= x^2 - (x_1 + x_2)x + x_1x_2 \\ \Rightarrow \quad x_1 + x_2 = \frac{b}{a} &\quad \text{and} \quad x_1x_2 = \frac{c}{a}\ [1]. \end{aligned} \tag{33}$$

If $2(1 - c) > 0$ we calculate

$$\mathrm{tTmp}_2 = -\frac{2(1-c) + \sqrt{(2(1-c))^2 + 16(1+c)}}{8} \tag{34}$$

$$\mathrm{tTmp}_1 = \frac{c}{4 \cdot \mathrm{tTmp}_2}. \tag{35}$$

Otherwise, we calculate

$$\mathrm{tTmp}_1 = -\frac{2(1-c) - \sqrt{(2(1-c))^2 + 16(1+c)}}{8} \tag{36}$$

$$\mathrm{tTmp}_2 = \frac{c}{4 \cdot \mathrm{tTmp}_1}. \tag{37}$$

Using those formulas, we can determine possibilities for ab using

$$abTmp_1 = \frac{c_2^2 - c_1^2}{2(2 \cdot tTmp_1^2 - tTmp_1 - 1)} \tag{38}$$

$$abTmp_2 = \frac{c_2^2 - c_1^2}{2(2 \cdot tTmp_2^2 - tTmp_2 - 1)}. \tag{39}$$

Now there are two possibilities for both t and ab. Hence it is required to select the suitable ones by the following rule:

1. **if** $!((c_1^2 - c_2^2) \to 0)$

   if $(abTmp_1 > 0 \quad and \quad abTmp_2 > 0)$ **then** $t_1 = tTmp_2; ab = abTmp_2;$

   else if $(abTmp_1 > 0)$ **then** $t_1 = tTmp_1; ab = abTmp_1;$

   else if $(abTmp_2 > 0)$ **then** $t_1 = tTmp_2; ab = abTmp_2;$

2. **else**

   if $((c_1^2 - c_3^2) \to 0)$ **then** $t_1 = 1; ab = 0;$

   else $t_1 = -0.5; ab = \frac{c_3^2 - c_1^2}{3};$

The rule was evaluated during testing and provides a minimal error rate.

For t_2, t_3, ξ_2 and ξ_3 the following rules apply:

$$t_2 = 2 \cdot t_1^2 - 1 \tag{40}$$

$$t_3 = 4 \cdot t_1^3 - 3 \cdot t_1 \tag{41}$$

$$\xi_2 = \texttt{shiftAngleToPi}(acos(t_2)) \tag{42}$$

$$\xi_3 = \texttt{shiftAngleToPi}(acos(t_3)). \tag{43}$$

After the determination of t_1 and ab, it is possible to calculate two candidates for a and b using the first formula of (31):

$$\begin{aligned}
& a^2 + b^2 + 2abt_1 &&= c_1^2 \\
\Leftrightarrow \quad & a^2 + b^2 + 2abt_1 - c_1^2 &&= 0 \\
\Leftrightarrow \quad & a^2 + \frac{(ab)^2}{a^2} + 2abt_1 - c_1^2 &&= 0 \\
\Leftrightarrow \quad & a^4 + (2abt_1 - c_1^2)a^2 + (ab)^2 &&= 0 \\
\overset{z := a^2}{\Leftrightarrow} \quad & z^2 + (2abt_1 - c_1^2)z + (ab)^2 &&= 0.
\end{aligned} \tag{44}$$

Now, we have to solve formula (44) and extract the square root to get all possibilities for a.

This time, we use again formula (33) to provide more stability to the calculations. However, $(2abt_1 - c_1^2)$ is always smaller or equal to zero, because of

$$2abt_1 - c_1^2 \overset{(2)}{=} 2abt_1 - (a^2 + b^2 + 2abt_1) = -a^2 - b^2 \leqslant 0.$$

Additionally, we do not have to consider the negative results of a, because a has to be positive. This leads to the following:

$$aTmp_1 = \sqrt{-\frac{2abt_1 - c_1^2 - \sqrt{(2abt_1 - c_1^2)^2 - 4(ab)^2}}{2}} \quad (45)$$

$$aTmp_2 = \sqrt{\frac{(ab)^2}{aTmp_1}} \quad (46)$$

$$bTmp_1 = \frac{ab}{aTmp_1} \quad (47)$$

$$bTmp_2 = \frac{ab}{aTmp_2}. \quad (48)$$

Considering the restriction from formula (14), a and b have to fulfill $a \geqslant b \geqslant 0$.

During testing we figured out that if the first candidates of a and b fit the condition from formula (14) they also have to be bigger than their second candidate. This leads to the following rules to select the suitable a and b:

1. **if** $((aTmp_1 \geqslant bTmp_1)$ and $(bTmp_1 \geqslant 0)$ and $(aTmp_1 \geqslant aTmp_2)$ and $(bTmp_1 \leqslant bTmp_2))$ **then** $a = aTmp_1; b = bTmp_1;$
2. **else** $a = aTmp_2; b = bTmp_2;$

The rules were evaluated during testing and provide a minimal error rate.

Afterwards it is possible to calculate two candidates for each α and β by using the following formulas:

$$\alpha_i^{(1)} = \arctan(z_2, z_1) \quad \text{with} \quad z = M_{t_i}^T \begin{pmatrix} \cos(\gamma_i) \\ \sin(\gamma_i) \end{pmatrix} \quad (49)$$

$$\beta_i^{(1)} = \alpha_i^{(1)} - \arctan\left(\sqrt{1 - t_i^2}, t_i\right) \quad (50)$$

$$\alpha_i^{(2)} = \arctan(z_2, z_1) \quad \text{with} \quad z = M_{t_i} \begin{pmatrix} \cos(\gamma_i) \\ \sin(\gamma_i) \end{pmatrix} \quad (51)$$

$$\beta_i^{(2)} = \alpha_i^{(2)} + \arctan\left(\sqrt{1 - t_i^2}, t_i\right) \quad (52)$$

with

$$M_{t_i} = \begin{pmatrix} a + b\, t_i & b\sqrt{1 - t_i^2} \\ -b\sqrt{1 - t_i^2} & a + b\, t_i \end{pmatrix} \text{ for } i = 1, 2, 3. \quad (53)$$

The notation means that candidate one of α_i is $\alpha_i^{(1)}$ and candidate two of α_i is $\alpha_i^{(2)}$.

Now we have got 8 possible solutions for α and β:

1. $\alpha = (\alpha_1^1, \alpha_2^1, \alpha_3^1), \quad \beta = (\beta_1^1, \beta_2^1, \beta_3^1)$
2. $\alpha = (\alpha_1^1, \alpha_2^1, \alpha_3^2), \quad \beta = (\beta_1^1, \beta_2^1, \beta_3^2)$
3. $\alpha = (\alpha_1^1, \alpha_2^2, \alpha_3^1), \quad \beta = (\beta_1^1, \beta_2^2, \beta_3^1)$
4. $\alpha = (\alpha_1^1, \alpha_2^2, \alpha_3^2), \quad \beta = (\beta_1^1, \beta_2^2, \beta_3^2)$
5. $\alpha = (\alpha_1^2, \alpha_2^1, \alpha_3^1), \quad \beta = (\beta_1^2, \beta_2^1, \beta_3^1)$
6. $\alpha = (\alpha_1^2, \alpha_2^1, \alpha_3^2), \quad \beta = (\beta_1^2, \beta_2^1, \beta_3^2)$
7. $\alpha = (\alpha_1^2, \alpha_2^2, \alpha_3^1), \quad \beta = (\beta_1^2, \beta_2^2, \beta_3^1)$
8. $\alpha = (\alpha_1^2, \alpha_2^2, \alpha_3^2), \quad \beta = (\beta_1^2, \beta_2^2, \beta_3^2)$

The correct solution has to fulfill the following requirements:

$$\alpha_k \equiv k \cdot \alpha_1 \bmod 2\pi \qquad \beta_k \equiv k \cdot \beta_1 \bmod 2\pi \quad \text{with} \quad k = 2, 3. \tag{54}$$

It can occur that some congruences are not fulfilled at all or that some congruences are fulfilled by multiple solutions. Hence it is required to determine the solution which is closest to the congruences (54). Therefore the following error function e has been developed:

$$\begin{aligned} e = &(\mathrm{modulo2Pi}(\alpha_2^{(q)} - 2 \cdot \alpha_1^{(p)})) + (\mathrm{modulo2Pi}(\alpha_3^{(r)} - 3 \cdot \alpha_1^{(p)})) \\ &+(\mathrm{modulo2Pi}(\beta_2^{(q)} - 2 \cdot \beta_1^{(p)})) + (\mathrm{modulo2Pi}(\beta_3^{(r)} - 3 \cdot \beta_1^{(p)})) \\ &\text{with} \quad p = 1, 2, \quad q = 1, 2, \quad r = 1, 2. \end{aligned} \tag{55}$$

It is required to minimize e by choosing one of the possible 8 solution sets which provides the smallest value for e.

4.4 MODIFICATIONS TO HANDLE NOISE

In chapter 4.1 it was already described that it is possible to add noise to the input. If noise is added, the algorithm does not know this and especially does not know which data contains noise. Besides, it is not sufficient to apply some kind of filters to the input data. If this algorithm is used to detect errors on glass, this filter might also erase this error. Therefore, we enhanced the algorithm to detect noise and to erase it, if possible.

The effect of the two kinds of noises (on amplitude or phase) are a little different, but in general one big problem occurs: the wrong optimal solution is selected, because noise influences the minimization of the error function. Take for example a case were noise influences the third channel (measurement) containing the values of α_3 and β_3 and the actual optimal solution would be:

$$\alpha = (\alpha_1^{(1)}, \alpha_2^{(1)}, \alpha_3^{(1)}), \quad \beta = (\beta_1^{(1)}, \beta_2^{(1)}, \beta_3^{(1)}).$$

During the minimization of the error function it is likely that another solution is selected, because the error on α_3 and β_3 might be bigger than the overall error of another solution set – selecting

$$\alpha = (\alpha_1^{(2)}, \alpha_2^{(2)}, \alpha_3^{(2)}), \quad \beta = (\beta_1^{(2)}, \beta_2^{(2)}, \beta_3^{(2)}).$$

In fact, the noise on α_3 and β_3 might cause the selection of a total different solution. Hence, the noise influences other αs and βs. In the worst case, every of the selected αs and βs is different from the optimal solution. This has a much higher impact on the final result than only wrong values in one channel.

This leads to our main question: How can we handle noise without influencing the noiseless use-cases? Because when we only adjust the error function, the cases without noise are highly influenced.

For this case we propose to apply the following procedure:

Algorithm 4.4.1 (Handling noise)
Input: All possibilities for α and β.

1. *Start with determining the solution with the lowest error function.*

2. *Check if each congruence is fulfilled:*

$$\begin{aligned}(\alpha_2^{(qq)} - 2 \cdot \alpha_1^{(pp)}) &\to 0\\ (\alpha_3^{(rr)} - 3 \cdot \alpha_1^{(pp)}) &\to 0\\ (\beta_2^{(qq)} - 2 \cdot \beta_1^{(pp)}) &\to 0\\ (\beta_3^{(rr)} - 3 \cdot \beta_1^{(pp)}) &\to 0\end{aligned}$$

   with pp, qq, rr are the αs and βs, selected by the error function.

3. *If one of these congruences is not fulfilled correctly, we assume that the error function has given the wrong result, because at least one channel of the αs and βs contains noise. Hence we apply a modified error function to all values and determine if we can make the error smaller by only taking two α and β values into account and by recalculating the third values out of the two others.*

4. *In doing so, it is very important to determine if the new error function did really improve the previous result. Only comparing the two function results is not sufficient, because the second error function contains less congruence calculations. Hence it is easier to get a second error function which is smaller than the first one. Instead we have chosen to determine if every congruence has improved since the new calculations and only in this case, we take the new values.*

Output: α and β.

The following example shows this approach containing noise on channel three:

$$\begin{aligned} e &= ((\alpha_2^{(q)} - 2 \cdot \alpha_1^{(p)}) \bmod 2\pi) \\ &\quad + ((\beta_2^{(q)} - 2 \cdot \beta_1^{(p)}) \bmod 2\pi) \\ &\text{with} \quad p = 1, 2, \quad q = 1, 2. \\ \Rightarrow \alpha_3^{(rr)} &= \mathtt{shiftAngleToPi}(3 \cdot \alpha_1^{(pp)}) \\ \beta_3^{(rr)} &= \mathtt{shiftAngleToPi}(3 \cdot \beta_1^{(pp)}) \end{aligned}$$

with pp, qq are the αs and βs, selected by the new error function.

4.5 MODIFICATIONS TO HANDLE QUADRATIC FUNCTIONS

Basically the same calculation which have been already applied to the images generated by the linear functions will be applied in this case as well. But it has to be considered that the following complications might occur:

1. The value of the calculated t_1 can be wrong and the correct value would be $-t_1$. Surprisingly, the incorrect t_1s provide better results regarding the pixels in the final α and β images than calculating with the correct t_1s.

2. Analysing the "correct" results shows that the following can be true:

$$\begin{aligned} t_1 &= -1, \ \xi_1 = -\pi \\ t_2 &= +1, \ \xi_2 = -\pi \\ t_3 &= -1, \ \xi_3 = -\pi \end{aligned}$$

   or

$$\begin{aligned} t_1 &= -1, \ \xi_1 = -\pi \\ t_2 &= +1, \ \xi_2 = -\pi \\ t_3 &= -1, \ \xi_3 = 0 \end{aligned}$$

   This shows that

$$\cos(\xi_i) = t_i$$

   is not longer valid.

3. The αs and βs do not entirely behave like in the previous chapters. This time it can occur, that $\alpha_1 = \pi, \alpha_2 = \pi, \alpha_3 = \pi$ or $\alpha_1 = -\pi, \alpha_2 = -\pi, \alpha_3 = -\pi$ or $\alpha_1 = 0, \alpha_2 = \pi, \alpha_3 = \pi$. The same applies for β as well. Thereby the congruences from formula (54) are not entirely fulfilled any more.

Especially the second behavior causes much more errors when the input data contains a noise.

4.6 EVALUATION

This sections shows the reconstructed results, which have been achieved by the proposed algorithm. Figure 3 shows two the reconstructed α in the left column and β in the right column. The first row shows an execution using perfect input data. Here, we can see that the result is absolutely accurate without any error. The second row shows the image perturbed by a random noise in the range of $[-\frac{1}{4}, \frac{1}{4}]$. It can be observed that nearly every pixel is falsified and especially β is almost destroyed.

Hence, we can conclude this approach provides a good accuracy for perfect input data. However, due to the fact that real life applications never provide perfect data, this approach can be considered as not sufficient. A proper reconstruction of noisy input data is not possible.

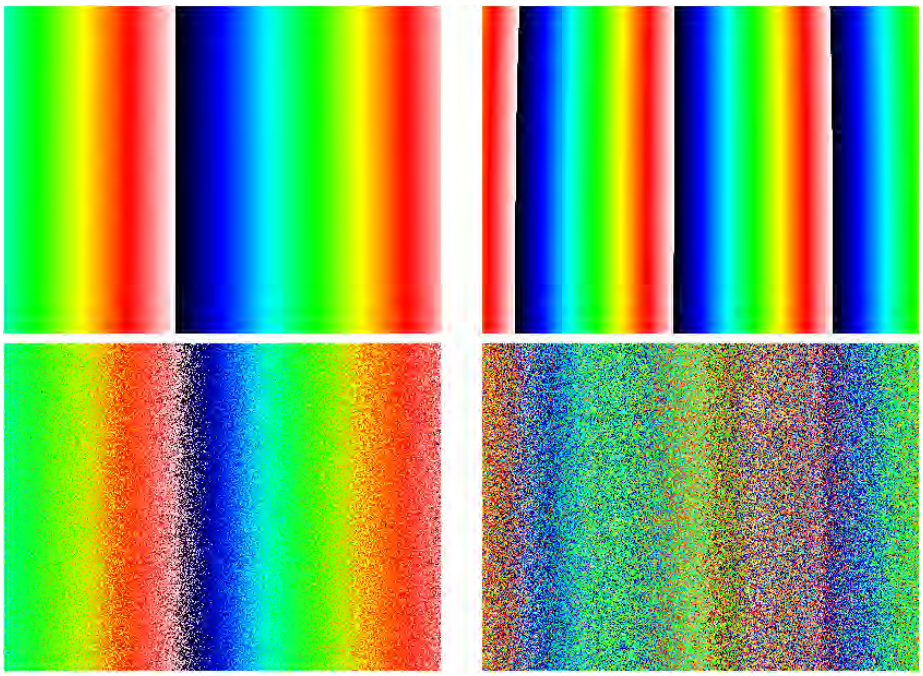

Figure 3: Reconstructed images of α (left side) and β (right side). The lower images are reconstructed from noisy input data.

Part III

PRONY'S METHOD

5

THEORTICAL BACKGROUND OF PRONY'S METHOD

Gaspard Clair Francois Marie Riche de Prony developed 1795 a method to solve a recovery problem, which arises from an application in physical chemistry. Nowadays, this method is also used in signal processing. [8]

The recovery problem is represented by a function f of samples $f(x)$ with $x \in X$. X is a finite set of points, where the function values are known.

Right now, the recovery problem has infinitely many solutions, e.g., there are infinitely many possibilities to connect two points. Hence, it is crucial to provide assumptions for f to get a unique or well-defined solution.

At first, we assume that f can be represented as a linear combination of know basis functions f_j and coefficients a_j

$$f = \sum_{j=1}^{n} a_j f_j, \quad a_j \in \mathbb{R}. \tag{56}$$

In this case, only the coefficients a_j have to be computed. The computation can be done by solving a linear system of equations. [8]

Particularly Prony considered the so-called **Prony functions**

$$z = f(x) = \sum_{j=1}^{n} \mu_j \nu_j^x, \tag{57}$$

which depend on coefficients $\mu_j \in \mathbb{R}$ and $\nu_j \in \mathbb{R}_+$. Compared to formula (56), solving the Prony function (57) requires to find the correct basis functions

$$f_j(x) = \nu_j^x,$$

which fulfill the measurements z_k. [8]

For equidistant sampling this results in

$$z_k = f(hk) = \sum_{j=1}^{n} \mu_j \nu_j^{hk} = \sum_{j=1}^{n} \mu_j \hat{\nu}_j^k, \tag{58}$$
$$\text{with} \quad \hat{\nu} := \nu^h \text{ and } k \in \mathbb{N}_+.$$

[8]

5.1 CLASSICAL PRONY METHOD

Prony started calculating numbers p_j with $j = 0, ..., n$ based on the measurements z_k with $k = 0, 1, 2, ...$, satisfying:

$$\begin{aligned} 0 &= z_0 \cdot p_0 + ... + z_n \cdot p_n \\ 0 &= z_1 \cdot p_0 + ... + z_{n+1} \cdot p_n \\ &... \\ 0 &= z_k \cdot p_0 + ... + z_{n+k} \cdot p_n, \quad \text{with} \quad k = 0, 1, 2, ... \end{aligned} \tag{59}$$

This correlation can also be written as

$$(z \star p)_k := \sum_{j=0}^{n} z_{k+j} \cdot p_j, \quad \text{with} \quad k = 0, 1, 2, ... \tag{60}$$

Hence, the requirement is

$$z \star p = 0, \tag{61}$$

which basically describes finding a filter p to erase the values of z. [5, 8]

Now, we apply formula (57) on (61) and get

$$\begin{aligned} 0 = (z \star p)_k &= \sum_{j=0}^{n} z_{k+j} \cdot p_j \\ &\overset{(57)}{=} \sum_{j=0}^{n} p_j \sum_{l=0}^{n} \mu_l \nu_l^{h(k+j)} \\ &\overset{(58)}{=} \sum_{j=0}^{n} p_j \sum_{l=0}^{n} \mu_l \hat{\nu}_l^{k+j} \\ &= \sum_{l=0}^{n} \mu_l \hat{\nu}_l^{k} \sum_{j=0}^{n} p_j \hat{\nu}_l^{j} \\ &\overset{p(x)=\sum_{j=0}^{n} p_j x^j}{=} \sum_{l=0}^{n} \mu_l \hat{\nu}_l^{k} p(\hat{\nu}_l). \end{aligned} \tag{62}$$

[8]

This leads to the so-called **Prony polynomials**

$$p(x) = (x - \hat{\nu}_1) \cdot ... \cdot (x - \hat{\nu}_n) = \sum_{j=0}^{n} p_j x^j, \tag{63}$$

which are a solution of (62) with $p_n = 1$. Additionally, any solution of (62) is an coefficient vector, which is a multiple of the Prony polynomial (63). Hence, if it is possible to find the smallest filter p so that (61) is fulfilled, then the zeros of the corresponding polynomial are exactly the $\hat{\nu}_j$ from formula (63). [4, 8]

For calculating these $\hat{\nu}_j$, it is required to solve the linear system (59). Therefore, different approaches are provided and covered in the following sub-sections.

5.1.1 *Eigenvalue Problem*

Plonka and Tasche considered the exponential sum

$$z = f(x) = \sum_{j=1}^{n} \mu_j e^{f_j x}, \quad x \geqslant 0 \tag{64}$$

with $f_j \in [-\alpha, 0] + i[-\pi, \pi)$ unique complex numbers, $\alpha > 0$ and $\mu_j \in \mathbb{C}\backslash\{0\}$.

The f_js are damping the exponential $e^{f_j x}$ because of the fact that the real part of the f_js is smaller or equal to zero. Hence the sum (64) consists of non-increasing $|e^{f_j x}|$ for $x \geqslant 0$.

It is possible to recover all parameters of such an exponential sum (64) if

$$z_k := \sum_{j=1}^{n} \mu_j e^{f_j k} = \sum_{j=1}^{n} \mu_j \nu_j^k \in \mathbb{C}, \quad k = 0, ..., 2n-1, \tag{65}$$

with $\nu_j = e^{f_j}$ distinct values, given noiseless sample data. This kind of problem is called **frequency analysis problem** and again leads to the so-called the **Prony polynomial**

$$p(x) = \prod_{j=1}^{n} (x - \nu_j) = (\sum_{j=0}^{n-1} p_j x^j) + x^n, \tag{66}$$

with $p_n = 1$. [3, 4]

Plonka and Tasche described solving the problem by using an eigenvalue problem. Therefore, they define the **companion matrix** $C_n(p) \in \mathbb{C}^{n\times n}$ of the Prony polynomial (66) by

$$C_n(p) = \begin{pmatrix} 0 & 0 & \cdots & 0 & -p_0 \\ 1 & 0 & \cdots & 0 & -p_1 \\ 0 & 1 & \cdots & 0 & -p_2 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -p_{n-1} \end{pmatrix}, \tag{67}$$

which fulfills

$$\det(xI_n - C_n(p)) = p(x), \quad x \in \mathbb{C}.$$

Hence, the zeros in the Prony polynomial (66) corresponds to the eigenvalues in the companion matrix (67). [3]

Besides, the relation (62) from section (5.1) is fulfilled as well, which leads to

$$\sum_{j=0}^{n-1} p_j z_{j+k} = -z_{n+k}, \quad \text{with} \quad p_n = 1. \tag{68}$$

This can be reformulated as the linear system

$$H_{n,n}(p_j)_{j=0}^{n-1} = -(z_{n+k})_{k=0}^{n-1} \tag{69}$$

with the **square Hankel matrix**

$$H_{n,n} := \begin{pmatrix} z_0 & z_1 & \cdots & z_{n-1} \\ z_1 & z_2 & \cdots & z_n \\ \vdots & \vdots & & \vdots \\ z_{n-1} & z_n & \cdots & z_{2n-2} \end{pmatrix} = (z_{j+k})_{j,k=0}^{n-1}. \tag{70}$$

This matrix is invertible and can again be reformulated using the **Vandermonde matrix**

$$V_n(\nu) := (\nu_k^{j-1})_{j,k=1}^{n,n} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ \nu_1 & \nu_2 & \nu_3 & \cdots & \nu_n \\ \vdots & \vdots & \vdots & & \vdots \\ \nu_1^{n-1} & \nu_2^{n-1} & \nu_3^{n-1} & \cdots & \nu_n^{n-1} \end{pmatrix}$$

to

$$H_{n,n} = V_n(\nu) \cdot \mathrm{diag}(\mu) \cdot V_n(\nu)^T, \tag{71}$$

with $\mathrm{diag}(\mu)$ is the diagonal matrix of the coefficients $\mu = (\mu_j)_{j=1}^n$ of the exponential sum (64). [3]

These steps can now be summarized the following algorithm.

Algorithm 5.1.1 (Classical Prony method)
Input: $n \in \mathbb{N}$, *sampled values* z_k, *with* $k = 0, \cdots, 2n-1$ *of the exponential sum* (64).

1. *Solve the linear system* (69).
2. *Computation of all zeros* ν_j *of the Prony polynomial* (66), *by determining the eigenvalues of the corresponding companion matrix* (67) *and form the complex logarithm* $f_j := \log(\nu_j)$ *for all* $j = 1, \cdots, n$.
3. *Solving of the Vandermonde system*

$$V_n(\nu)(\mu_j)_{j=1}^n = (z_k)_{k=0}^{n-1}.$$

Output: $f_j \in [-\alpha, 0] + i[-\pi, \pi),\ \alpha > 0,\ \mu_j \in \mathbb{C}\backslash\{0\},\ j = 1, ..., n.$ *[3]*

5.2 ESPRIT METHOD

The ESPRIT method requires an additional upper bound $L \in \mathbb{N}$ with $M \leqslant L \leqslant n$ to form the rectangular Hankel matrix

$$H_{2n-L,L+1} := (z_{l+m})_{l,m=0}^{2n-L-1,L} \in \mathbb{C}^{(2n-L)\times(L+1)}. \tag{72}$$

If the sampled data does not contain noise, the Hankel matrix is rank deficient with rank n, because of (62). [3]

In the following, we will describe the algorithm of the ESPRIT-method, which basically applies a Singular Value Decomposition (SVD) to the Hankel matrix $H_{2N-L,L+1}$. The detailed mathematical derivation of this algorithm is explained by Plonka and Tasche. [3]

Algorithm 5.2.1 (ESPRIT method for equispaced sampling)
Input: $2n \in \mathbb{N}$ *number of samples,* $3 \leqslant L \leqslant n$ *window length and samples* z_k *with* $k = 0, ..., 2n-1$.

1. *Computation of the SVD* $H_{2n-L,L+1} = U_{2n-L} D_{2n-L,L+1} W_{L+1}^H$ *of the Hankel matrix (72). The singular values* $\sigma_l (l = 1, ..., \min\{L, n-L+1\})$ *are ordered decreasingly. Determination of the rank* M *of (72), such that* $\sigma_M \geqslant \epsilon \sigma_1$ *with* $0 < \epsilon \ll 1$. *This leads to*

$$W_{M,L}(s) := W_{L+1}(1 : M, 1+s : L+s) \quad (\textit{with} \quad s = 0, 1).$$

2. *Calculation of the matrix*

$$F_M := (W_{M,L}(0)^H)^\dagger W_{M,L}(1)^H,$$

where $(W_{M,L}(0)^H)^\dagger$ *denotes the Moore-Penrose pseudoinverse of*

$$W_{M,L}(0)^H.$$

3. *Determination of all eigenvalues* $\nu_j (j = 1, ..., M)$ *of* F_M. *And calculate all* $f_j := \log(\nu_j)$ *for* $j = 1, ..., M$.

4. *Computation of the coefficient vector* $\mu := (\mu_j)_{j=1}^M \in \mathbb{C}^n$, *which is a solution of the overdetermined Vandermonde-like system:*

$$V_{2n,M}(\nu)(\mu_j)_{j=1}^M = (z_k)_{k=0}^{2n-1}$$

with the rectangular Vandermonde matrix $V_{2n,M}(\nu) := (\nu_j^{k-1})_{k,j=1}^{2n,M}$.

Output: $M \in \mathbb{N}$, $f_j \in [-\alpha, 0] + i[-\pi, \pi)$, $\alpha > 0$, $\mu_j \in \mathbb{C}\backslash\{0\}$, $j = 1, ..., M$. *[3]*

5.3 MATRIX PENCIL METHOD

Another approach is based on the fact, that the Prony method can be transformed to a **matrix pencil** method. It is nearly similar to the ESPRIT method, with the main difference that the Hankel matrix is not decomposed by a SVD, but by a *QR*-decomposition. The detailed mathematical derivation of this algorithm is explained by Potts and Tasche. [4]

Algorithm 5.3.1 (Matrix pencil method)
Input: $2n \in \mathbb{N}$ *number of samples,* $3 \leqslant L \leqslant n$ *window length,* z_k *with* $k = 0, ..., 2n-1$ *samples.*

1. *Computation of the QR decomposition*

$$H_{2n-L,L+1} \Pi_{L+1} = Q_{2n-L} R_{2n-L,L+1}$$

of the Hankel matrix (72). Determination of the rank M *of (72), such that* $R_{2n-L,L+1}(M+1, M+1) < \epsilon R_{2n-L,L+1}(1,1)$ *with* $0 < \epsilon \ll 1$.

2. *Calculation of*

$$T_{M,L}(s) := S_{2n-L,L}(1:M, 1+s:L+s), \quad (s=0,1)$$

with

$$S_{2n-L,L+1} := R_{2n-L,L+1}\Pi_{L+1}^{T}$$

and

$$S_{2n-L,L}(s) := S_{2n-L,L+1}(1:2n-L, 1+s:L+s) \quad (s=0,1),$$

3. *Calculation of the matrix*

$$F_M^{QR} := (T_{M,L}(0)^T)^{\dagger} T_{M,L}(1)^T.$$

4. *Determination of all eigenvalues* $\nu_j (j = 1, ..., M)$ *of* F_M^{QR}. *And calculate all* $f_j := \log(\nu_j)$ *for* $j = 1, ..., M$.

5. *Computation of the coefficient vector* $\mu := (\mu_j)_{j=1}^{M} \in \mathbb{C}^n$, *which is a solution of the overdetermined Vandermonde-like system:*

$$V_{2n,M}(\nu)(\mu_j)_{j=1}^{M} = (z_k)_{k=0}^{2n-1}$$

with the rectangular Vandermonde matrix $V_{2n,M}(\nu) := (\nu_j^{k-1})_{k,j=1}^{2n,M}$.

Output: $M \in \mathbb{N}$, $f_j \in [-\alpha, 0] + i[-\pi, \pi)$, $\alpha > 0$, $\mu_j \in \mathbb{C}\backslash\{0\}$, $j = 1, ..., M$. *[4]*

To compare the actual behaviour and performance of the proposed algorithms, the following chapter describes their implementation.

6 IMPLEMENTATION OF PRONY'S METHOD

The previous chapter already explained the theoretical background and different algorithms to apply Prony's method. This chapter will give an overview over the implementation using Matlab/Octave to determine μ_j and f_j for the samples $z_k = \sum_{j=1}^{n} \mu_j e^{f_j k}$ with $k = 0, ..., 2n-1$.

6.1 GENERATION OF TEST-DATA

To test the different Prony algorithms it is required to provide some testing data to check if a proper reconstruction of functions is possible. Additionally, it might be interesting to provide different sample distributions to observe the behaviour of the different algorithms. Therefore, five different functions for sample generation will be provided, which basically require the same input parameters.

6.1.1 *Equispaced Sampling*

This function expects the following parameters:

- `n`: `n` is the number which is used to create $2 \times n$ sample values.
- `stepSize`: The step size between two points on the x-axis.
- `startPoint`: The start point on the x-axis.
- `polynomial`: A coefficient vector describing the parameters of an algebraic polynomial, e.g., the vector `[1 2 0 3]` represents the polynomial $x^3 + 2x^2 + 3$.

Using these parameters, the matlab function creates the vectors `x` and `z`. `x` contains all $2 \times n$ points on the x-axis, beginning with the given start point. `z` contains all function evaluations of the given polynomial at the points in `x`.

Listing 5: Matlab function to generate equispaced samples.

```
%"/InputProvider/generateSamplesWithGivenStepSize.m"

function [x,z] = generateSamplesWithGivenStepSize(n, stepSize,
    startPoint, polynomial)
        x = zeros(2*n,1);
        z = zeros(2*n,1);
        [size1,size2] = size(polynomial);
        for i = 1 : 2*n
```

```
                x(i) = startPoint + (i-1)*stepSize;
                if(size1 == 1 && size2 == 3 && polynomial == "sin
                    ")
                        z(i) = sin(x(i));
                else
                        z(i) = polyval(polynomial,x(i));
                end
        end
end
```

6.1.2 *Randomly-spaced Sampling*

This function is used to sample with a random step size. Hence, the input `stepSize` is multiplied with a random number in the range $[0, 1]$. The remaining input parameters are used like in the equispaced function.

Listing 6: Matlab function to generate randomly-spaced samples.

```
%"/InputProvider/generateSamplesWithRandomStepsize.m"

function [x,z] = generateSamplesWithRandomStepsize(n, stepSize,
    startPoint, polynomial)
x = zeros(2*n,1);
        z = zeros(2*n,1);
        [size1,size2] = size(polynomial);
        for i = 1 : 2*n
                if i == 1
                        x(i) = startPoint + rand()*stepSize;
                else
                        x(i) = x(i-1) + rand()*stepSize;
                end

                if(size1 == 1 && size2 == 3 && polynomial == "sin
                    ")
                        z(i) = sin(x(i));
                else
                        z(i) = polyval(polynomial,x(i));
                end
        end
end
```

6.1.3 *Equispaced-Sampling with two Points converging*

This function works basically the same way like the equispaced sampling. But it differs in the last point, because the last point of the x values is chosen very close to the last but one x value.

Listing 7: Matlab function to generate samples with two point converging to each other.

```
%"/InputProvider/
    generateSamplesWithGivenStepSizeTwoPointsConverging.m"

function [x,z] =
    generateSamplesWithGivenStepSizeTwoPointsConverging(n,
    stepSize, startPoint, polynomial)
        x = zeros(2*n,1);
        z = zeros(2*n,1);
        [size1,size2] = size(polynomial);
        for i = 1 : 2*n
                if i == 2*n
                        x(i) = x(i-1) + 0.001;
                else
                        x(i) = startPoint + (i-1)*stepSize;
                end

                if(size1 == 1 && size2 == 3 && polynomial == "sin
                    ")
                        z(i) = sin(x(i));
                else
                        z(i) = polyval(polynomial,x(i));
                end
        end
end
```

6.1.4 *Sampling with convergent distances*

This function provides samples which start with the given `stepSize`, but afterwards the step size between each x-value convergences to zero. Right now, the implementation requires a provided stepsize in the range of $[0, 1]$ to fulfill the convergence.

Listing 8: Matlab function to generate samples with a stepsize converging to zero.

```
%"/InputProvider/generateSamplesWithGivenStepSizeAndConvergence.m
    "

function [x,z] = generateSamplesWithGivenStepSizeAndConvergence(n
    , stepSize, startPoint, polynomial)
        x = zeros(2*n,1);
        z = zeros(2*n,1);
        [size1,size2] = size(polynomial);
        for i = 1 : 2*n
                if i == 1
                        x(i) = startPoint + stepSize;
                        stepSize = stepSize - 0.1;
                else
                        x(i) = x(i-1) + stepSize^i;
                end
```

```
                if(size1 == 1 && size2 == 3 && polynomial == "sin
                    ")
                        z(i) = sin(x(i));
                else
                        z(i) = polyval(polynomial,x(i));
                end
        end
end
```

6.1.5 *Sampling by randomly determining* μ *and* f

The following sampling procedure in listing 9 uses a slightly different approach. This time, we are going to generate sample values for randomly chosen parameters μ and f. Hence, instead of providing a polynomial, a constant `M` is provided. This `M` determines the number of coefficients μ and f. Besides, it has to be ensured, that $M \leqslant n$ is fulfilled.

Afterwards, `M` random, complex-valued numbers are created for each μ and f, with the restriction to $\mu_j \in \mathbb{C}\backslash\{0\}$ and $f_j \in [-\alpha, 0] + i[-\pi, \pi]$ with $\alpha > 0$. Using these μ and f it is possible to determine z by calculating

$$z(k) = \sum_{j=1}^{M} \mu_j \cdot e^{f_j \cdot k} \text{ for } k = 0, ..., 2n-1.$$

Listing 9: Matlab function to generate samples with a stepsize by randomly chosen μ and f.

```
%"/InputProvider/generateSamplesWithGivenStepSizeFromMuAndF.m"

function [x,z,mu,f] = generateSamplesWithGivenStepSizeFromMuAndF(
    M, n, stepSize, startPoint)
        mu = (rand(M,1) - 1/2) * 10 + i * 2 * pi * (rand(M,1) -
            1/2);
        f = -rand(M,1) + i * 2 * pi * (rand(M,1) - 1/2);

        x = zeros(2*n,1);
        z = zeros(2*n,1);
        for i = 0 : 2*n-1
                x(i+1) = startPoint + i*stepSize;
                for j = 1 : M
                        z(i+1) = z(i+1) + mu(j)*exp(f(j)*i);
                end
        end
end
```

6.2 IMPLEMENTATION OF PRONY ALGORITHMS

6.2.1 *Classical Prony Algorithm*

The implementation of the classical Prony method is adapted from algorithm 5.1.1. In line 4 of listing 10, we start with generating the Hankel matrix[1]

$$H = \begin{pmatrix} z_1 & z_2 & \cdots & z_n \\ z_2 & z_3 & \cdots & z_{n+1} \\ \vdots & \vdots & & \vdots \\ z_n & z_{n+1} & \cdots & z_{2n-1} \end{pmatrix},$$

which will be used to determine the Prony polynomial p by solving

$$H \cdot p = - \begin{pmatrix} z_{n+1} \\ z_{n+2} \\ \vdots \\ z_{2n} \end{pmatrix}.$$

Afterwards, line 7 is used to calculate the zeros of the Prony polynomial ν by determining the eigenvalues of the companion matrix

$$C = \begin{pmatrix} 0 & 0 & \cdots & 0 & -p_1 \\ 1 & 0 & \cdots & 0 & -p_2 \\ 0 & 1 & \cdots & 0 & -p_3 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -p_n \end{pmatrix}$$

of p. The calculation of the companion matrix is described in listing 11.

The Vandermonde matrix

$$V = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ \nu_1 & \nu_2 & \nu_3 & \cdots & \nu_n \\ \vdots & \vdots & \vdots & & \vdots \\ \nu_1^{n-1} & \nu_2^{n-1} & \nu_3^{n-1} & \cdots & \nu_n^{n-1} \end{pmatrix}$$

is used to calculate μ out of the previously determined zeros of the Prony polynomial. The calculation of the Vandermonde matrix is described in listing 12.

Listing 10: Matlab function using the classical Prony method.

```
1  %"/PronyAlgorithms/ClassicalProny.m"
2
3  function [f,mu] = ClassicalProny(n, z)
4          H = hankel(z(1:n),z(n:2*n-1));
5          p = H \ (-z(n+1:2*n))
6
7          nu = eig(calculateCompanionMatrix(n,p));
8          f = log(nu);
9
10         mu = calculateVandermondeMatrix(n/2,nu,n) \ z(1:n);
11 end
```

1 In the following our indices will always start at 1, because of the matlab notation.

Listing 11: Matlab function to calculate the companion matrix.

```
%"/MatrixFunctions/calculateCompanionMatrix.m"

function C = calculateCompanionMatrix(n,p)
        C = diag(ones(n-1,1),-1);
        C(:,n) = -p;
end
```

Listing 12: Matlab function to calculate the Vandermonde matrix.

```
%"/MatrixFunctions/calculateVandermondeMatrix.m"

function V = calculateVandermondeMatrix(n,nu,M)
        if nargin() == 2
                V = calculateMatrix(n,n,nu);
        else
                V = calculateMatrix(n,M,nu);
        end
end

function V = calculateMatrix(n,M,nu)
        exponent = (0:1:2*n-1);
        V = (ones(2*n,M)*diag(nu)).^(diag(exponent)*ones(2*n,M));
end
```

6.2.2 *ESPRIT Method*

The ESPRIT method from algorithm 5.2.1 starts with the calculation of the rectangular Hankel matrix

$$H = \begin{pmatrix} z_1 & z_2 & \cdots & z_{2n-L} \\ z_2 & z_3 & \cdots & z_{2n-L+1} \\ \vdots & \vdots & & \vdots \\ z_{L+1} & z_{L+2} & \cdots & z_{2n} \end{pmatrix},$$

depending on the additional boundary $3 \leqslant L \leqslant n$. Afterwards, a SVD is applied to this Hankel matrix

$$H = U \cdot D \cdot W^H$$

to get the hermite matrix `WHermite`, which is required for all further computations. Hence, this time no companion matrix is required, because the companion matrix is replaced by a matrix `FM`. This `FM` can be determined by solving the following system of linear equations consisting of parts of W^H:

$$\begin{pmatrix} w^H_{(1,1)} & w^H_{(1,2)} & \cdots & w^H_{(1,M)} \\ w^H_{(2,1)} & w^H_{(2,1)} & \cdots & w^H_{(2,M+1)} \\ \vdots & \vdots & & \vdots \\ w^H_{(L,1)} & w^H_{(L,2)} & \cdots & w^H_{(L,M)} \end{pmatrix} \cdot FM = \begin{pmatrix} w^H_{(2,1)} & w^H_{(2,2)} & \cdots & w^H_{(3,M)} \\ w^H_{(3,1)} & w^H_{(3,1)} & \cdots & w^H_{(3,M+1)} \\ \vdots & \vdots & & \vdots \\ w^H_{(L+1,1)} & w^H_{(L+1,2)} & \cdots & w^H_{(L+1,M)} \end{pmatrix}$$

$$\text{with } M = \operatorname{rank}(H).$$

The remaining algorithm proceeds like the previously discussed algorithm, except that we need `M = rank(H)` as column dimension during the calculation of the Vandermonde matrix. Hence, this time our Vandermonde matrix looks like

$$V = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \nu_1 & \nu_2 & \cdots & \nu_M \\ \vdots & \vdots & & \vdots \\ \nu_1^{n-1} & \nu_2^{n-1} & \cdots & \nu_M^{2n-1} \end{pmatrix}.$$

Listing 13: Matlab function using the ESPRIT method.

```
%"/PronyAlgorithms/EspritProny.m"

function [f,mu,M] = EspritProny(n, L, z)
        H = hankel(z(1:L+1),z(L+1:end));
        [U,D,WHermite] = svd(H);

        M = rank(H);

        W0Hermite = WHermite(1:L,1:M);
        W1Hermite = WHermite(2:L+1,1:M);

        FM = W0Hermite\W1Hermite;

        nu = eig(FM);
        f = log(nu);

        mu = calculateVandermondeMatrix(n,nu,M)\z;
end
```

6.2.3 *Matrix Pencil Method*

The matrix pencil method from algorithm 5.3.1 looks very similar to the ESPRIT method. The only difference is, that no SVD is applied to the rectangular Hankel matrix, but the *QR-decomposition* `[Q,R,P] = qr(H)`. With these matrices, it is possible to calculate `S = R * P'`, which corresponds to the `W` (not `WHermite`) in the ESPRIT method. Hence, the following procedure of the matrix pencil method looks totally the same like in the ESPRIT method.

Listing 14: Matlab function using the matrix pencil method.

```
%"/PronyAlgorithms/MatrixPencilProny.m"

function [f,mu,M] = MatrixPencilProny(n, L, z)
        H = hankel(z(1:L+1),z(L+1:end));
        [Q,R,P] = qr(H);
        M = rank(H);

        S = R * P';
```

```
        T0 = S(1:M,1:L);
        T1 = S(1:M,2:L+1);

        FM = T0.'\T1.';

        nu = eig(FM);
        f = log(nu);

        mu = calculateVandermondeMatrix(n,nu,M) \ z;
end
```

6.2.4 *Reducing of the resulting coefficients*

After the coefficients μ_j and f_j have been determined using a previously described algorithm like the classical method it was possible to notice a certain pattern in the μ_js and f_js. Therefore, we can look at the following coefficients, which have been calculated to approximate the sine function using 20 samples on the range $[-10, 9]$ ($n = 10$) with equispaced sampling:

$$f = \begin{pmatrix} -0.15901+0.30266i \\ -0.15901-0.30266i \\ 0.00000+1.00000i \\ 0.00000-1.00000i \\ -0.20750+1.68055i \\ -0.20750-1.68055i \\ -0.28377+2.85141i \\ -0.28377-2.85141i \\ -0.26134+2.26940i \\ -0.26134-2.26940i \end{pmatrix} \quad \mu = \begin{pmatrix} -1.3380e-15-6.0716e-17i \\ -1.0736e-15-1.9762e-16i \\ \mathbf{2.7201e-01+4.1954e-01i} \\ \mathbf{2.7201e-01-4.1954e-01i} \\ 4.1823e-16-1.2710e-15i \\ 2.5931e-16+1.0258e-15i \\ 9.4572e-16-3.5264e-16i \\ 9.9128e-16+4.0062e-16i \\ 7.4065e-16-5.3382e-16i \\ 7.5744e-16+7.3574e-16i \end{pmatrix}.$$

Looking at μ shows, that only the real part of μ_3 and μ_4 is significantly bigger than zero. All remaining μ_j will cause that every summand

$$\mu_j e^{f_j k} = 0 \quad \forall j \in \{1, ...n\} \backslash \{3, 4\}.$$

Hence, it seams that only μ_3 and μ_4 are required to determine the result of the sum. To test this hypothesis, also f will be reduced to f_3 and f_4 (the only real zero parts of f).

The detailed function, which was applied using Matlab, is shown in Listing 15. In the code, `fInput` and `muInput` represent the previously determined f and μ of length n. The `epsilon` is a predefined constant which will be used for comparison, whether a value is zero or not.

Listing 15: Matlab function to reduce f and μ.

```
%"/PronyAlgorithms/ClassicalPronyReducedFAndMu.m"

function [f,mu] = reduceFAndMu(fInput, muInput, epsilon)
        f = [];
        mu = [];
        for i = 1 : size(fInput)(1)
```

```
                if (abs(real(fInput(i))) < epsilon) && (abs(real(
                    muInput(i))) > epsilon)
                        f = [f;fInput(i)];
                        mu = [mu;muInput(i)];
                end
        end
end
```

Finally, it was possible to use this approach for reconstructing functions. But it has to be considered, that it only worked for equispaced sampling. All other evaluated sampling distributions failed absolutely.

6.2.5 *Classical Prony Algorithm with a stepwise Reduction of the Hankel Matrix*

This algorithm uses basically the idea of the classical Prony method, with the main difference to find the Prony polynomial with minimal degree. Therefore, in listing 16 we start again with calculating the Hankel matrix

$$H = \begin{pmatrix} z_1 & z_2 & \cdots & z_n \\ z_2 & z_3 & \cdots & z_{n+1} \\ \vdots & \vdots & & \vdots \\ z_n & z_{n+1} & \cdots & z_{2n-1} \end{pmatrix}.$$

To minimize the degree of the Prony polynomial, it is required to reduce the columns of the Hankel matrix. Therefore, we take the linear system (59) and try iteratively to find the smallest number r of columns in H for which the linear system has a solution p

$$H(:, 1 : r+1) \cdot \begin{pmatrix} p \\ 1 \end{pmatrix} = 0.$$

This procedure is described in listing 17. In doing so, the first step is to determine p by solving

$$H(:, 1 : r) \cdot p = -H(:, r+1).$$

The minimal number r is found when the following norm

$$||H(:, 1 : r) \cdot p + H(:, r+1)||$$

becomes small enough. r also represents the rank of the matrix H.

The remaining procedure in listing 16 continues similar to the classical Prony method, except that every number n is replaced by r and the reduced Prony polynomial p is used.

Listing 16: Matlab function to use a stepwise reduction of the Hankel matrix during the classical Prony method.

```
%"/PronyAlgorithms/ClassicalPronyStepwiseHankel.m"
```

```
function [f,mu,r] = ClassicalPronyStepwiseHankel(n, z, epsilon)
H = hankel(z(1:n),z(n:2*n-1));

[r,p] = minimizeHankel(H,z,n,epsilon);

nu = eig(calculateCompanionMatrix(r,p));
f = log(nu);

mu = calculateVandermondeMatrix(r/2,nu,r) \ z(1:r);
end
```

Listing 17: Matlab function to minimize the Hankel matrix.

```
%"/PronyAlgorithms/minimizeHankel.m"

function [size,p] = minimizeHankel(H,z,n,epsilon)
        size = n-1;
        for col = 2 : n - 1
                reducedH = H(:,1:col);
                p = reducedH \ (-H(:,col+1));
                normValue = norm( reducedH * p + H(:,col+1))
                if(normValue < epsilon)
                        size = col;
                        break;
                end
        end
end
```

6.3 EVALUATION

6.3.1 *Accuracy depending on Sampling*

Now we are going to evaluate the accuracy of the different algorithms depending on the chosen sampling method. Therefore, Table 1 gives a brief overview using the following explanation:

- $\surd$: Successful approximation.
- ($\surd$): Nearly everything was approximated successful.
- X: Very few parts of the function have been approximated successful. It is also considered as a failure of approximation.
- XXX: Total failure of approximation.

It can be observed that the classical approach had an overall good accuracy and only failed when two points of the sampling set were converging to each other.

Compared to that, the classical approach which determined a minimal Prony polynomial stepwise performed also very good. In the

two cases with convergence it occurred that in the end of the sampling interval the two functions are slightly different. This can also be observed in Figure 4.

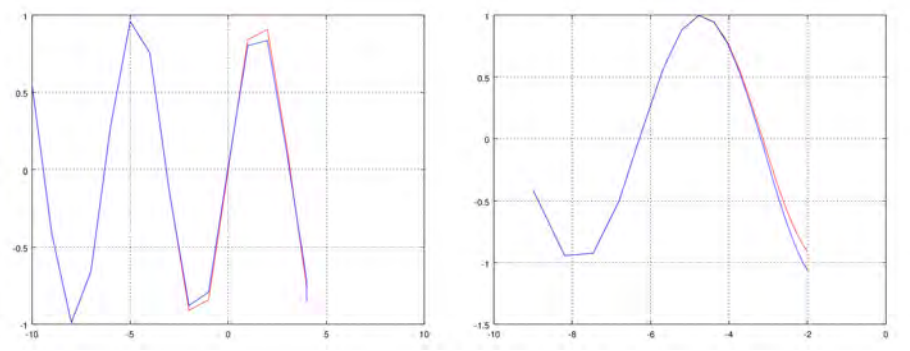

Figure 4: This figure shows the two samplings of the classical stepwise approach, which do not have a perfect reconstruction of the original function. The red line is the original function and the blue line is the calculated approximation. The left image shows the sampling with two converging points and the right image sampling with an overall convergence.

The matrix pencil method showed a very similar behavior. Figure 5 also shows a problem when two points are converging to each other and when the samples have random distances between each other.

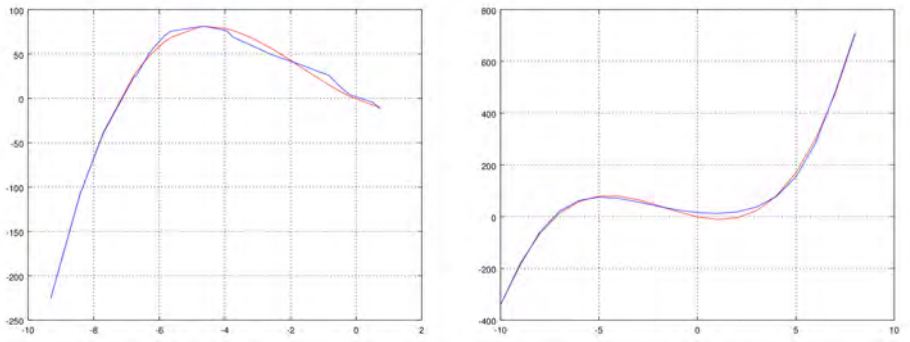

Figure 5: This figure shows the two samplings of the matrix pencil method, which do not have a perfect reconstruction of the original function. The left image shows random sampling and the right image shows sampling with two point converging.

The ESPRIT method worked for equispaced samples and for samples with converging distances. But it failed for randomly-spaced samples and samples where the last two points are very close.

Like it was already mentioned in the previous section, the approach which reduces f and the μ in the end only works for equispaced samples. Otherwise, it fails absolutely.

Table 1: Comparison of the different algorithms and sampling of functions

	Equi-spaced	Randomly-spaced	Convergent distances	Convergence of two points
Classical	√	√	√	X
Reduced-Classical	√	XXX	XXX	XXX
Stepwise-Classical	√	(√)	√	√
ESPRIT	√	X	√	X
Matrix Pencil	√	(√)	√	(√)

In general, when we have a closer look at the error

$$\|\mathtt{originalFunction}(x) - \mathtt{reconstructedFunction}(x)\|,$$

we see that the error of the three classical approaches lies in a range of 10^{-14}, but for ESPRIT and the Matrix Pencil method it only lies in the range of 10^{-4}.

When we have a closer look at the error range of the approach using the stepwise reduction, which is applied to different functions generated from random μ and f in Table 2, we see that this approach seems to be very robust.

Because of this and the fact, that the stepwise Prony method uses the Prony polynomial with minimal degree, we are going to use the approach from chapter 6.2.5 in the following.

Table 2: Comparison of the reduced Prony method with samples generated from random μ and f

	μ, f random	$\mu \to 0$	$f \to 0$	μ only imaginary	f only imaginary
Error of Reduced-Classical	10^{-14}	10^{-15}	10^{-7}	10^{-14}	10^{-11}

7 DENOISING

Like it was already mentioned in the last chapter, we will continue our work with the approach from chapter 6.2.5, which reduces the Hankel matrix stepwise.

In the following, we are going to describe two approaches to handle an input signal which contains noise.

7.1 DENOISING BY AVERAGING THE PRONY POLYNOMIAL

In general, if the input data is not perturbed by noise, it does not matter which sub-Hankel matrix of the size $n \times r$ is used to determine p:

$$H(:, i : i + r) \cdot p = 0 \quad \forall i \in \{1, ..., n - r + 1\}. \tag{73}$$

This leads to the main idea behind this denoising: We solve formula (73) for every $i \in \{1, ..., n - r + 1\}$ and build the average over all ps.

But it has to be considered that not every sub-Hankel matrix might have the same rank, which would result in different dimensions of the ps. Now, there are two possible solutions for this problem:

1. We simply sum up all ps and if the dimensions do not match, we append the required number of zeros.
2. We only sum up the ps with dimension $rank(H) \times 1$.

Testing the first solution, showed a very noisy result, which can be explained by the fact that the inserted zeros might falsify the denoising process. On the contrary, the second solution showed a very low error rate. Hence, we will only use the second solution in the following.

The remaining steps to determine the coefficients f and μ will be applied like already described in chapter 6.2.5 using the p from solution 2.

The following algorithm will give a detailed overview over the procedure of denoising. The sourcecode of this procedure can be found in listing 18.

Algorithm 7.1.1 (Denoising)
Input: Hankel matrix H *of size* $n \times n$.

1. *Determine the rank* r *of the Hankel matrix.*
2. *For every* i *from* 1 *to* $n - r + 1$*:*

a) *Determine the rank* `rSub` *and the Prony polynomial* `pSub` *of the sub-Hankel matrix* `H(:,i:n)` *by calling* `minimizeHankelWithNoise` *in listing 19.*

`minimizeHankelWithNoise` *works almost the same as* `minimizeHankel`*, which was already described in listing 17. There is only a difference regarding the termination. Originally, the algorithm terminates when the norm converges to zero. But this might not happen, when the matrix contains noise. In this case, the value of the norm tends to alternate. This means, it does not become smaller in every step. Sometimes, it happens that the norm jumps to a higher value and starts becoming smaller again. But these jumps occur randomly.*

Hence, we determine our procedure in the step right before the norm jumps to a higher value for the first time.

b) *If* `rSub` *is equal to the rank of the entire Hankel matrix, we sum up* `pSub` *to the already existing sum of* ps.

3. *Divide* p *point-wise by the number of* ps *which have been summed.*

Output: Coefficients p *of the Prony polynomial.*

Listing 18: Matlab function to determine the avery p over all sub-Hankel matrices.

```
%"/PronyAlgorithms/determineAverageP.m"

function [r,averageP] = determineAverageP(H,n,epsilon)
        [r,p] = minimizeHankelWithNoise(H,epsilon);

        averageP = zeros(r,1);
        cnt = 0;
        for step = 1 : n-r+1
                [rSub,pSub] = minimizeHankelWithNoise(H(:,step:n)
                    ,epsilon);

                if(r == rSub)
                        averageP = averageP + pSub;
                        cnt = cnt + 1;
                end
        end

        averageP = averageP./ cnt;
end
```

Listing 19: Matlab function to determine the rank and the Prony polynomial p by minimizing the Hankel matrix for input data containing noise.

```
%"/PronyAlgorithms/minimizeHankelWithNoise.m"
```

```
function [rank,p] = minimizeHankelWithNoise(H,epsilon)
        [m,n] = size(H);
        rank = n-1;
        oldNorm = inf;
        oldP = 0;
        for col = 2 : n - 1
                reducedH = H(:,1:col);
                p = reducedH \ (-H(:,col+1));
                normValue = norm( reducedH * p + H(:,col+1));
                if(oldNorm < normValue)
                        rank = col - 1;
                        p = oldP;
                        break;
                elseif(normValue < epsilon)
                        rank = col;
                        break;
                end
                oldNorm = normValue;
                oldP = p;
        end
end
```

7.2 DENOISING BY STACKING HANKEL SUB-MATRICES

This time, we want to use every information directly from the Hankel matrix to minimize the noise. Therefore, we again minimize the Hankel matrix, but only to determine the size r of the reduced Hankel matrix. Afterwards we take every sub-Hankel matrix and stack them on top of each other to solve formula (75):

$$\begin{bmatrix} H(:,1:r+1) \\ H(:,2:r+2) \\ \vdots \\ H(:,end-r:end) \end{bmatrix} \cdot \begin{bmatrix} p \\ 1 \end{bmatrix} = 0 \tag{74}$$

$$\Leftrightarrow \begin{bmatrix} H(:,1:r) \\ H(:,2:r) \\ \vdots \\ H(:,end-r:end-1) \end{bmatrix} \cdot p = - \begin{bmatrix} H(:,r+1) \\ H(:,r+2) \\ \vdots \\ H(:,end) \end{bmatrix}. \tag{75}$$

Formula (74) shows that we are stacking Hankel matrices of the size $(n \times r+1)$. The size $r+1$ is required to determine a Prony polynomial p with r coefficients in formula (75).

Algorithm 7.2.1 (Denoising by Stacking)
Input: Hankel matrix H *of size* $n \times n$ *and an* $\epsilon > 0$.

1. *Determine the rank* r *of the Hankel matrix by using the matlab function 19.*

2. *If the rank* r *is smaller than the size* n *of the Hankel matrix:*

   a) *Determine the matrix which consists of all sub-Hankel matrices of the size* $r+1$ *by calling function 21.*

b) Calculate p *by solving formula (75).*

3. *If the rank* r *is not smaller than the size* n *means that it was not possible to reduce the Hankel matrix. Hence, the resulting Prony polynomial* p *was already determined by function 19.*

Output: Coefficients p *of the Prony polynomial.*

Listing 20: Matlab function to determine p and r by stacking the sub-Hankel matrices.

```
function [r,p] = calcPByStackingH(H,n,epsilon)
        [r,p] = minimizeHankelWithNoise(H,epsilon);
        if(r < n)
                stackedH = stackH(H,n,r+1);
                p = stackedH(:,1:r) \ (-stackedH(:,r+1));
        end
end
```

Listing 21: Matlab function to stack sub-matrices on top of each other.

```
function stackedH = stackH(H, n, r)
        stackedH = [];
        for(i = 1 : n-r+1)
                stackedH = [stackedH; H(:,i:i+r-1)];
        end
end
```

7.3 COMPARISON

Using the previously determined Prony polynomials it is possible to calculate the coefficients μ and f, which are used to reconstruct the function.

When we compare the reconstructed functions with the original function without noise, we use again the two-norm. The setup for our comparison looks like the following:

- We are using randomly determined functions by using listing 9 with $M = 5$ and $n = 50$.
- Our epsilon is chosen $\epsilon = 1e - 8$.

Applying this setup resulted in an average error of 2.089657 for Algorithm 7.1.1, which determines the average over all ps. Algorithm 7.2.1 resulted in an average error of 1.4728845 using 20 random generated test functions. The exact error values for every test run can be examined in the appendix in Table 5.

The difference in the error values shows that the algorithm which stacks all sub-Hankel matrices on top of each other performs slightly

better. The same can be observed in Figure 6 were the gaps between the red and the blue line are smaller in the third graph than in the second one. However, Figure 6 also shows that Algorithm 7.1.1 does not improve the result at all. The first graph shows the classical Prony method without any handling of noise. Comparing the first two graphs shows that there is no difference at all.

Hence we will continue in the following using Algorithm 7.2.1.

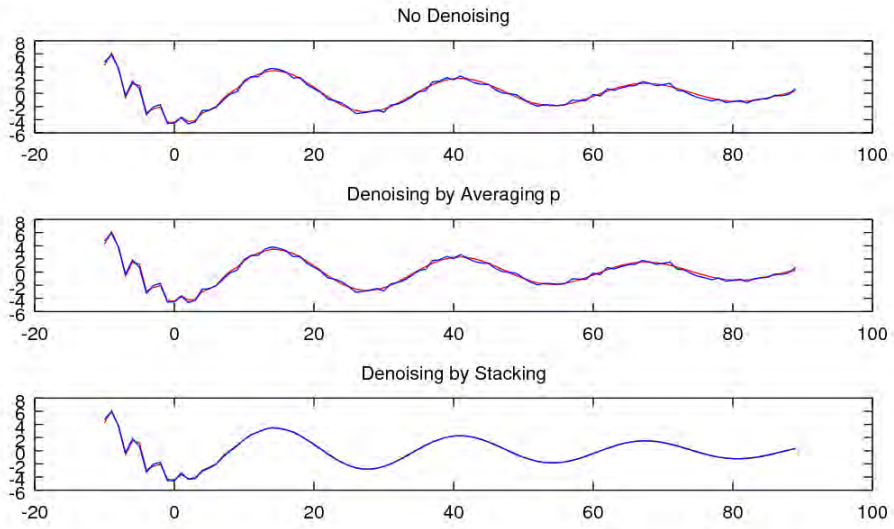

Figure 6: This figure contains three graphs. In general, the red line is the original function and the blue line is the calculated reconstructed. The first graph shows the classical Prony algorithm without handling noise explicitly. The second graph shows the denoising by determining the average Prony polynomial and the third graph shows the denoising by stacking the sub-Hankel matrices on top of each other.

8

SOLVING THE BACKSIDE REFLECTION BY USING PRONY'S METHOD

Chapter 3 already gave a definition of a sinusoidal function

$$c_j \cdot \sin(x + \gamma_j), \tag{76}$$

which is displayed on a screen during the deflectometry. This sinusoidal function consists of an amplitude c_j and a phase γ_j with j, the number of sinusoidal functions (i.e., the number of samples). After the backside reflection formula (1) shows that this sinusoidal function can be split up into

$$\begin{aligned} a \cdot \sin(x + \alpha_j) + b \cdot \sin(x + \beta_j) &= c_i \cdot \sin(x + \gamma_j) \\ \text{with } j &= 1, ..., n \text{ the number of samples,} \end{aligned}$$

with α_j, the phases from the front-side reflection and β_j the phases from the back-side reflection. The corresponding amplitudes are a and b.

Reformulating this formula

$$\begin{aligned} & a \cdot \sin(x + \alpha_j) + b \cdot \sin(x + \beta_j) = c_j \cdot \sin(x + \gamma_j) \\ \Leftrightarrow \quad & \left(\frac{a}{2i} \cdot e^{i\alpha_j} \cdot e^{ix} - \frac{a}{2i} \cdot e^{-i\alpha_j} \cdot e^{-ix}\right) \\ & + \left(\frac{b}{2i} \cdot e^{i\beta_j} \cdot e^{ix} - \frac{b}{2i} \cdot e^{-i\beta_j} \cdot e^{-ix}\right) = \frac{c_j}{2i} \cdot e^{i\gamma_j} \cdot e^{ix} \\ & \qquad - \frac{c_j}{2i} \cdot e^{-i\gamma_j} \cdot e^{-ix} \\ \Leftrightarrow \quad & \left(\frac{a}{2i} \cdot e^{i\alpha_j} + \frac{b}{2i} \cdot e^{i\beta_j}\right) \cdot e^{ix} \\ & - \left(\frac{a}{2i} \cdot e^{-i\alpha_j} + \frac{b}{2i} \cdot e^{-i\beta_j}\right) \cdot e^{-ix} = \frac{c_j}{2i} \cdot e^{i\gamma_j} \cdot e^{ix} \\ & \qquad - \frac{c_j}{2i} \cdot e^{-i\gamma_j} \cdot e^{-ix} \end{aligned}$$

leads to

$$\frac{c_j}{2i} \cdot e^{i\gamma_j} = \frac{a}{2i} \cdot e^{i\alpha_j} + \frac{b}{2i} \cdot e^{i\beta_j} \tag{77}$$

and

$$-\frac{c_j}{2i} \cdot e^{-i\gamma_j} = -\left(\frac{a}{2i} \cdot e^{-i\alpha_j} + \frac{b}{2i} \cdot e^{-i\beta_j}\right), \tag{78}$$

because of the linear independence of e^{ix} and e^{-ix}.

When we take a closer look at formula (77), we observe that

$$c_j \cdot \frac{e^{i\gamma_j}}{2i} \tag{79}$$

can be split up into

$$\sum_{k=1}^{2} \mu_k \cdot e^{f_k}, \text{ with } \mu_1 = \frac{a}{2i}, \mu_2 = \frac{b}{2i}, f_1 = i\alpha_j \text{ and } f_2 = i\beta_j \tag{80}$$

using Prony. The real part of f_k is always zero for sinusoidal functions.
This leads to the following algorithm to calculate the requested αs and βs:

Algorithm 8.0.1 (Decomposition into α and β)
Input: All amplitude and phase samples and an $\epsilon > 0$. The number of samples has to be an even number N with $N >= 4$.

1. *Determining the number of rows n and number of columns m of the images.*
2. *Determining the number of samples N, by counting the number of provided amplitudes or phases.*
3. *For each pixel (k, l):*
   a) *Determine* `gammaj = [Phases{1,1}(k,l); ... ; Phases{1,N}(k,l)]` .
   b) *Determine* `cj = [Amplitudes{1,1}(k,l); ... ; Amplitudes{1,N}(k,l)]`.
   c) *Determine* `cGamma = cj.*[1/(2*i)*exp(i*gammaj)]` *from formula (79).*
   d) *Apply* `ClassicalPronyStepwiseHankel` *from listing (16) to* `cGamma` *to get* `[f,mu,newDegree]`.
   e) *Switch the ordering in f and μ locally if required by using Algorithm 8.0.2.*
   f) *Set β to zero if required using Algorithm 8.0.3.*
4. *Check again if β should be set to 0, but this time more globally using Algorithm 8.0.4.*
5. *When the correct α_1 and β_1 have been determined, it is possible to calculate the remaining α_j, β_j for $j = 2, ..., N$. In doing so, the following has to hold true:*

$$\begin{aligned} \alpha_j &\equiv (j \cdot \alpha_1) \mod 2\pi \\ \beta_j &\equiv (j \cdot \beta_1) \mod 2\pi. \end{aligned}$$

   Hence, to calculate α_j, α_1 is multiplied with j and the resulting value is mapped back to the torus.

Output: αs and βs.

8.0.1 *Switching order of f and μ and setting β to zero using the left Neighbor*

According to formula (80) `[f,mu,newDegree]` looks like the following:

$$\begin{aligned} newDegree &= 2 \\ f &= \begin{pmatrix} 0+i\cdot\alpha_1 \\ 0+i\cdot\beta_1 \end{pmatrix} \\ \mu &= \begin{pmatrix} \frac{a}{2i} \\ \frac{b}{2i} \end{pmatrix}. \end{aligned}$$

However, it has to be considered that the summed elements in formula (80) are commutative. Hence, it can happen that f and μ have switched orders like:

$$\begin{aligned} f &= \begin{pmatrix} 0+i\cdot\beta_1 \\ 0+i\cdot\alpha_1 \end{pmatrix} \\ \mu &= \begin{pmatrix} \frac{b}{2i} \\ \frac{a}{2i} \end{pmatrix}. \end{aligned}$$

To determine whether the components in μ and f are switched, it is required to look at the values of a and b. We have to always ensure that $a \geqslant b$. If this is not fulfilled, we have to switch the coefficients.

This leads to the following algorithm:

Algorithm 8.0.2 (Switching order of f and μ)
Input: f *and* μ.

For each pixel (k, l)*:*

1. *Set:* $\alpha = \mathrm{Im}(f_1), \beta = \mathrm{Im}(f_2), a = |2 \cdot i \cdot \mu_1|$ *and* $b = |2 \cdot i \cdot \mu_2|$.
2. *If* $a < b$*: Switch* α *with* β *and* a *with* b.

Output: α *and* β.

Additionally, we consider the following use case: If α is equal to β it is also possible that β has to be zero. Because it is not possible to differentiate between the case that front- and backside reflection have the same phase or that there is no backside reflection at all. Therefore, we have to again examine b afterwards the switching. Examining the values of α, β and b in those cases shows that the absolute values of α and β are nearly the same and that b is almost zero. Hence, we can set β to zero when b is small enough. Sometimes, it is not sufficient to only use an $\epsilon > 0$ to check if b is small enough. However, we know that two neighboring pixels are always very similar. We can assume some kind of continuity between neighboring pixel values. Hence, we check if b is smaller than a predefined boundary and if the value of β does not fit in regarding a neighboring pixel.

If it is required to set β to zero afterwards, can be determined using the following algorithm:

Algorithm 8.0.3 (Setting β to zero)
Input: α *and* β.

For each pixel (k,l)*: If*

$$\begin{aligned} |b| < \epsilon \quad &\wedge \quad !(|\beta(k,l-1)-\beta(k,l)| < \epsilon \\ &\vee \quad (|\beta(k,l-1)| < -2.9 \wedge |\beta(k,l)| > 2.9) \\ &\vee \quad (|\beta(k,l-1)| > 2.9 \wedge |\beta(k,l)| < -2.9)) \end{aligned}$$

then $\beta(k,l) := 0$.
Output: α *and* β.

8.0.2 *Using the TV-norm for Replacing*

The previously described procedure always uses the left neighbor of the current pixel to determine if the current pixel of β fits in a continuous way or if it should be replaced by 0. In case, the left neighbor was determined wrong all following pixels in that row might be influenced. Hence, it is crucial to add another step to the procedure after α and β are assigned for every pixel. The next step again loops over β, but this time a quadratic area around the current pixel is selected. The size of this quadratic area is provided by an additional parameter. Inside this area we determine whether the value of β or 0 fits better. In doing so, we use the *TV-norm* for this procedure. The TV-norm of an image I helps finding differences and is determined by

$$\mathrm{TV}(I) = \sum_{k=1,l=1}^{n,m} \sqrt{\mathrm{gradX}(I(k,l))^2 + \mathrm{gradY}(I(k,l))^2}, \tag{81}$$

with $\mathrm{gradX}(I(k,l))$ is the gradient of I at pixel (k,l) in x-direction and $\mathrm{gradY}(I(k,l)$ is the corresponding gradient in y-direction [7]. The gradient of an image is basically the difference of two neighboring pixels vertically or horizontally.

This leads to the following procedure which is applied to the first solution of the reconstruction:

Algorithm 8.0.4 (Using the TV-norm for replacing)
Input: α *and* β*. A size* s *for the area where the TV-norm will be applied.*

For each pixel (k,l)*:*

1. *Set*

$$\mathrm{areaBeta}_1 := \begin{pmatrix} \beta_{k-s,l-s} & \cdots & \beta_{k-s,l} & \cdots & \beta_{k-s,l+s} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \beta_{k,l-s} & \cdots & \beta_{k,l} & \cdots & \beta_{k,l+s} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \beta_{k+s,l-s} & \cdots & \beta_{k+s,l} & \cdots & \beta_{k+s,l+s} \end{pmatrix}$$

$$\mathrm{areaBeta}_2 := \begin{pmatrix} \beta_{k-s,l-s} & \cdots & \beta_{k-s,l} & \cdots & \beta_{k-s,l+s} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \beta_{k,l-s} & \cdots & \ddot{0} & \cdots & \beta_{k,l+s} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \beta_{k+s,l-s} & \cdots & \beta_{k+s,l} & \cdots & \beta_{k+s,l+s} \end{pmatrix}$$

$$\mathrm{areaB} := \begin{pmatrix} b_{k-s,l-s} & \cdots & b_{k-s,l} & \cdots & b_{k-s,l+s} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{k,l-s} & \cdots & b_{k,l} & \cdots & b_{k,l+s} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{k+s,l-s} & \cdots & b_{k+s,l} & \cdots & b_{k+s,l+s} \end{pmatrix}.$$

$\mathrm{areaBeta}_2$ *is zero at position* (k,l).

2. *Determine* $\|\mathrm{areaBeta}_1\|_{TV}$, $\|\mathrm{areaBeta}_2\|_{TV}$ *and* $\|\mathrm{areaB}\|_{TV}$.

3. *If*

$$\begin{aligned} |b_{k,l})| &< \epsilon_1 \\ \wedge \quad |\alpha_{k,l} - \beta_{k,l}| &< \epsilon_2 \\ \wedge \quad \|\mathrm{areaB}\|_{TV} &< \epsilon_3 \\ \wedge \quad \|\mathrm{areaBeta}_1\|_{TV} &> \|\mathrm{areaBeta}_2\|_{TV} \end{aligned}$$

then set $\beta_{k,l} = 0$.

Output: α *and* β.

8.1 HANDLING NOISE

If the input does not contain noise, exactly four samples of amplitude and phase are sufficient to reconstruct the two angles α and β. If more than 4 samples (e.g., 6, 8 or 10 samples) are used, the rank r of the used Hankel matrix is still 2. Hence, more input samples provide the same resulting α and β.

This leads us to our main idea behind the denoising. As already described in the previous chapter, we are going to stack $r+1$ packages of the Hankel matrix on top of each other. However, previously r was determined individually for every function, because the functions were generated randomly. In this case, it was possible that not the smallest possible rank was found, because of noise. This time, we know that the functions behave very similarly, because they fulfill formula (76). Hence, we know that the rank of every Hankel matrix is 2. Therefore, each time the Prony polynomial p is calculated out of the Hankel matrix, we use a fixed rank of 2.

However, when noise is present, sometimes it is not sufficient any more to only switch α and β regarding their amplitudes. Noise can influence the amplitudes, so that $a \geqslant b$ might not be fulfilled any more. For this case it is important to move over the final result and check if α and β should be switched using the TV-norm. This leads to the following algorithm:

Algorithm 8.1.1 (Using the TV-norm for reordering)
Input: α *and* β. *A size* s *for the area where the TV-norm will be applied.*

For each pixel (k, l)*:*

1. *Set*

$$\mathrm{areaAlpha}_1 := \begin{pmatrix} \alpha_{k-s,l-s} & \cdots & \alpha_{k-s,l} & \cdots & \alpha_{k-s,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{k,l-s} & \cdots & \alpha_{k,l} & \cdots & \alpha_{k,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{k+s,l-s} & \cdots & \alpha_{k+s,l} & \cdots & \alpha_{k+s,l+s} \end{pmatrix}$$

$$\mathrm{areaBeta}_1 := \begin{pmatrix} \beta_{k-s,l-s} & \cdots & \beta_{k-s,l} & \cdots & \beta_{k-s,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \beta_{k,l-s} & \cdots & \beta_{k,l} & \cdots & \beta_{k,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \beta_{k+s,l-s} & \cdots & \beta_{k+s,l} & \cdots & \beta_{k+s,l+s} \end{pmatrix}.$$

2. *Set*

$$\mathrm{areaAlpha}_2 := \begin{pmatrix} \alpha_{k-s,l-s} & \cdots & \alpha_{k-s,l} & \cdots & \alpha_{k-s,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{k,l-s} & \cdots & \beta_{k,l} & \cdots & \alpha_{k,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{k+s,l-s} & \cdots & \alpha_{k+s,l} & \cdots & \alpha_{k+s,l+s} \end{pmatrix}$$

$$\mathrm{areaBeta}_2 := \begin{pmatrix} \beta_{k-s,l-s} & \cdots & \beta_{k-s,l} & \cdots & \beta_{k-s,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \beta_{k,l-s} & \cdots & \alpha_{k,l} & \cdots & \beta_{k,l+s} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \beta_{k+s,l-s} & \cdots & \beta_{k+s,l} & \cdots & \beta_{k+s,l+s} \end{pmatrix}.$$

   $\mathrm{areaAlpha}_2$ *and* $\mathrm{areaBeta}_2$ *are almost equal to* $\mathrm{areaAlpha}_1$ *and* $\mathrm{areaBeta}_1$*, but the pixels at position* (k, l) *are switched.*

3. *Determine* $\|\mathrm{areaAlpha}_1\|_{TV}$, $\|\mathrm{areaBeta}_1\|_{TV}$, $\|\mathrm{areaAlpha}_2\|_{TV}$ *and* $\|\mathrm{areaBeta}_2\|_{TV}$.

4. *If*

$$\begin{aligned} & \|\mathrm{areaAlpha}_1\|_{TV} > \|\mathrm{areaAlpha}_2\|_{TV} \\ \wedge \quad & \|\mathrm{areaBeta}_1\|_{TV} > \|\mathrm{areaBeta}_2\|_{TV} \end{aligned}$$

   then switch the current pixel of α *and* β.

Output: α *and* β.

8.2 EVALUATION

This section evaluates the proposed procedure by using three different kinds of images of the size 300×400 pixels. For every kind of image we provide 4, 6, 8 and 10 samples. The first two kinds are generated with a maximum wavelength of 96. Originally, the generation does not contain any noise. Hence, after the reconstruction of the *perfect* images, noise is added to test the performance with more realistic conditions. The noise was calculated by adding a random number between $-\frac{1}{4}$ and $\frac{1}{4}$ to the amplitude and phase. The amplitude has a value range of $[0, 0.55]$ and the phase is represented by the torus. During different test runs, we have evaluated that the switching of α and β using the TV-norm performs best with $s = 1$.

The only difference between those two kind of images is in the parts of reflection. In the first image every pixel contains a front- and backside reflection. However, the second image contains also parts without any reflection at all and some parts with only a frontside reflection.

The third kind of image was taken during an experiment from a lens. Hence, we do not have to add noise, because it already contains noise.

8.2.1 *Evaluation of generated basic Images*

The following Table 3 gives an overview over the number of incorrectly calculated pixels in α and β. A pixel is incorrectly calculated

if its value differs less than 10^{-5} from its ground truth. The ground truth is provided, because right now we use generated data.

If no noise is present, the number of errors is very small. The same can be observed in Figure 7, which shows the corresponding reconstruction. Basically, β contains one column which was determined wrong. Examining this column a little closer, we see that at this column α and β were almost the same. In that case it is almost impossible to decide absolutely correct whether β is also α or zero. In this image the procedure sometimes decided incorrectly to set β zero.

Table 3: Number of errors in reconstruction (400×300 pixels)

	No Noise	Noise			
	4 Samples	4 Samples	6 Samples	8 Samples	10 Samples
α_1	0	22519	12765	6288	4661
β_1	528	47769	31000	14160	8938

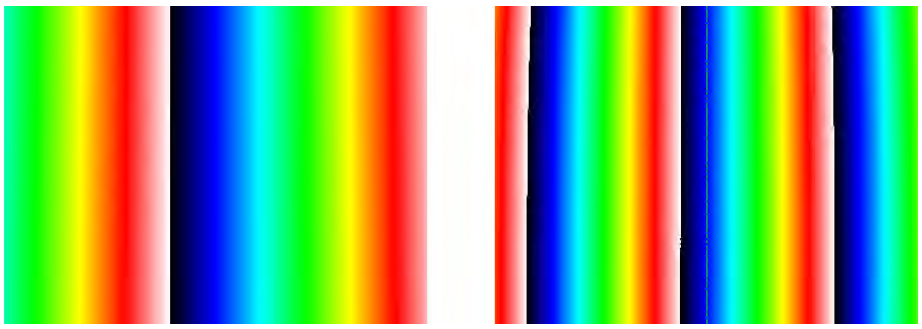

Figure 7: Reconstructed images of α (left side) and β (right side) with maximal wavelength and 4 samples (without noise).

When we have a closer look to the reconstruction of the noisy data in Table 3, we see that the more samples are provided, the less errors occur in the reconstruction. This can also be observed in Figure 8, which shows all reconstructed αs on the left and all reconstructed βs on the right side. Each row represents a different number of used samples.

Comparing the left- with the right-hand side of Figure 8, we see especially at the borders of one color that the noise is bigger, because the values of α and β are sometimes switched. Hence, the calculated values itself are very accurate, but they are too inaccurate to be assigned correctly to the images. This behavior improves with an increasing number of provided samples. Hence, we assume that this behavior improves even more when further samples are provided.

However, in the noisy case we have still the problem with one incorrectly determined column where α and β are equal.

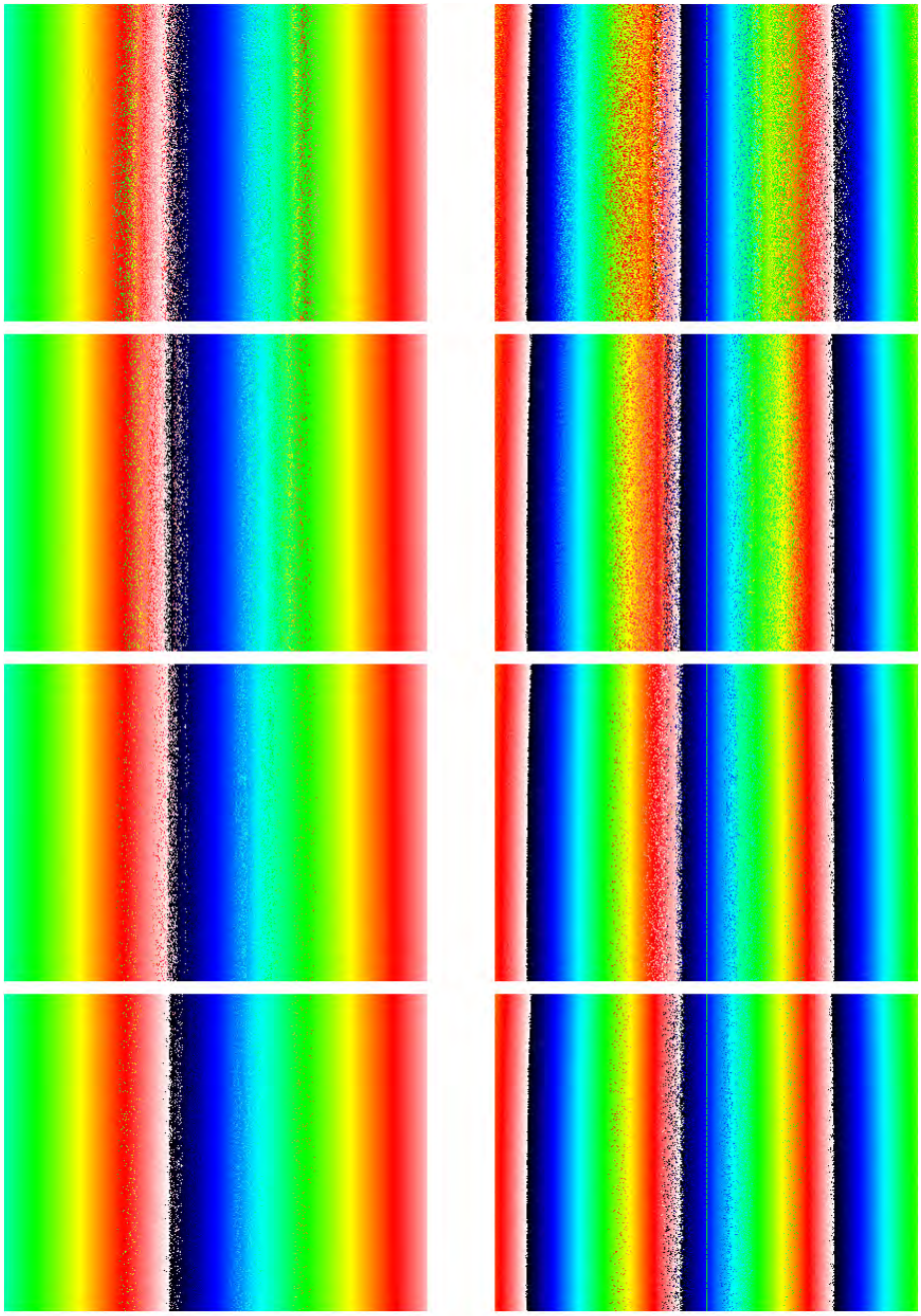

Figure 8: Reconstructed images of 4, 6, 8 or 10 samples (4 samples are in the first row until 10 samples are in the last row) containing noise. On the left side the reconstruction of α and on the right side the reconstruction of β is displayed.

8.2.2 *Evaluation of generated Images containing different Kinds of Reflection*

The previous sample consisted of images where always a front- and backside reflection is present. However, it is common that on parts of the image there is no reflection at all, e.g., when the image also

contains a part of the floor around the glass. Moreover, it is also common that there is only a frontside but no backside reflection. Hence, we apply the same algorithm to images which contain areas without any reflection and areas with only a frontside reflection. The number of errors in reconstruction can be observed in Table 4. Figure 9 shows the reconstruction of this use case without noise. The left image shows again the reconstruction of α and on the right we can see the reconstruction of β. We notice that in the left upper corner there is no reflection at all. Both images show green in this area, which represents zero. Beside this area we see the area where only a frontside reflection occurs. Here we observe that β is in this area almost everywhere zero, except at some points where it was not possible to determine β correctly. In general, this problem can be caused by the fact that the algorithm can not differentiate directly if α and β are equal or if β should be zero in this case. The algorithm is only able to calculate a value for β which is almost equal to the corresponding value of α. Afterwards, it is necessary to determine the correct value for β by evaluating the coefficient b. If b is almost zero β can be set zero. But this procedure depends on limits, which are sometimes not entirely sufficient. In this case it was not possible to change the limit in a way to improve the result.

Table 4: Number of errors in reconstruction (containing areas without reflection and with only frontside reflection on an image with 400×300 pixels)

	No Noise	Noise			
	4 Samples	4 Samples	6 Samples	8 Samples	10 Samples
α_1	0	15718	8288	3998	2832
β_1	5140	40437	29971	16767	13871

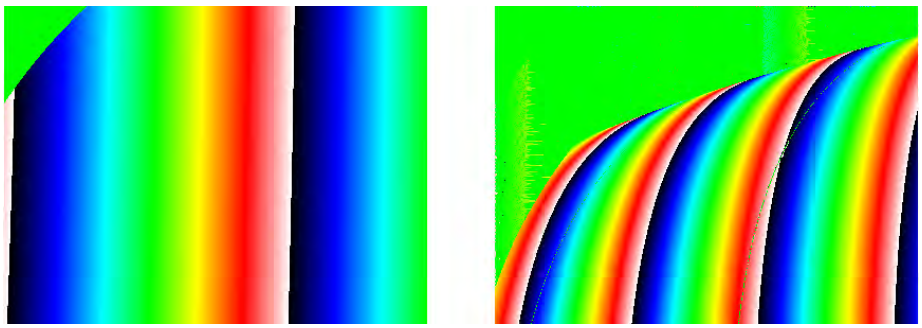

Figure 9: Reconstructed images of α (left side) and β (right side) with maximal wavelength and 4 samples (without noise).

Figure 10 shows how this kind of images handles noise. We see again that noise influences the result less when more samples are

used for the reconstruction. But we still see that there are sometimes points which are assigned incorrectly. The area where β should be zero is denoised as well so that the most of this area is zero. But the parts where pixels are determined incorrectly are now more connected.

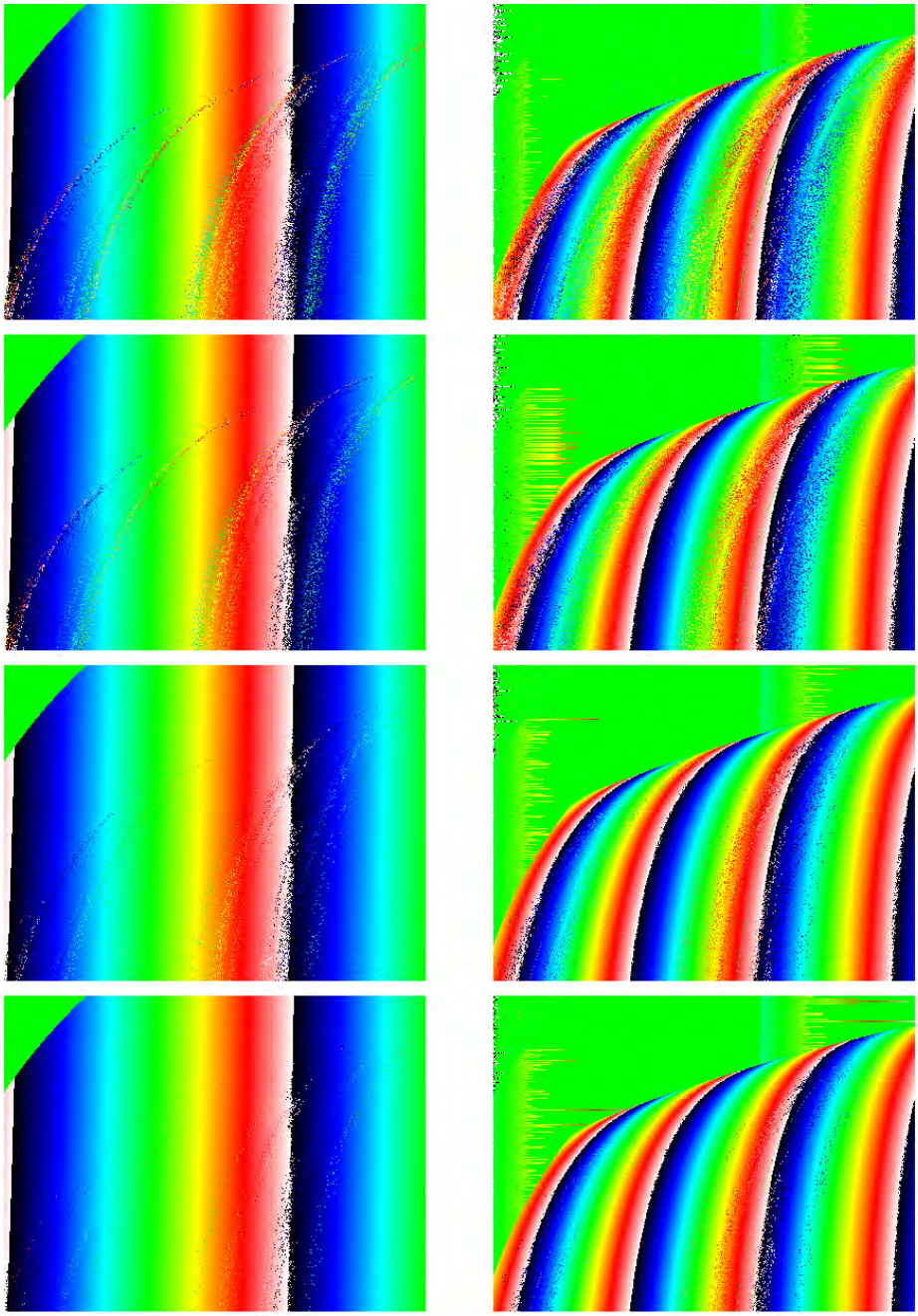

Figure 10: Reconstructed images of 4, 6, 8 or 10 samples (4 samples are in the first row until 10 samples are in the last row) containing noise. On the left side the reconstruction of α and on the right side the reconstruction of β is displayed.

8.2.3 *Evaluation of real Images*

Figure 11 shows the reconstruction of real images, that were recorded during an experiment. Hence, there is no table to evaluate the number of incorrectly determined pixels, because there is no ground truth. This time, we also do not have a relation between the different wavelengths so that we have one maximum wavelength λ_{max} and $\lambda_j = \frac{\lambda_{max}}{j}$. Hence, the resulting images do not show the same image using a different number of samples. The pattern is shifted. However, we can see that the reconstructed images look relatively smooth, except at the area where α switches from $-\pi$ to π. There, we can still see a noise in the reconstruction of α.

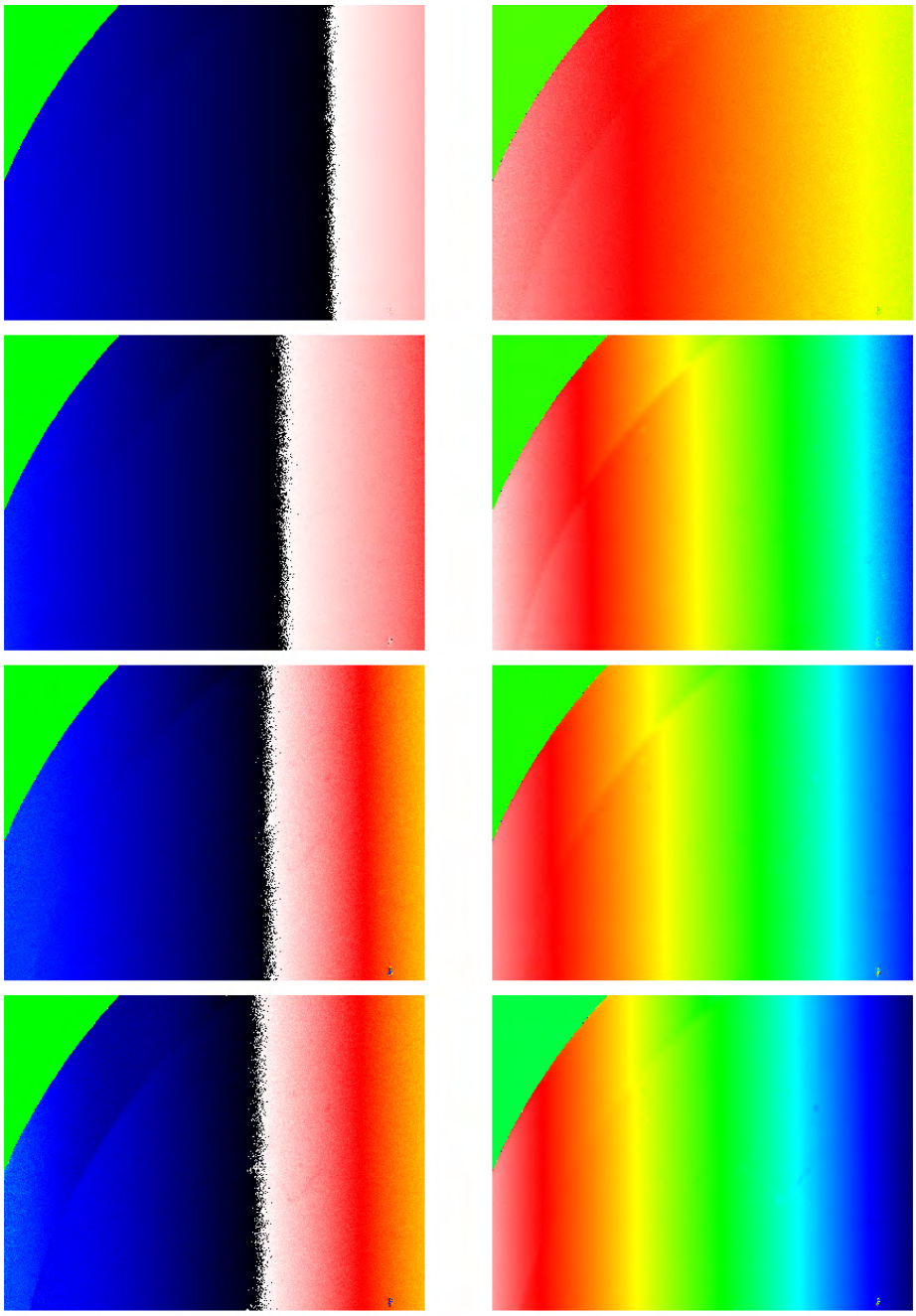

Figure 11: Reconstructed images of 4, 6, 8 or 10 samples (4 samples are in the first row until 10 samples are in the last row) of real data. On the left side the reconstruction of α and on the right side the reconstruction of β is displayed.

Part IV

CONCLUSIONS

9 CONCLUSION

The goal of this master thesis was to solve the problem of backside reflection by splitting up the measured phases into its front- and backside. Therefore, we provided two approaches.

The first approach uses three samples to create a system of quadratic equations. Using those equations it is possible to split up the provided amplitudes and phases into α and β very accurately when no noise is present. However, if the samples contain noise a sufficient reconstruction is not possible. The resulting α and β are almost destroyed.

The second approach uses Prony's method to solve the problem of backside reflection. This time the reconstruction of samples without noise was sufficient and almost as accurate as the first approach. However, this procedure is much more tolerant regarding noise, because a proper reconstruction is possible. Hence, this approach outruns the first one especially under real life conditions.

In general, we can conclude that we found a suitable method to solve the problem of backside reflection using Prony's method. However, this procedure still has potential for improvement. Looking at reconstructions of noisy input data shows that sometimes our conditions are not fulfilled and pixels of the two phases are switched. Until now we are not able to do this with our algorithm.

10 FUTURE WORK

Like already mentioned, we have found a procedure using Prony's method to solve the problem of backside reflection. However, sometimes we are facing difficulties when we are assigning the calculated values to the front- or backside reflection with noisy data. We are able to calculate those values very accurately, but sometimes we are not able to assign them to the correct kind of reflection. In future, we have to focus on a method to assign those calculated values globally to the correct kind of reflection.

Furthermore, the current implementation of the Prony based approach uses Matlab as programming language. This results in a runtime of several minutes to calculate a reconstruction of 400×300 pixels per image. Additionally, nothing is run in parallel until now. Hence, we assume that C++ is in general a better choice regarding performance. Especially, due to the value calculations of the reconstruction being independent to each other, we can run these calculations on GPUs. This should decrease runtime drastically.

Part V

APPENDIX

A

APPENDIX

Table 5: Comparison of the error in Algorithm 7.1.1 and 7.2.1

Test Run	Algorithm 7.1.1	Algorithm 7.2.1
1	1.565	1.4339
2	2.6526	1.50307
3	1.4495	1.1627
4	1.0249	1.10751
5	1.2361	1.1260
6	1.1104	1.2075
7	3.6609	2.0344
8	0.69813	0.67921
9	2.3032	1.4580
10	1.4998	1.3359
11	0.80171	1.2763
12	2.5664	1.3686
13	4.8923	2.4177
14	2.1361	1.8085
15	2.2998	1.4197
16	1.7188	1.4196
17	1.4831	1.0650
18	1.9818	1.4359
19	3.2293	2.5332
20	3.4833	1.6650

BIBLIOGRAPHY

[1] Gisela Jordan-Engeln and Fritz Reutter. *Numerische Mathematik für Ingenieure*. Bibliographisches Institut Mannheim, 1978.

[2] Markus C Knauer, Jurgen Kaminski, and Gerd Hausler. "Phase measuring deflectometry: a new approach to measure specular free-form surfaces." In: *Photonics Europe*. International Society for Optics and Photonics. 2004, pp. 366–376.

[3] Gerlind Plonka and Manfred Tasche. "Prony methods for recovery of structured functions." In: *GAMM-Mitteilungen* 37.2 (2014), pp. 239–258.

[4] Daniel Potts and Manfred Tasche. "Parameter estimation for nonincreasing exponential sums by Prony-like methods." In: *Linear Algebra and its Applications* 439.4 (2013), pp. 1024 –1039.

[5] Gaspard C. de Prony. "Essai éxperimental et analytique: sur les lois de la dilatabilité d fluides élastique et sur celles de la force expansive de la vapeur de l'alkool, à différentes températures." In: *Journal de l'École polytechnique Floréal et Plairial* 1.22 (1795), pp. 24–76.

[6] Holger Rapp and Christoph Stiller. "Deflektometrische Methoden zur Sichtprüfung und 3D-Vermessung voll reflektierender Freiformflächen." In: *Forum Bildverarbeitung*. Vol. 2010. 2010.

[7] Leonid I Rudin, Stanley Osher, and Emad Fatemi. "Nonlinear total variation based noise removal algorithms." In: *Physica D: Nonlinear Phenomena* 60.1-4 (1992), pp. 259–268.

[8] Tomas Sauer. "Prony's method: an old trick for new problems." In: 2015.

[9] Ron Synowicki. "Suppression of backside reflections from transparent substrates." In: *physica status solidi (c)* 5.5 (2008), pp. 1085–1088.

[10] Alexander Zimmermann. FORWISS, 2016. URL: https://www.forwiss.uni-passau.de/de/.

DECLARATION

Hiermit versichere ich, dass ich diese Masterarbeit selbsständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder singemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, October 2017

Miriam Marx