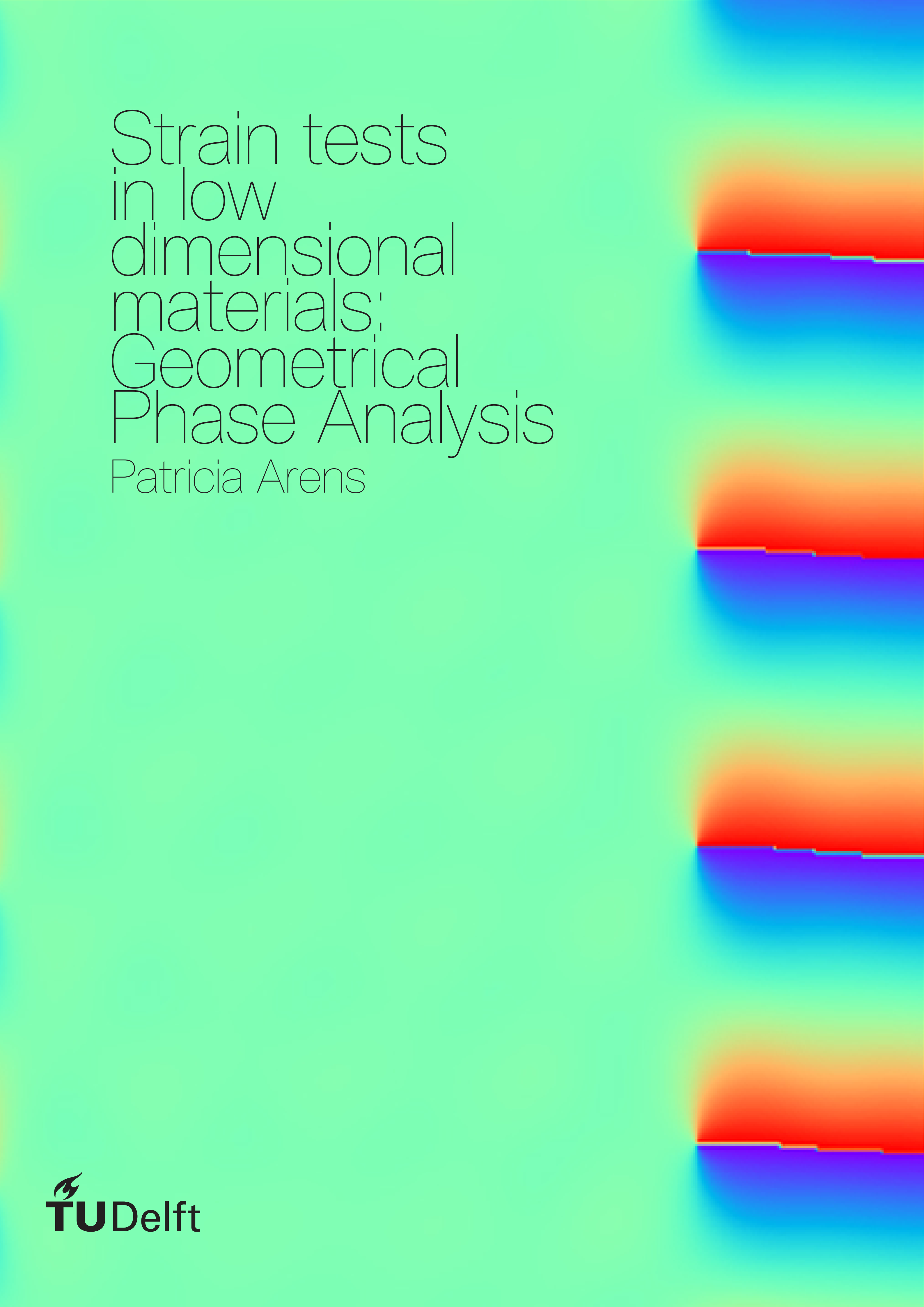

Strain tests in low dimensional materials: Geometrical Phase Analysis

Patricia Arens

TU Delft

Strain tests in low dimensional materials: Geometrical Phase Analysis

by Patricia Arens

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Monday August 29, 2022 at 12:00 PM.

Student number:	5072353	
Project duration:	March 1, 2022 – August 29, 2022	
Thesis committee:	Prof. dr. Sonia Conesa-Boj,	TU Delft, supervisor
	Prof. dr. Paul M. Visser,	TU Delft, supervisor
	Prof. dr. Yaroslav Blanter,	TU Delft
	Prof. dr. Bas Janssens,	TU Delft

TU Delft

Abstract

In the last decades there has been an increasing interest in computing the local strain at the atomic scale of materials. By knowing aspects of the local strain in a lattice, one has information about measurements of distortions of lattice parameters concerning shifts, deformations and defects computed with respect to a smooth, defect-free reference region. Multiple methods have been implemented so far in order to map the strain of two-dimensional lattice patterns, which are obtained through means of a High Resolution Electron Microscope (HRTEM). The functioning of a HRTEM is based on the same principles as an optical microscope, but it uses a beam of electrons instead of visible light.

One of the computational methods which then processes the obtained two-dimensional images is called the Geometrical Phase Analysis (GPA) and makes use of a very important mathematical tool, the Fourier transform. The GPA method lies at the center of this project and consists of several steps. First, the Fourier transform of the lattice image is plotted and two Bragg peaks corresponding to two linearly independent frequency vectors in the power spectrum are chosen. Next, a mask is applied around these peaks, separately. In my project I have chosen to apply the Hann smoothing filter. Then, the inverse Fourier transform is applied to the masked image and the phase (also called the raw phase) is plotted. The next step is to compute the reduced phase, which is defined at a local pixel as being the raw phase from which the following product is subtracted: $2\pi\vec{g}\cdot\vec{r}$, where $\vec{r}$ is the vector corresponding to a pixel in the real space and $\vec{g}$ the frequency vector corresponding to the Bragg peak around which one has applied the mask. At this point the reference region is computed by choosing a smooth, homogeneous area in the reduced phase image. In order to obtain the strain, one needs an optimal frequency vector $\vec{g}$ defined at every pixel of lattice. In order to do so, a minimization process defined in the context of a computer algorithm in the programming language Python has been implemented. These computations should lead to obtaining the lattice strain, which is calculated by taking the symmetric part of the derivation of the displacement obtained from the two linearly independent Fourier components, which is in turn called the distortion. The antisymmetric part of the distortion is the rotation component and serves as a check for the correctness of the computational method.

The goal of my project is not only to provide a solid theoretical background for the GPA method and to discuss the strain at atomic level in several lattice patterns, but to also provide a rigorous computer algorithm that makes these computations reality. This algorithm, opposed to pre-existing software, facilitates the reader's process immensely by the large amount of detail which is given at every step, detail which easily motivates and supports the reader in potential side-steps they would want to take in order to make the method their own. Unfortunately, at the moment of the discussion of this thesis I have not yet been able to solve an issue which does not allow the correct output of optimal values for $\vec{g}$ at every point of the lattice.
I want to take upon me the task of fixing my computer algorithm in the context of my future Master thesis, in order to give the reader more flexibility and more additional features than most computer algorithms and software which already exist do (concerning the use of the GPA method to calculate the strain).

Contents

1 Introduction

High-Resolution Transmission Electron Microscopy (also known as HRTEM) yields direct images of crystallographic structure of materials taken at an atomic structure. It is an extremely powerful tool to obtain several kinds of information of the atomic structure and properties of materials, at a nanoscale [5]. The principle on which the creation of images is based is the phenomenon of interference [5]. The high resolution of the images which one gets through this technique is a very elegant and indispensable tool which can be used to investigate properties of atoms, crystal structures, defects in materials which find their root at the atomic scale, but also crystallographic quantities such as lattice strain. Local lattice strain is defined as being the symmetrical part of the local distortion field, obtained by differentiating the displacement field (the displacement between the pixel which one is performing the calculations at with respect to a previously chosen reference region in the lattice)[18]. In the context of this project it is useful to be able to visualise the strain. I therefore also introduce the concept of strain mapping, a numerical image processing technique that measures the local shifts of image details around a crystal defect with respect to the ideal, defect-free, positions in the bulk [7]. The fact that one is able to identify these displacements in a dot-like image. Moreover, in two-dimensional HRTEM images, the local lattice is characterized by the tangent planes of two geometrical phase images, which, as will be shown later throughout the project, is an important step of the method used to compute the strain. Therefore, HRTEM is without a doubt the tool one needs in order to obtain the lattice images and to later compute the strain.

It is highly important to be able to quantify the strain in a material, due to several reasons, all related to the fact that being able to have a correct quantitative measurement of strain on the nanometre scale is significant for many studies of materials. For instance, a crucial element in diode-laser fabrication is matching the atomic spacing of successive layers. Semiconductors can accommodate small differences in their atomic spacing, which leads to the producing of strain at the level of the lattice of the crystal [11]. For the correct functioning of the semiconductor, one must be able to quantify this strain in order to make sure that it is not too large to cause damage. Another application could be the computation of a limit in this case: what is the highest possible strain which a semiconductor can support at lattice level in order to not malfunction?

This project is centered around obtaining the strain in certain lattice patterns representing single-layered materials. The reason for this choice is the fact that in the last few decades there has been an increasing significance in performing strain measurements at the atomic scale. The 2D images which are normally obtained through HRTEM are simulated in this project with the help of coding tools in the programming language called Python [20]. The method which is applied in order to compute the strain in the considered lattices is called the Geometrical Phase Analysis Method (GPA) [18], a very elegant series of mathematical steps of digital signal processing. The GPA method works in Fourier space (also called reciprocal space). If one takes the Fourier transform of a HRTEM image and applies a filter to it centered on one of the Bragg peaks and performs the inverse Fourier transform subsequently analyzing the phase image of this transform (which will later offer information on the displacement fields) they have described in big lines the first steps of the GPA methodology [18]. The elegance of the GPA method lies in the fact that it is able to identify with precision the location Bragg peaks (in other words the length of the Bragg-vectors in reciprocal space), a very important aspect which allows an accurate computation of the strain in a certain pixel with respect to a chosen reference region. The later is computed by taking the derivative of the displacement with respect to the corresponding lattice coordinate. It is important to note that in order to obtain the displacement in 2D images one needs two non-colinear vectors from the reciprocal space. After having computed the strain in one point one can then proceed to constructing a matrix of the entire strain field and plot it [18], [14].

The scope of this project is to analyze the previously mentioned steps of which the Geometrical Phase Analysis consists, apply them to several lattice patterns in order to eventually obtain the strain at the subatomic level of these materials, all of this computed with the aid of an original computer program in Python created by me, with the goal of facilitating the reader in the process of understanding by adding not only physical, mathematical and computer science support, but also to clarify the main steps of GPA method. The main computational path follows the outline of [18]. The motivation of the project rises primarily from the desire of having a clear version of steps of which the GPA consists, starting by rigorously explaining how the lattice simulations are obtained and how they are made in order to closely resemble a HRTEM image. Moreover, systematic explanations are added to several steps which are not clear enough from a mathematical or computational point of view. Furthermore, this project offers the complete code which was used in order to obtain the desired results. Challenges which were encountered along the way are thoroughly explained in order to help the reader not only with the potential issues which arise along the way, but also with how to solve them.

After having read the thesis and by using the presented code, the reader will not only be able to reproduce the calculations, but also to create their own desired lattice pattern and to apply the GPA method to it, in order to obtain the wanted strain field, and gather information on the properties of the atomic scale of the material in question. Unfortunately, at the time of the discussion of the thesis I have not yet managed to solve some of the issues in my code, which therefore does not yet fully express its interesting potential. This is because of a discordance between a mathematical function and its corresponding version used in the computer algorithm that lies at the basis of the computational method. Nevertheless, the strain that one should obtain through means of the defined lattice patterns is calculated and shown in images obtained with the aid of a computer algorithm previously implemented in C++ [13]. I am planning on implementing an updated version of my code in the context of my Master thesis in order to make it fully operational. Nevertheless, the elegance of my code, even at this moment, lies not only in its originality, but also in the fact that the methods regarding the intermediary steps are not set in stone, as generally is the case with already implemented GPA software. Therefore, the reader can choose the preferred technique while computing specific intermediary steps, while still being guided by the main lines of my code.

It important to note that even though the GPA method is very efficient in quantifying the strain, there are several other techniques, one of which is worth introducing in the context of my project: the Peak Pairs algorithm (PP) [8]. The peak-finding method, opposite to the GPA method, works in real space by superimposing the reference lattice to the experimental lattice. The experimental lattice is constructed from the intensity images present in the HRTEM image. One then proceeds to calculate the local displacement field at each node of the superimposed image, from which, just as is done in the context of my project, one obtain the strain field. One can see that these two techniques, while being similar, also have their differences. For instance, seeing as the PP method does not involve the computation of a two-dimensional Fourier transform, it presents much less memory requirement than the GPA method. If one, however, is not interested in the performance of the computer algorithm, then the GPA method might be more suitable for them seeing as this one has been proven to be more precise in determining strain fields corresponding to lattice defects. Applying the PP algorithm to the HRTEM images which I have simulated with my Python code and confronting the results obtained through the GPA method is a project plan for the near future, in order to test the previously stated claims [8].

Chapter 1 shines some light on the state of the art regarding GPA and local strain, the motivation and the context in which this project has come to exist. As stated before, in order to compute the strain, I implemented a Python code following the theoretical steps outlined in literature [18]. In the code I have also used pre-implemented functions, introduced in the Github repository corresponding to [14]. Chapter 2 contains information regarding mathematical and physical tools needed to perform the computations, the later being outlined in Chapter 3. In Chapter 4 I add my own partial results and I argue where I expect the error to be. The computer code can be found in A, alongside a table of symbols and variables. Chapter 5 gives the final remarks on the results, the computational method and future projects.

2 Theory

This chapter is dedicated to introducing the necessary theory on which my project is based. It contains information on mathematical and physical tools, presented in different sections.

2.1. Geometry of solids

In this section several notions regarding the geometry of solids are introduced. Taking into account the fact that in this project the simulated materials can be approximated as being two-dimensional (because in one direction I consider the specimen in question to be monoatomic), it is enough to give only the definitions corresponding to two dimensions.

2.1.1. Lattices

By definition, a lattice is an infinite set of points defined by integer sums of a set of linearly independent lattice vectors, which in turn are the vectors that span the lattice [19]. In order to properly define a two-dimensional lattice, one needs two lattice vectors which are linearly independent [9]. The condition of linear independence refers to the fact that one vector cannot be a multiple of the other. In other words, no vector can be written as a linear combination of the other one. Lattices are found in the so-called real (or direct) space seeing as it is a concept of describing material's structures directly [19], [9]. Later in this chapter the term reciprocal space will also be introduced. During the entire project there are many references to these two space, therefore it is important to distinguish amongst them.

In equation 2.1 one can see the way the lattice points are defined, as a function of a couple of integers n_1 and n_2 and of the lattice vectors $\vec{a_1}$ and $\vec{a_2}$. These vectors are not uniquely defined.

$$\vec{R} = n_1 \cdot \vec{a_1} + n_2 \cdot \vec{a_2} \tag{2.1}$$

Next I am introducing the square lattice, one of the main focuses of this project, which has the property of being translationally symmetric. The symmetry reason, the compactness of the lattice and the convenient geometry that lies at its basis are some of the reasons why the square lattice is such an elegant and stable structure and is found in nature in many chemical elements and components, by taking a monoatomic layer of structures such as Iron [12], Scandium Nitride [15], some transition metal oxides such as Titanium Oxide (TiO), Vanadium Oxide (VO) and Manganese Oxide (MnO) [10].

The use and investigation of square lattices in this project are hereby also justified.

2.1.2. High Resolution Transmission Electron Microscopy (HRTEM)

Now that the concept of a lattice has been clarified, the following question might rise: how can one observe then with a high resolution? High resolution transmission electron microscopes have the scope of forming images by using an electron beam, in a very similar way to how optical microscopes form images, by means of visible light. Firstly, the beam of electrons is transmitted through a thin specimen [3]. This is very important specification seeing as in the case of this project the specimen is always a

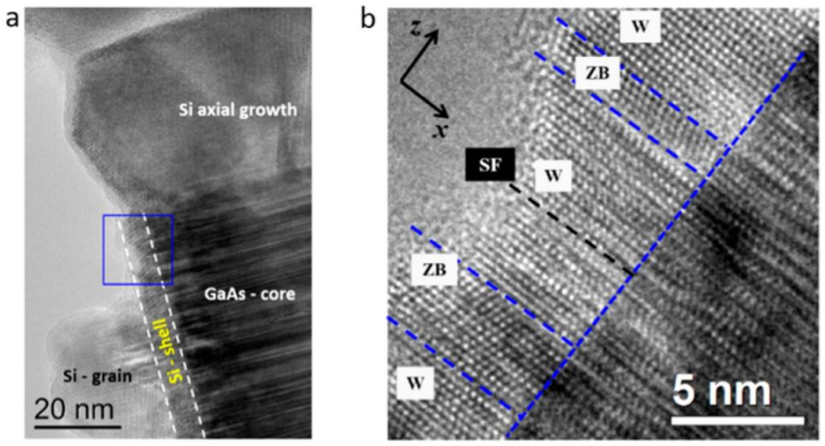

Figure 2.1: Adapted from [5]. (a) Bright-field TEM image revealing the axial and radial silicon growth on a GaAs core, as well as the presence of the Si crystalline grains on the sidewalls of the core−shell system. (b) High-resolution TEM image of the region marked with a blue square in (a)

two-dimensional material. The system of lenses then focuses and magnifies the beam of electrons in order to then project the final image which is then saved. The magnifications which are used in the process are high enough for one to easily be able to see the lattice spacing of inorganic materials. These dimensions can typically be as low as a fewÅ (in the order of 10^{-10}m) [3]. An image taken with a HRTEM can be seen in figure 2.1 [5]. This project simulates similar images to this one (see Chapter 3 for more details). It is of high importance to also note the fact that an HRTEM image is a discrete periodic image.

2.2. Fourier transform

This operation lies at the basis of every computation performed in this project, therefore it is one of the most important concepts introduced in this chapter.

By definition [2], the Fourier Transform's role is that of converting functions which depend on spacial variables into functions that depend on spacial frequencies. With other words, one converts between the real space and the reciprocal space through the Fourier transform. In case of a signal, the Fourier transform decomposes its waveform into peaks, which in turn contain information on the specific frequencies that make up the original signal.

Suppose one has a one dimensional signal described by $f(x)$ (x being a spacial variable and f an arbitrary function), then its continuous Fourier transform has the form:

$$F(\omega) = \int_{-\infty}^{\infty} f(x)e^{-i2\pi\omega x}dx \tag{2.2}$$

In equation ω refers to the spacial frequency variable corresponding to the Fourier space, i is the imaginary unit (in the complex plane), F refers to the Fourier function which is obtained after applying the Fourier transform to the spacial variable x. In the case of an image, however, one has 2 dimensions and therefore also 2 variables. Let's denote the frequency corresponding to the x variable in the real space with h and the one corresponding to the y variable with k. Therefore, in the Fourier space, each pixel has the following coordinates: (h, k). For example, figure 2.2 one can see the Fourier transformation of the second subplot of figure 2.1. Therefore, the Fourier transform for this later case (where x and y are the spacial variables) is also defined, for an arbitrary function $f(x, y)$:

$$F(h, k) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y)e^{-i2\pi(hx+ky)}dxdy \tag{2.3}$$

In equation 2.3 F is once again the transformation of the arbitrary function f in the real space to its corresponding value in the frequency space. Now the integral must be taken over two dimensions,

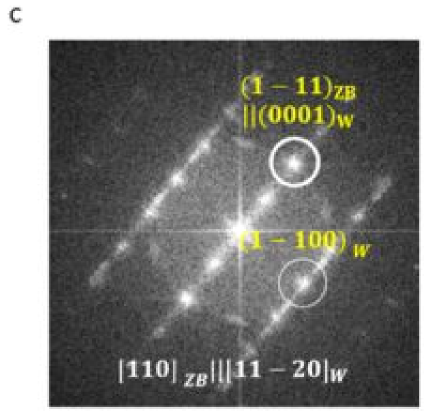

Figure 2.2: Adapted from [5]. Fourier transform corresponding to figure 2.1 (a). The bright Bragg peaks are noticeable and correspond to the bright spots indicating a high value of intensity

leading to the appearance of a double integral in the equation. As mentioned before, h and k are the frequencies corresponding to the Fourier space (also called the power spectrum). It is clear that the Fourier space of a two dimensional image is also a two dimensional image. It is important to note that one refers to the Fourier space also as the power spectrum or k-space. These namings are used interchangeably throughout the report when referring to the result which one obtains after performing the Fourier transform

2.3. Inverse Fourier transform

The inverse Fourier transform, as the name suggest, is the inverse operation to the mathematical tool defined in the previous section: the Fourier transform. In the context of this project it is applied to an image in the Fourier space in order to transform it back to the real space and is defined by the following formula:

$$f(x,y) = \frac{1}{4\pi^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(x,y)e^{i2\pi(hx+ky)}dhdk \tag{2.4}$$

It is important to note that in order to get the discrete Fourier transform and inverse Fourier transform it is enough to replace the integration and double integration, respectively by a double sum. With other words, one changes the infinite summation with a finite summation. This is done for equations 2.3 and 2.4.

2.3.1. Reciprocal lattices

The reciprocal space is a mathematical construct which immensely facilitates several computations in the domain of Physics. It is generally used in order to describe wavelike phenomena in crystals. For example, a very well-know phenomena which makes use of the reciprocal space is X-ray diffraction, a technique which allows the observation of material at an atomic level.

By definition, the reciprocal lattice is the Fourier transform of the real (or direct) lattice [19]. Similar to equation 2.1, one can also define an equation which describes the lattice points of the reciprocal lattice:

$$\vec{G} = m_1 \cdot \vec{b_1} + m_2 \cdot \vec{b_2} \tag{2.5}$$

In equation 2.5 m_1 and m_2 are integers, $\vec{b_1}$ and $\vec{b_2}$ are the vectors describing the points in the reciprocal lattice.

There is an important relation between real and reciprocal lattices: $e^{i\vec{G}\cdot\vec{R}} = 1$ (in this exponential function i is the complex unit number), or $\vec{a_i} \cdot \vec{b_j} = \delta_{ij}$, where δ_{ij} is the Kronecker Delta function and has value

1 when $i = j$ and value 0 when $i \neq j$. (see 2.1.1 and 3.5 for explanations regarding $\vec{G}$ and $\vec{R}$)

2.4. Bragg peaks

As seen in the previous section, the GPA method relies heavily on the selection of Bragg peaks (which so far have been called g-vectors), I believe it is important to shine some more light on this topic as well. A Bragg peak, which has already been introduced as a high intensity peak in the reciprocal (Fourier) space, is also a particular set of sinusoidal lattice fringes [4]. This g-vector has has a length which is equal to to the reciprocal of the lattice fringe spacing (which corresponds to the the lattice plane). A lattice fringe is a periodic fringe in a (high resolution) transmission electron microscope image and is formed by two waves: a wave which is transmitted through the crystal and a wave diffracted by one lattice plane of the crystal. A lattice plane is a plane whose intersections with the lattice are periodic.[1].

2.5. Geometrical Phase Analysis method (GPA)

The Geometrical phase analysis is a powerful mathematical tool (consisting of a series of operations) which allows the computation of distortion, strain and rotation in lattices visible in images obtained through means of a HRTEM. As the name suggests, rotation is the amount with which certain atoms are rotated with respect to a certain reference region. Lattice distortion (also called microstrain) represents the departure of the atom position from an ideal structure (also called reference region in my case). I will continue this section by introducing the main steps of which the GPA method consists and some necessary theory needed to understand them. These steps are followed chronologically in the next chapter and are the basis of the computational method. Therefore, the next chapter also provides additional information which are presented succinctly below as an overview.

1. Fourier transform: After the lattice has been obtained, the Fourier transform is applied to it. According to which lattice is being used as the object in real space, a number of peaks corresponding to high intensities will appear in the Fourier image. These peaks are high-valued frequencies from the power spectrum. The idea is to obtain the coordinates of these peaks. These coordinates, defined by pixels (h,k), can also be written in a more compact what in the form of vectors, which are going to be called throughout this report g-vectors. For convenience, I will however only take the real part of these values and use the following definition: $\vec{g} = h \cdot \hat{h} + k \cdot \hat{k}$, where $\hat{h}$ and $\hat{k}$ are, respectively, the unit vector corresponding to the axis of the Fourier image.

2. Peak selection: Seeing as I am working with two dimensional spaces, in order to define an appropriate basis, two g-vectors which are linearly independent must be selected. These peaks are also referred to as Bragg peaks (see next section for more detail)

3. Masking around one peak: A mask is applied to the Fourier image centered around the previously chosen peak. The mask could be a Gaussian Lorentzian, Hanning, or other (smoothening) functions.

4. Inverse Fourier transform and raw phase plot: of the previously masked image in the Fourier space the inverse Fourier transform is taken and its raw phase (angle) is plotted

5. Second peak: The process is repeated for the other selected peak

6. Reduced phase: The reduced phase for both images corresponding to the raw phase images are calculated (see formula 3.4 for further detail on what the reduced phase and how it is defined)

7. Reference selection: Seeing as the strain must always be computed with respect to a reference, one must be selected. This is done by observing the reduced phase image and choosing a homogeneous area.

8. Computation of optimal value of g: in step 2, the peak selection has been performed more or less in an approximate manner. However, the GPA method offers the possibility of computing optimal g-vectors corresponding to each pixel of the image. This is done through a minimization process (see section 3.7 for more detail)

9. Strain computation: Through means of the previously computed g-vectors and other operations the strain is computed (see section 3.8), as well as distortion and rotation.

2.6. Lattice strain

Computing the lattice strain in between layers of different materials is the goal of this project, therefore, it is of high importance to introduce the concept of lattice strain, which I will refer to throughout this project as simply strain, for convenience. One can define the strain, often denoted by , a two by two matrix, as being a measure of the distribution of lattice constants arising from crystal imperfections, such as lattice dislocations [21].

There is an important relation which is established between the phase of at a certain pixel $\vec{r}$, the g-vector and the displacement found at a particular pixel of the lattice: $\vec{P}_{\vec{g}}(\vec{r}) = -2\pi\vec{g}\vec{u}(\vec{r})$. In this equation $\vec{P}_{\vec{g}}(\vec{r})$ is the phase at $\vec{r}$ calculated by taking the mask around a vector . $\vec{u}(\vec{r})$ is called the displacement of a certain pixel ($\vec{r}$) of the lattice with respect to a chosen reference region. By computing the optimal value of through the GPA algorithm and the phase value (see previous section), one can then determine the displacement vector, and in turn the strain. Further information on how the strain is defined mathematically using these parameters and applied in the case of this project can be found in section 3.8.

3
Computational method

This chapter is dedicated to presenting the steps in chronological order of which the computational method consists.

3.1. Defining the lattice patterns

The first step in computing the strain for several virtual materials is constructing the respective desired lattices. These lattices are the virtual representation of the HRTEM images which have been introduced in Chapter 2. There are various patterns which are being analysed, as it is listed below. All these virtual lattices have been created with the aid of a computer program written in Python in the form of a Jupiter notebook. The variables introduced in this chapter, as well as the formulas, are all similar (and in some cases identical) to the ones used in the writing of the code (see appendix A).

1. The first and most simple pattern is a 500 by 500 pixels image, consisting of a single layer of a square pattern of atoms, as can be seen in figure 3.1. The coordinates span from 0 to 499 pixels (horizontally, as well as vertically) and the simple square lattice is generated by rotating with 90° four k-vectors (see Chapter 2). The function which is used to create the lattice in figure 3.1 depends on the following parameters:

   - r_k, the length of the lattice vectors in the k-space ($\vec{r_k}$). Larger values for r_k automatically imply smaller real space lattice constants.
   - θ, the angle which the first lattice vector makes with the positive horizontal
   - $size$, the length of the side of the square lattice
   - $shift$, the shift which can be introduced in the positions of the atoms, horizontal, vertical or both. In figure 3.3 one can see a Gaussian shift performed in the vertical direction, given by formula 3.1, which is used to obtain another type of lattice pattern.

   In this case r_k is the number of atoms on one row (or column), divided by the number of pixels on that respective row (or column). Therefore, choosing $r_k = 0.1$ in this first simple pattern, leads to the existence of 50 atoms on each row and column, leading to 2500 atoms in total. Moreover, $\theta = 0$ for this simple lattice, $size = 500$ pixels, $shift = 0$. Seeing as this pattern consists of only one layer with atoms arranged perfectly, one does not expect for there to be any strain in this particular case.

2. In the second pattern another layer of atoms is introduced. The new lattice consists for the left half, in other words for the first layer (seen on the horizontal axis in the region 0-249 pixels) of the simple square lattice and for the right half, in other words the second layer (seen on the horizontal axis in the region 250-499 pixels) of the same simple square lattice, but rotated to the right by 6°. The right half therefore consists of the previously introduced lattice, with the only difference being that $\theta = 6°$. Now, seeing as there are two layers, one expects for there to be a nonzero strain

in between the layers, which is later checked in Chapter 4. Moreover, $size$=500 and $shift = 0$. This second pattern can be seen in figure 3.2.

3. The third pattern which is analysed, similar to the second one, also consists of two layers and is created in the following way: image 3.2 is taken and in the tilted part of the image (right vertical half) the atoms are shifted vertically with the amount given by the quantity "x_{shift}", defined in formula 3.1. With other words, it is identical to the second pattern, but now $shift = x_{shift} \neq 0$.

$$x_{shift} = 0.5 \cdot x_c \cdot e^{-0.5 \cdot \left(\left(\frac{x_c}{\frac{S}{4}} \right)^2 + 1.2 \cdot \left(\frac{y_c}{\frac{S}{3}} \right)^2 \right)} \tag{3.1}$$

In equation 3.1 "x_c" and "y_c" are, respectively, the x and y coordinates of the particular atom in the pattern which is being shifted (stretched or compressed). The variable "S" represents half of the length of the whole image (in the case of this report it always holds that $S = 500/2 = 250$). This third pattern is visible in figure 3.4. Formula 3.1 has been chosen, amongst other reasons, for the fact that it is a 2 dimensional Gaussian function, with the property of being smooth. Moreover, the parameters are chosen such that the displacement becomes 0 near the edges. Furthermore, anticipating the second step of which the GPA method consists (computing the Fourier transform for the lattices) another interesting property of the Gaussian function which led to it being used is the fact that the Gaussian function is an eigenfunction of the continuous Fourier transform.

4. Especially after having performed more steps of the GPA method (see Chapter 2), one can see that for the previous three patterns (figure 3.1, figure 3.2 and figure 3.4) several discontinuities and abrupt decays occur at the edges of the lattices. In order to avoid this, the forth pattern which I am introducing in this section is similar to the one presented in figure 3.4, with the addition of a window spanning several pixels from each edge.

The window which is applied is subject to the following function:

$$window(x, \delta) = \begin{cases} sin(\frac{\pi \cdot x}{2\delta})^2, \text{ if } |x| < \delta \text{ or } |x| > 500 - \delta \\ 1, \text{ else} \end{cases} \tag{3.2}$$

In equation 3.2, x refers to the spacial coordinate and δ is the amount of pixels from the edge on which the window is applied. In order to apply it, it is sufficient to multiply the function $window$ in both the x and y directions with the pattern which one wants to transform. This fourth pattern with smooth edges is portrayed in figure 3.6. This particular image has a window spanning 10 pixels away from each edge towards the center of the lattice. (In other words in figure 3.6: $\delta = 10$)

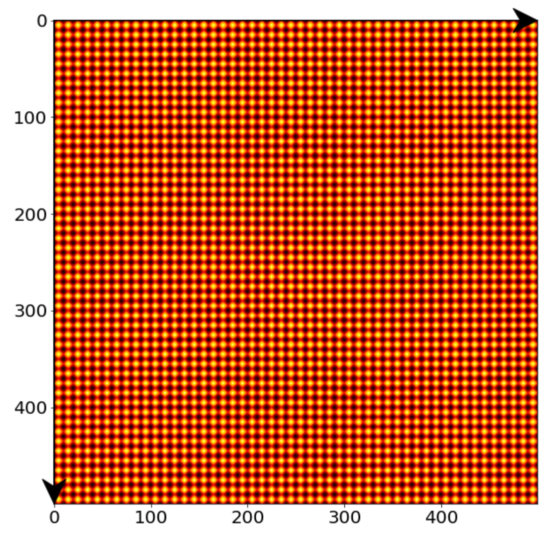

Figure 3.1: Original square lattice, defined with a size of 500 by 500 pixels, depicting 50 by 50 atoms, corresponding to $r_k = 0.1$ One can see the x axis pointing downwards and the y axis pointing to the right. This lattice presents no noise, no shift and no tilt. Defining this lattice is very important, since all the future patterns are computed by slightly adapting this particular image. Moreover, this lattice is a very important benchmark test in the later computation of the strain, because, seeing as it is made out a a perfect pattern, no strain is to be expected at its level

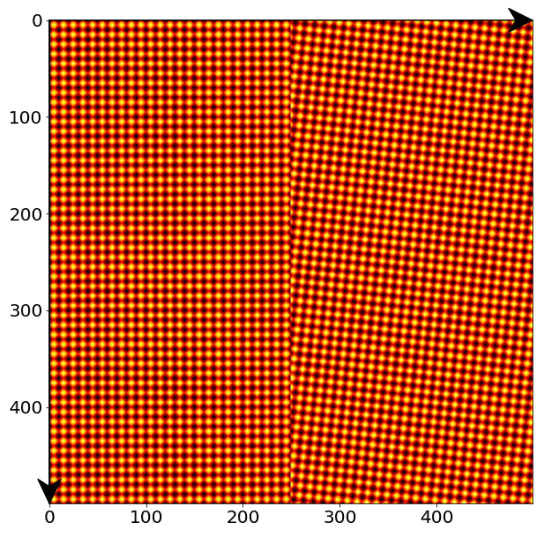

Figure 3.2: Lattice composed out of two layers which meet at the boundary defined at the center of the large lattice. The right layer is rotated with $\theta = 6°$ with respect tot the left layer. This image contains 500 by 500 pixels , corresponding to $r_k = 0.1$. The introduction of this pattern is of high importance to the project, seeing as it is the first and simplest pattern to present strain at the boundary where the two layers meet. The difference between this figure and a real HRTEM is that this one contains half-atoms, fact which is not possible outside the simulation; $r_k = 0.1$

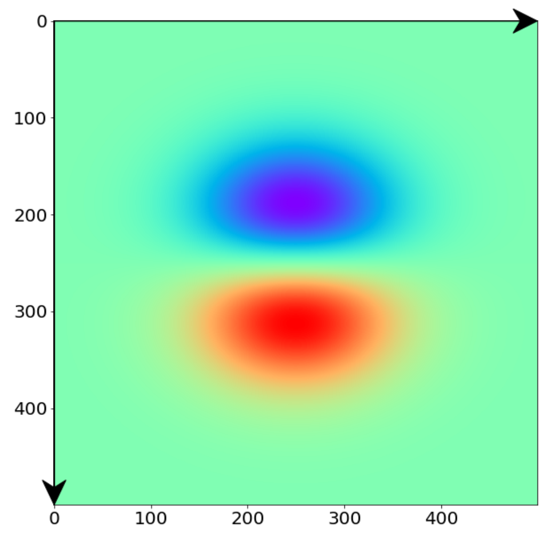

Figure 3.3: The Gaussian shift defined earlier in this chapter. It introduces a shift in the position of the atoms which has the property of smoothening out towards the edges. This shift is applied to the already tilted right image. One does so because they might want to think ahead towards the end goal and keep their reference region on the left, in an area as smooth as possible, in order to be able to properly understanding the mapping. The dimensions are 500 by 500 pixels

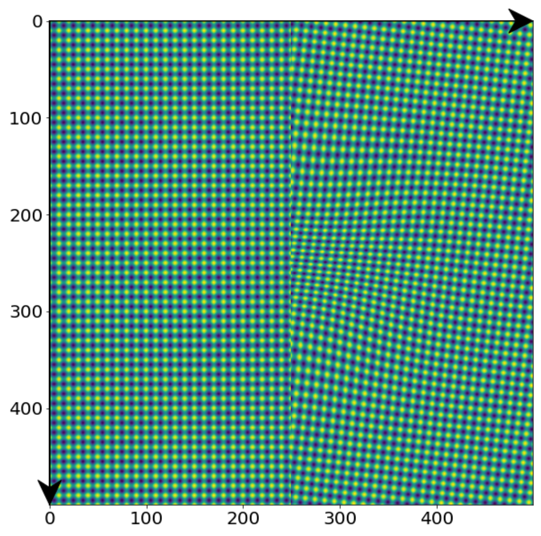

Figure 3.4: A 500 by 500 pixel lattice formed half by a straight lattice and half by a tilted and shifted pattern. The shift is Gaussian and plotted in figure 3.3. This is the reason why, at the center of the lattice, one observes this large deformation in the atom positions. This particular image simulates very well an image with one might obtain of a real deformed crystal by means of the HRTEM. Therefore, defining it is of high interest; In this case r_k is still 0.1

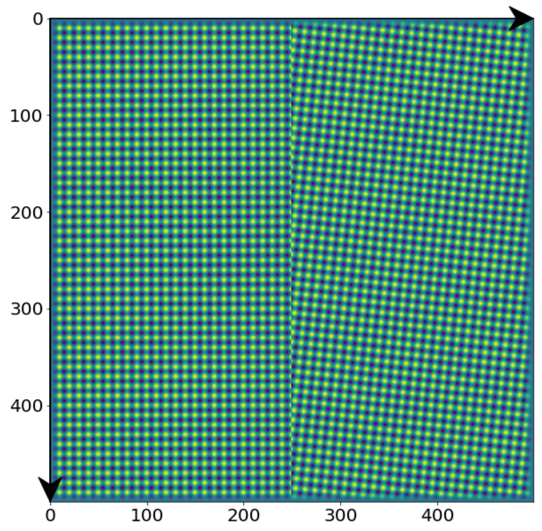

Figure 3.5: Lattice which represents a mathematical concept: it contains 500 by 500 pixels, $r_k = 0.1$ and it is formed out of two layers: the left layer resembles the straight lattice the right layer represents a tilted lattice with an angle of $\theta = 6°$ with respect to the left layer. Moreover, the window function defined earlier in this chapter is applied to it, in order to smoothen the boundaries and be able to get rid of potential discontinuities which occur at the edges and potentially facilitate the computations. This is particularly useful when the lattice is part of a larger crystal structure

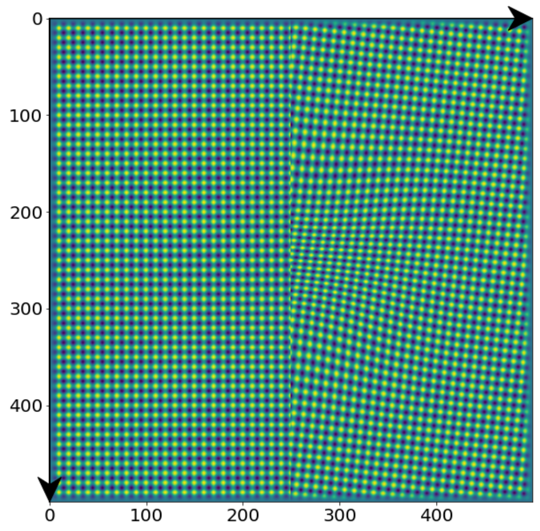

Figure 3.6: Lattice which represents a mathematical concept: it contains 500 by 500 pixels, $r_k = 0.1$ and it is formed out of two layers: the left layer resembles the straight lattice the right layer represents a tilted lattice with Gaussian shift and an angle of $\theta = 6°$ with respect to the left layer. Moreover, the window function defined earlier in this chapter is applied to it, in order to smoothen the boundaries and be able to get rid of potential discontinuities which occur at the edges and potentially facilitate the computations. This is particularly useful when the lattice is part of a larger crystal structure

3.2. Fourier transform of lattice patterns

As has been illustrated in Chapter 2, with the aid of the Fourier transform, one can easily convert between the spacial variables (in this case the real space lattice vectors) and the spacial frequencies, respectively. Following conventions used throughout the report, one can also call the Fourier transform and the inverse Fourier transform the functions which convert between real space and g-space and the other way around, respectively.

Once the patterns are introduced, the next step of the GPA method that leads to computing the desired strain in between the layers of atoms is performing the Fourier transform on the lattices.

The Fourier transform is applied using a function implemented in the Python program. As expected from literature (see Chapter 2), in this power spectrum several peaks corresponding to the highest intensities of the spacial frequencies are visible. The Fourier transform images allow the selection of the frequency given by the g-vector $\vec{g_1}$, which is consistent with the peak location which one chooses to mask around. (Here the notation $\vec{g_1}$ has been chosen in order to emphasize the fact that it refers to the first g-vector being selected)

Figures 3.7, 3.8, 3.9 and 3.10 portray the Fourier transforms of the original simple square lattice (figure 3.1), the simple square lattice tilted by $\theta = 6°$, the Fourier transform of figure 3.5 and the Fourier transform of figure 3.6, respectively.

3.3. Peak selection and estimating the g-vectors

The next step is choosing the peak(s) to be masked. As is stated in Chapter 2, in order to be able to later obtain the strain of a two-dimensional image, one also needs a second frequency vector, $\vec{g_2}$, linearly independent from g_1 to form a basis. For each of the 4 Fourier transforms which have been performed, 2 non-parallel and non-collinear vectors are chosen. In order to convert between the length of a certain vector and the location of the peak in pixel coordinates the following formula is used, which can be applied to either of the components of the vectors:

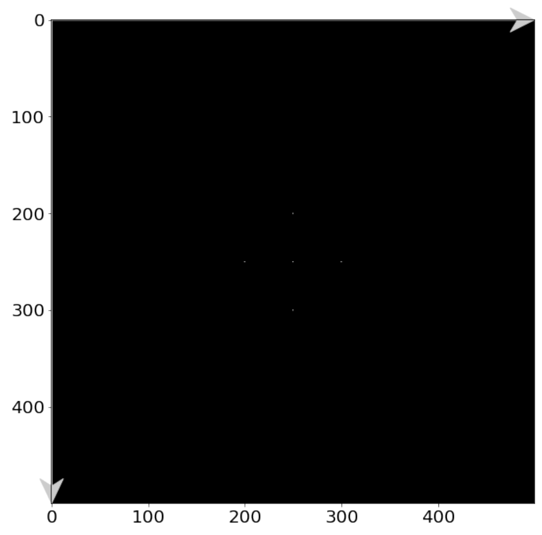

Figure 3.7: Fourier transform applied to figure 3.1, depicting a 500 by 500 pixels image defined in the g-space. This Fourier transform depicts the presence of 4 bright peaks, which correspond to high intensities values of the spacial frequency. Out of this transformation 2 linearly independent vectors must be chosen in order to mask around. This image is very important to the whole project, as it lies at the basis of the GPA method

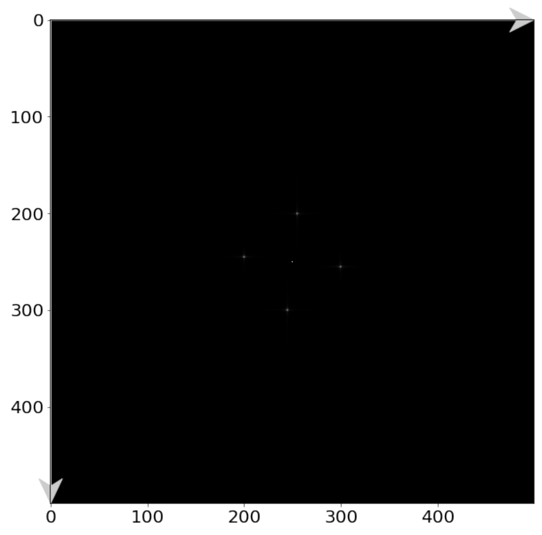

Figure 3.8: Fourier transform applied to the tilted version of figure 3.1, depicting a 500 by 500 pixels image defined in the g-space. This Fourier transform depicts the presence of 4 bright peaks, which correspond to high intensities values of the spacial frequency. Out of this transformation 2 linearly independent vectors must be chosen in order to mask around. This image is very important to the whole project, as it lies at the basis of the GPA method

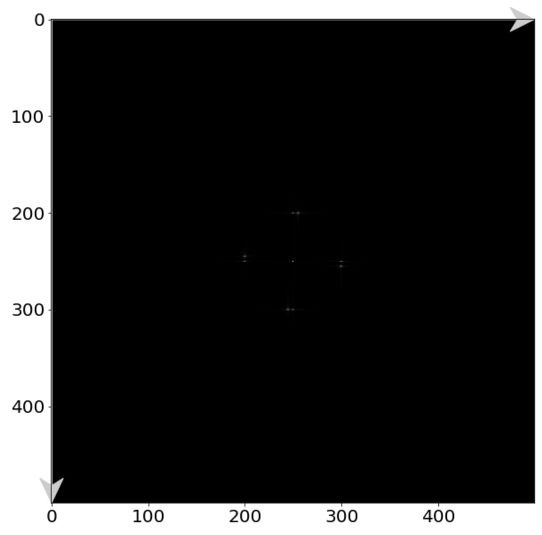

Figure 3.9: Fourier transform applied to figure 3.5, depicting a 500 by 500 pixels image defined in the g-space. This Fourier transform depicts the presence of 8 bright peaks, which correspond to high intensities values of the spacial frequency. Out of this transformation 2 linearly independent vectors must be chosen in order to mask around. This image is very important to the whole project, as it lies at the basis of the GPA method

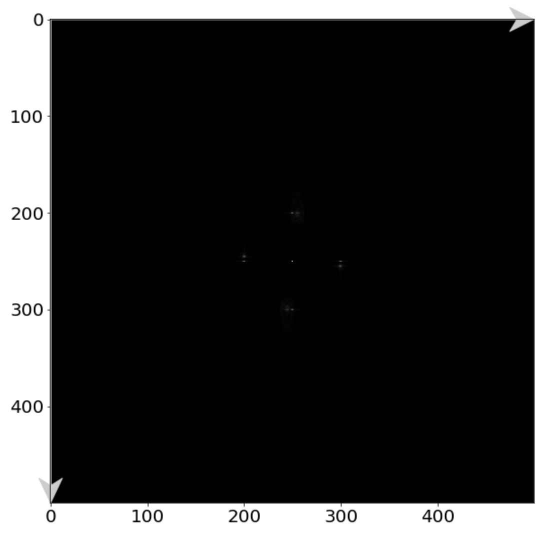

Figure 3.10: Fourier transform applied to figure 3.6, depicting a 500 by 500 pixels image defined in the g-space. This Fourier transform depicts the presence of 8 bright peaks, which correspond to high intensities values of the spacial frequency. Out of this transformation 2 linearly independent vectors must be chosen in order to mask around. This image is very important to the whole project, as it lies at the basis of the GPA method

$$g_{x,y} = \frac{peak_{coordinate_{x,y}} - center_{coordinate_{x,y}}}{500} \tag{3.3}$$

In equation 3.3 the difference in between the coordinates is divided by 500 because that is the length of one side of the square lattice and g_x and g_y are, respectively, the x and y components of generic vector $\vec{g}$, therefore in equation 3.3 the vector sign is left out. Due to the way the axis are defined, the coordinates of the center of the lattice are: (249, 249). The coordinates of the peaks are at first approximated visually. The elegance of the GPA method lies, amongst others,in the fact that it returns the optimal g-vectors which one needs in order to compute the strain. For now, however, it is enough to have the approximation done in equation 3.3.

It is more convenient and computationally correct, however, to do the peak selection through a function[14] performed in the computer algorithm, which has the properties of returning a certain number of peak vectors that the user needs. For instance, since I am looking to compute the strain of a two-dimensional lattice, I need two non-collinear vectors to form a basis, for two regions (the reference region and the region I am looking to compute the strain at), which means four in total. Of course, there are eight Bragg peaks visible, but due to symmetry they do not offer additional information. Therefore, in this particular case, I would run the function in such a way that that it returns four sets of g-vectors. This function, as one expects, facilitates the identification of the peak locations. In table 3.1 one can see characteristics of the four vectors obtained computationally through peak selection. As I have said in the previous lines, I have chosen to return four sets of g-vectors in the Fourier space. Concretely, they are: two for the Fourier image corresponding to the straight lattice ($\theta = 0°$) and two for the Fourier image corresponding to the tilted lattice ($\theta = 6°$) because the other four sets are, respectively, collinear to the first four and do not offer any additional information. Moreover, due to the perfect lattice pattern which is created, the values of the later four sets can be established in a symmetric way. Suppose these vectors were collinear, then they would fail to make a basis which is later needed for the computation of the strain.

In order to distinguish between the naming of the peaks which I have talked about until now in this chapter and in Chapter 2, from this point on the naming of the 4 peaks will be as follows:

- $g_{1selected}$ will be the g-vector corresponding to location 1 in figure 3.22 and it depicts the peak extracted with the use of the Python function, corresponding to the straight lattice (figure 3.1). This will be the first peak corresponding to the straight lattice. The vector arrow is left out in this case in order not to make the parameter less uncluttered, thus it is important to note that this parameter, as well as the three other ones which are yet to be defined, are vectors.
- $g_{2selected}$ will be the length of the g-vector corresponding to location 2 in figure 3.22 and it will depict the peak extracted with the use of the Python function, corresponding to the straight lattice (figure 3.1). This will be the second peak corresponding to the straight lattice.
- $g_{3selected}$ will be the length of the g-vector corresponding to location 3 in figure 3.22 and it will depict the peak extracted with the use of the Python function, corresponding to the tilted lattice (figure 3.2). This will be the first peak corresponding to the tilted lattice.
- $g_{4selected}$ will be the length of the g-vector corresponding to location 4 in figure 3.22 and it will depict the peak extracted with the use of the Python function, corresponding to the tilted lattice (figure 3.2). This will be the second peak corresponding to the tilted lattice.

In table 3.1 is presented an additional overview of the respective peaks, their naming, their location, and the corresponding g-vector lengths.

3.4. Masking the Fourier transform

Once the peaks are determined and clearly listed, one can proceed to the next step in the GPA method which is filtering (also called masking) the Fourier transform previously obtained.

Table 3.1: Table depicting the naming of the 4 selected g-vectors, their x and y coordinates and their respective coordinates

Vector in question	$g_{1selected}$	$g_{2selected}$	$g_{3selected}$	$g_{4selected}$
x-component	0	-0.1	-0.1	0.01
y-component	0.1	0	0.01	0.1
x-coordinate	249	199	199	254
y-coordinate	299	249	254	299

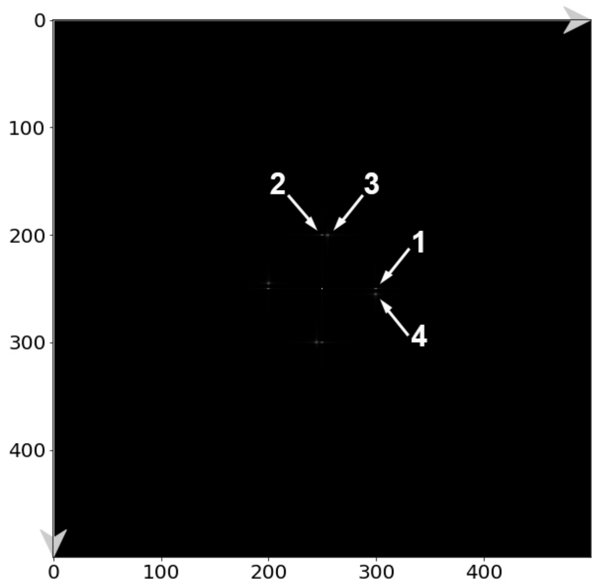

Figure 3.11: Fourier transform applied to figure 3.5, depicting a 500 by 500 pixels image defined in the g-space. This Fourier transform depicts the presence of 8 bright peaks, which correspond to high intensities values of the spacial frequency. Out of this transformation 2 linearly independent vectors must be chosen in order to mask around. This image is very important to the whole project, as it lies at the basis of the GPA method. This particular image is representative because it indicates, by white arrows corresponding to numbers 1 through 4, where each of the 4 selected g-vectors lies.

The mask which I have chosen in the context of this project is a Hanning filter and is centered around the selected peak. This function has been chosen due to its smoothing properties. The Hanning filter, also called the "Cosine Bell" has the scope of smoothing discontinuities at the beginning and end of sampled signals [6].
The Hanning filter is applied, firstly, around vector $g_{1selected}$ and has a radius of ten pixels, in order to also include vector $g_{4selected}$, the peak corresponding to the tilted lattice. Secondly, the Hanning filter is applied around vector $g_{2selected}$, also with a radius of 10 pixels, in order to include vector $g_{3selected}$, the peak corresponding to the tilted lattice.

3.5. Raw phase

. The next step in the GPA method is performing the inverse Fourier transform on the filtered image computed in the previous section with means of the Hanning filter and then plotting its raw geometrical phase, $P_M(\vec{r})$. The M is used as an index in order to emphasize the fact that the raw phase is obtained after having masked the Fourier transform. These operations are done in the Python program using specific functions. The function which computes the phase has the property of reducing it to the interval $[-\pi, \pi]$ (due to the lack of information which can be extracted from the periodicity of the phase and the inconvenience of working with high numbers). The raw phase images which are obtained by masking vectors $\vec{g}_1$ and $\vec{g}_2$ are portrayed in figures 3.12 and 3.13.

For figure 3.6 the raw phase images for vectors $\vec{g}_1$ and $\vec{g}_2$ are, respectively: images 3.14 and 3.15.

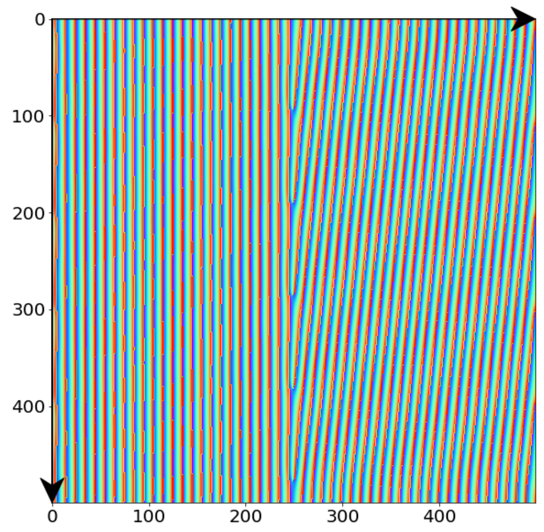

Figure 3.12: Raw phase image obtained by taking the phase of the inverse Fourier transform of the Fourier transform of figure 3.5 which was masked around $g_{1select}$

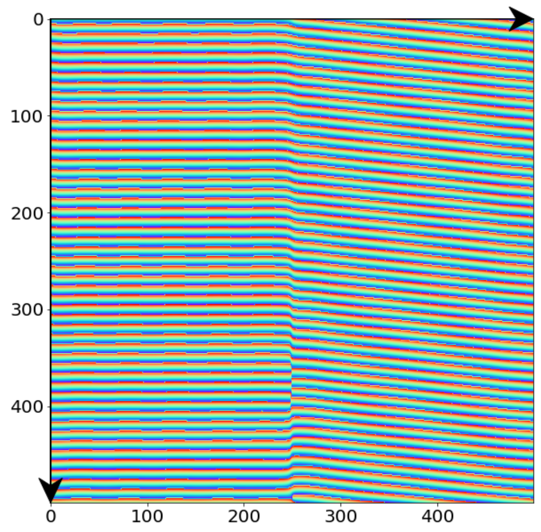

Figure 3.13: Raw phase image obtained by taking the phase of the inverse Fourier transform of the Fourier transform of figure 3.5 which was masked around $g_{2select}$

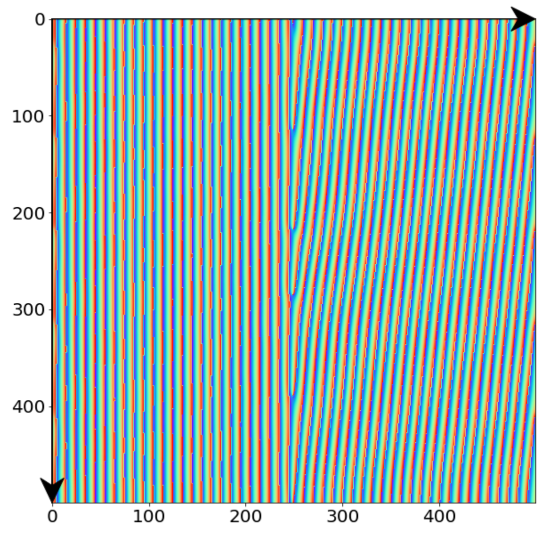

Figure 3.14: Raw phase image obtained by taking the phase of the inverse Fourier transform of the Fourier transform of figure 3.6 which was masked around $g_{1select}$

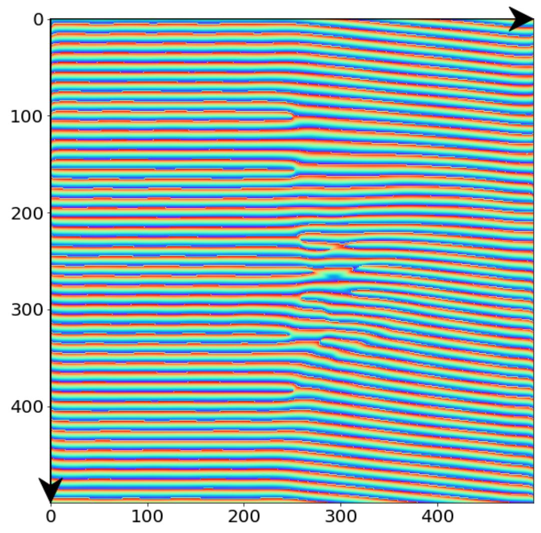

Figure 3.15: Raw phase image obtained by taking the phase of the inverse Fourier transform of the Fourier transform of figure 3.6 which was masked around $g_{1select}$

3.6. Reduced phase calculation and reference selection

Once the raw phase is computed, one starts to anticipate and think more towards the end goal: obtaining the strain. It is crucial to select a reference region in order to be able to compute the strain with respect to it. Ideally, the reference region must be chosen as a smooth surface and having close to no discontinuities. In order to achieve this, the reduced phase is plotted and the reference area is selected according to the obtained figure and to where the smooth areas are located.

By definition, the reduced phase $P_M^r(\vec{r})$ is equal to:

$$P_M^r(\vec{r}) = P_M(\vec{r}) - 2\pi\vec{g}\cdot\vec{r} \tag{3.4}$$

In equation 3.4 the r in the superscript of P_M^r refers to the fact that the phase is reduced. The vector $\vec{r}$ refers to the coordinate of the pixel of the image at which one is computing the reduced phase. $\vec{g}$ is, as always, the vector corresponding to the spacial frequency in Fourier space. In this particular case of equation 3.4 it is the vector corresponding to the peak which was used to mask around, or the second one close to it, which is not exactly the center, but part of the mask due to the radius of 10 pixels.

For instance, if one chooses to compute the reduced phase image corresponding to the mask made around $g_{1selected}$ (figure 3.12), then either $g_{1selected}$ itself must be used in equation 3.4, or $g_{4selected}$. If $g_{1selected}$ is chosen, then one expects for the smooth region to be found on the left side. This is intuitive, seeing as the left side corresponds to the simple straight lattice, just as $g_{1selected}$ does. On the other hand, if $g_{4selected}$ is used to compute the reduced phase in equation 3.4, then one expects for the smooth region to be found on the right side, since the right side corresponds to the tilted lattice, just as $g_{4selected}$ does. Checking whether this holds is a very important step in the process, seeing as this is the optimal verification to see the following:

- whether the right non-colinear peaks have been selected to form the reduced phase image
- whether one has correctly used the correlations amongst the peaks, their coordinates and their lengths
- whether all the steps performed so far (constructing the lattices, performing the Fourier transforms, masking the Fourier transforms and taking the) work appropriately

Getting back to equation 3.4, the reduced phase is calculated by subtracting from the previously determined raw phase a term given by $2\pi\vec{g}\cdot\vec{r}$. As said before, $\vec{r}$ depicts the location of one specific pixel at which the previously mentioned equation is defined. In order to obtain the sought for image pattern one must apply the equation for every pixel, using an iterative loop (performed in Python in the context of this project).

The main challenge in doing so lies in transforming the vectorial product $\vec{g}\cdot\vec{r}$ into a phase image with values which are also found in the interval $[-\pi, \pi]$, just as the raw phase.

Therefore, the following modification is applied to equation 3.4:

$$P_M^r(\vec{r}) = P_M(\vec{r}) - ((2\pi\vec{g}\cdot\vec{r} + \pi)mod(2\pi) - \pi) \tag{3.5}$$

Equation 3.5 makes sure of the fact that the values for the reduced phase image remain in the interval $[-\pi, \pi]$.

To conclude the clarification regarding how to obtain the reduced phase image, one can see in figure 3.16 the reduced phase image corresponding to vector $g_{1selected}$ and in figure 3.17 the reduced phase image corresponding to vector $g_{2selected}$. In order to perform the checks mentioned above regarding the choice of g-vectors the reduced phase images obtained by using vectors $g_{4selected}$ and $g_{3selected}$ are also portrayed in images 3.19 and 3.18, respectively. One can now see that the g-vectors in table 3.1 have been defined correctly and that the steps performed so far are fitting. In figure 3.19 one observe in the right half that there is a completely smooth pattern.This is normal, since the masking has been done around $g_{1selected}$, not around $g_{4selected}$. Therefore, subtracting the term containing

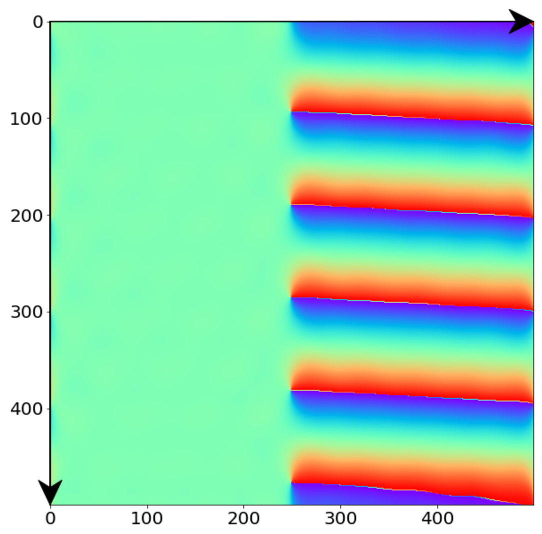

Figure 3.16: Reduced phase image corresponding to figure 3.12 and using the subtract term corresponding to vector $\vec{g_{1select}}$

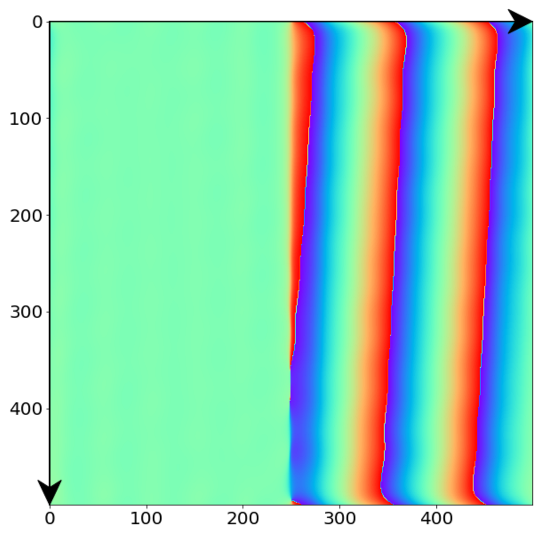

Figure 3.17: Reduced phase image corresponding to figure 3.12 and using the subtract term corresponding to vector $\vec{g_{2select}}$

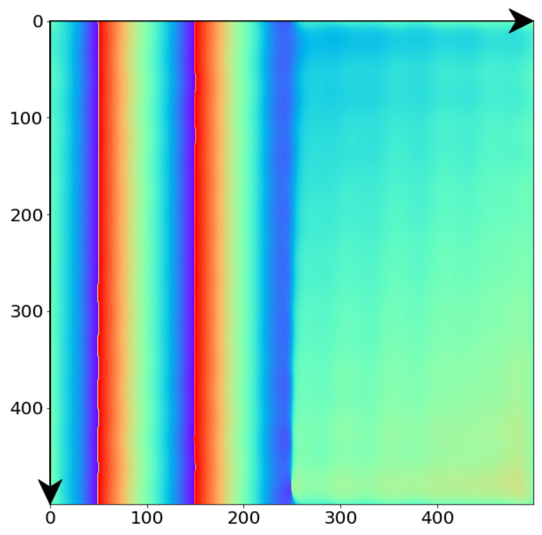

Figure 3.18: Reduced phase image corresponding to figure 3.12 and using the subtract term corresponding to vector $\vec{g_{3select}}$

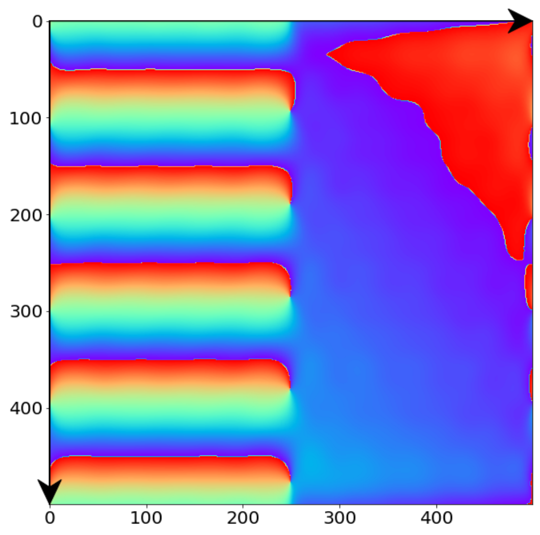

Figure 3.19: Reduced phase image corresponding to figure 3.12 and using the subtract term corresponding to vector $\vec{g_{4select}}$

$g_{4selected}$ leads to the possible presence of a few extra wave vectors. This is exactly the case in figure 3.19. This possibility depends on the distance between the peaks which are masked together, the amount of pixels over which they span and the sharpness of the peaks. This also explains why in figure 3.18 one observes a homogeneous pattern on the entire right side: perhaps the peak corresponding to $g_{3selected}$ is defined better, more towards a high sharpness region, or perhaps this peak is spanned over fewer pixels. One must also keep in mind that the establishing the peak positions has still been done through visualisation, the use of a Python function which returns a number that only roughly indicates the length of the g-vector, or perhaps by trial and error using the checks of the reduced phase image, depending on the reader's choice.

Now that the reduced phase images have been established, one can define the reference region. It is important, as stated previously, to select a smooth region, away from the edges of the lattice in order to avoid discontinuities. As long as these conditions are met, the choice of the reference region

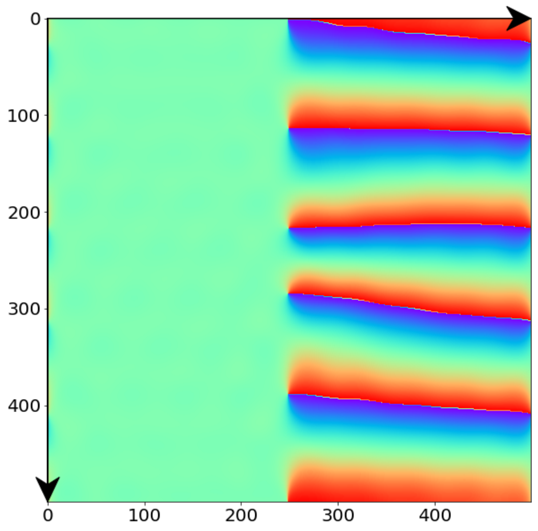

Figure 3.20: Reduced phase image corresponding to figure 3.14 and using the subtract term corresponding to vector $\vec{g_{1select}}$

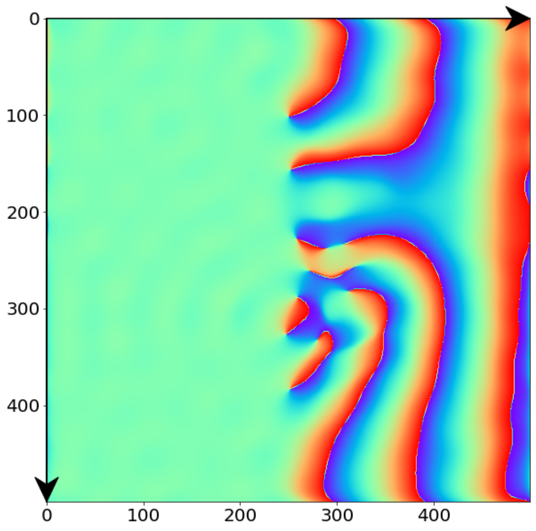

Figure 3.21: Reduced phase image corresponding to figure 3.14 and using the subtract term corresponding to vector $\vec{g_{2select}}$.

is completely arbitrary and left to the reader. I have chosen the rectangle with the following points, with coordinates: (50, 149),(50, 149) and it can be visualised in image 3.22. Then reason to do so is because the left region is homogeneous, perfect and straight, without any shifts. Therefore, calculating the strain with respect to it is more intuitive and easy to visualise. From now on this region will be called $\mathcal{O}$.

3.7. Accurate computation of g-vector

As has been stated before, the elegance of the GPA method lies, amongst others, in the fact that it gives as output an accurate and optimal value of the g-vector. This section is dedicated towards showing the mathematical steps which lead to obtaining this vector.

The main idea is that one must find the value for $\vec{g}$ for which the following function is minimised:

$$f(\vec{g}, c) = \sum_{\vec{r} \in \mathcal{O}} ||P_M(\vec{r}) - 2\pi(\vec{g} \cdot \vec{r} - c)||^2 \tag{3.6}$$

In equation 3.6 the summation over all pixels in the previously chosen region $\mathcal{O}$ is performed over the absolute values of the term $P_M(\vec{r}) - 2\pi(\vec{g} \cdot \vec{r} - c)$ and then taken to the power of 2. The corresponding values of the components of the vector $\vec{g}$ (the x and y components, respectively) and the constant c for which the function in equation 3.6 has a minimal value are the sought for answers. Moreover, all of the terms in equation 3.6 are real, which means that the absolute value can be left out in all the future computations.

Solving the previously stated minimisation problem can be performed in two ways: ”by hand”, with a process which involves partial differential equations, or through the computer algorithm which uses built-in functions. The results presented in this project are obtained by using the later method, but for mathematical completeness I am also presenting in this section the steps which one would perform ”by hand” to solve the minimisation problem. At the start of the project I have attempted using this later method. However, this gave an error regarding the output of the optimal g-vectors.

Assuming that a solution exists, it means that by taking the partial derivatives of 3.6 with respect to the x component of $\vec{g}$ (called g_x), the y component of $\vec{g}$ (called g_y) and the constant c are all equal to 0. In other words:

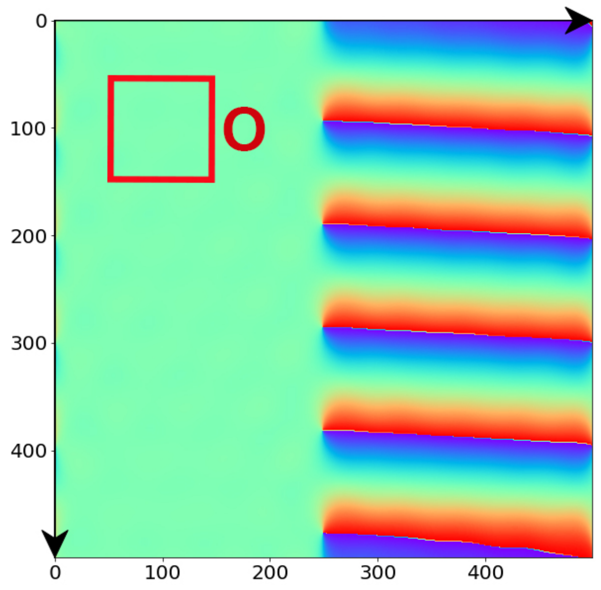

Figure 3.22: Here figure 3.16 is represented again, depicting the reduced phase image corresponding to figure 3.12 and using the subtract term corresponding to vector $\vec{g_{1select}}$. This figure is used in order to properly portray the reference region labeled O by a square with red border. The coordinates of the square are (50, 149) in both directions: alongside the x axis, as well as alongside the y axis. This is part of one of the steps of the GPA method

$$\frac{\partial f(\vec{g},c)}{\partial g_x} = 0, \tag{3.7}$$

$$\frac{\partial f(\vec{g},c)}{\partial g_y} = 0, \tag{3.8}$$

$$\frac{\partial f(\vec{g},c)}{\partial c} = 0 \tag{3.9}$$

Solving the left members of equations 3.7, 3.8, 3.9, using the compact vectorial writing, eliminating the multiplication constants that appear after the differentiation (which have no influence seeing as the right hand side is 0), one gets the following equations for c and $\vec{r}$, respectively:

$$\sum_{\vec{r}} P_M(\vec{r}) - 2\pi(\vec{g}\cdot\vec{r} - c) = 0 \tag{3.10}$$

$$\sum_{\vec{r}} (P_M(\vec{r}) - 2\pi(\vec{g}\cdot\vec{r} - c))\vec{r} = \vec{0} \tag{3.11}$$

Splitting equation 3.11 into the x and y components and rearranging the systems of equations into a matrix equation one gets:

$$\sum_{(x,y)} \begin{pmatrix} 2\pi & -2\pi x & -2\pi y \\ 2\pi x & -2\pi x^2 & -2\pi xy \\ -2\pi y & -2\pi xy & -2\pi y^2 \end{pmatrix} \cdot \begin{pmatrix} c \\ g_x \\ g_y \end{pmatrix} = \sum_{(x,y)} \begin{pmatrix} -P_M(\vec{r}) \\ -P_M(\vec{r})\cdot x \\ -P_M(\vec{r})\cdot y \end{pmatrix} \tag{3.12}$$

Equation 3.12 can then be solved using a computer algorithm. The raw phase P_M is known at each point (x, y) (shown in figure 3.12), each point with coordinates (x, y) is known, therefore, with the above three components of the equation one can easily compute the corresponding values for c, g_x and g_y.

All these operations have thus been performed on a chosen region of reference O. As stated before, equation 3.12 was ultimately not used in the computational method, due to the cumbersome layout of formulas in the code, which made it difficult for the error to be found, but also due to the fact that the code did not give the right g-vector as output.

Next, it must be specified that for any pixel $\vec{r}$ of the lattice (in real space), there is always the possibility of defining a region R centered around it. It is important, however, for this region R to contain at least three atoms. If not, the local lattice vectors are undefined. In order to obtain the strain, the g-vectors must be computed in each pixel of the lattice. The computer algorithm, therefore, making use of a recursive method, computes the optimal value of $\vec{g}$ in each region R. The code lets the user vary this the dimensions R. However, seeing as the line of code responsible with computing the minimum is quite demanding with respect to the running time, I have chosen it 20 by 20 pixels. In such a region there can be at most 4 atoms.

Seeing as this method did not work for me, I then tried another approach: creating a function to be minimised [16]. This function was the right member of equation 3.5, taken to the power of 2. It was clear, however, that after performing a benchmark test the output of the g-vectors did not correspond with what one expected. Moreover, since this operation must be performed in order to compute the g-vectors corresponding to each pixel of the matrix, one quickly realises that the running time of the algorithm is of the order of a few hours, which can be cumbersome and inconvenient. I believe that the implemented minimisation function in the Python algorithm gives the wrong results when applied to this particular saw-tooth function,

A third method to try and compute the minimum g-vectors was then implemented: using the built-in function in Pyhton of performing the curve-fit [17]. The output, in this case, should then have been the optimal array formed out of optimal values for g_x, g_y and c, respectively. Unfortunately, this method does not return the expected results either, probably also due to the discontinuities present in the function which must be fitted. In this case I chose to fit the function given by $(2\pi\vec{g}\cdot\vec{r}+\pi)mod(2\pi)-\pi$, onto the raw phase values for each pixel given by $\vec{r}$.

It is important to mention that all the methods presented above compiled in Python and what I believe that one needs to do at this point is to do some deep research regarding the minimisation of this type of discontinuous function which this project demands to be used. Unfortunately, due to the strict timeline of the project, at the moment of the discussion of the thesis I have not yet had time to do. However, it is also particular that the Python built-in functions are not able to return the parameters corresponding to the minimum value of this particular function. At a certain point, the pure mathematical operations and the built-in numerical methods in Python stop agreeing and that is where the error occurs. Why this exactly happens, is, unfortunately, still a mystery.

Assuming that one has succeeded in obtaining the sought-for optimal g-vectors, I am going to introduce the following notations: $G(R)$, which is a 2 by 2 matrix, defined in a certain pixel $\vec{r}$ which in turn is the center of a certain region R. $G(O)$ is also a 2 y 2 matrix defined in a certain pixel $\vec{r_O}$ which indicates the center of the region O. By definition, the previously defined matrices are equal to:

$$G(R) = \begin{pmatrix} g_{x_1}(\vec{r}) & g_{x_2}(\vec{r}) \\ g_{y_1}(\vec{r}) & g_{y_2}(\vec{r}) \end{pmatrix} \tag{3.13}$$

In equation 3.13 $\vec{r}$ is the center of the region R. g_{x_1} and g_{y1} are, respectively, the x and y components of the g-vector corresponding to

An analogue computation is performed for the case of P_{M_2} (shown in figure 3.12). One now has found optimal values for the following variables: c_1, g_{x_1}, g_{y_1}, c_2, g_{x_2} and g_{y_2}. In the computer program all these values are stored, for each pixel (x,y) in a dictionary and can easily be accessed. From this dictionary all the matrices and vectors defined and used in section 3.8 can be computed. A disclaimer, however must be added: due to the fact that at the boundaries of the lattices abrupt discontinuities occur, one must stay away from them. This is done by leaving all the values corresponding to any of the boundary pixels equal to zero (such as $\vec{g}$, $\vec{a}$, c and so on), 15 pixels away from each side.

3.8. Strain computation

Once the optimal values for c_1, g_{x_1}, g_{y_1}, c_2, g_{x_2} and g_{y_2} are found, one has all the necessary parameters and variables in order to compute the desired strain. For this, however, I need to introduce a few more notions before proceeding.

From literature (see Chapter 2), to each vector in the reciprocal space of the form $\vec{g}$ corresponds a direct lattice vector of the form $\vec{a}$. In order to give a better overview of the computations, seeing as one is working with two non-parallel vectors $\vec{g}$, each having two components, I will introduce once again matrix notation. Therefore, at a particular pixel given by the vector $\vec{r}$ with components (x,y), one has the following relation:

$$A(\vec{r}) = \begin{pmatrix} a_{x_1}(\vec{r}) & a_{x_2}(\vec{r}) \\ a_{y_1}(\vec{r}) & a_{y_2}(\vec{r}) \end{pmatrix} = (G(\vec{r})^{-1})^T = \left(\begin{pmatrix} g_{x_1}(\vec{r}) & g_{x_2}(\vec{r}) \\ g_{y_1}(\vec{r}) & g_{y_2}(\vec{r}) \end{pmatrix}^{-1} \right)^T \tag{3.14}$$

Equation 3.14 can be used only if $G(\vec{r})$ is a non-singular matrix, in other words if $\det(G(\vec{r})) \neq 0$. In case this doesn't occur, for the regions where $G(\vec{r})$ is a singular matrix I set the value of $A(\vec{r}) = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$. Furthermore, the symbols T and -1 in the exponent in equation 3.14 refer to the transpose and inverse operations, respectively.

In particular, for any pixel in a certain region R one can write the following relation between the positions of the atoms and the local lattice vectors (in real space):

$$\vec{r}_{\alpha\beta}(R) = (c_1(R) + \alpha) \cdot \vec{a_1}(R) + (c_2(R) + \beta) \cdot \vec{a_2}(R) \tag{3.15}$$

In equation 3.15 α and β are integers which describe one particular lattice atom found at any position $\vec{r}$ in region R. Rewriting this equation into matrix form one gets:

$$\begin{pmatrix} x(R) \\ y(R) \end{pmatrix} = \begin{pmatrix} a_{x_1}(R) & a_{x_2}(R) \\ a_{y_1}(R) & a_{y_2}(R) \end{pmatrix} \begin{pmatrix} c_1(R) + \alpha \\ c_2(R) + \beta \end{pmatrix} \tag{3.16}$$

I define now A_O as being the A matrix defined in the center of region O. The coordinates of this particular pixel are named: x_O and y_O. In the case where the center is float, the computer program selects the rounded number and transforms it in an integer, seeing as the pixels $\vec{r}$ are only defined as integers in the interval $[0, 499]$. In addition, I also define the constants $c_1(O)$ and $c_1(O)$ as being the c_1 and c_2 values in point (x_O, y_O), respectively.

A similar equation to equation 3.16 can be thus written for the coordinate (x_O, y_O):

$$\begin{pmatrix} x(O) \\ y(O) \end{pmatrix} = \begin{pmatrix} a_{x_1}(O) & a_{x_2}(O) \\ a_{y_1}(O) & a_{y_2}(O) \end{pmatrix} \begin{pmatrix} c_1(O) + \alpha \\ c_2(O) + \beta \end{pmatrix} \tag{3.17}$$

Equations 3.16 3.17 are needed in order to be able to establish a relation between $(x(R), y(R))$ and $(x(R), y(R))$. Eliminating α and β one gets the desired relation:

$$\begin{pmatrix} x(R) \\ y(R) \end{pmatrix} = A(R)A(O)^{-1} \begin{pmatrix} x(O) \\ y(O) \end{pmatrix} + A(R) \begin{pmatrix} c_1(R) - c_1(O) \\ c_2(R) - c_2(O) \end{pmatrix} \tag{3.18}$$

Using a more compact notation, $A = A(R)$,$(x_O, y_0) = \vec{r_O}$, $A(O) = A_O$, $(c_1(R), c_2(R)) = \vec{c}$ and $(c_1(O), c_2(O)) = \vec{c_O}$ one eventually obtains:

$$\vec{r} = A{A_O}^{-1}\vec{r_O} + A(\vec{c} - \vec{c_O}) \tag{3.19}$$

The difference between $\vec{r}$ and $\vec{r_O}$ is the displacement in the GPA. In other words, $\vec{r} - \vec{r_O} = \vec{u}$. Assuming that this displacement is infinitesimal, the local distortion matrix can now be introduced as having the following components: $\vec{e}(R)_{ij} = \frac{\partial u_i}{\partial x_j}$. Computing in detail the right hand side of the previous equation, one obtains the following equation:

$$\vec{e}(R) = I - A_O A^{-1} \tag{3.20}$$

It is important to note that since each component of the displacement vector is being differentiated with respect to two variables, the strain matrix, as well as the rotation matrix will each have 2x2 elements. In case of an infinitesimal distortion, the strain (ϵ) is defined as the symmetrical part of the distortion matrix. The rotational matrix (R_M), on the other hand, is equal to the anti-symmetric part. In other words:

$$\epsilon = \frac{1}{2}(e(R) + e(R)^T) = \begin{pmatrix} \epsilon_{xx}(R) & \epsilon_{xy}(R) \\ \epsilon_{yx}(R) & \epsilon_{yy}(R) \end{pmatrix} \tag{3.21}$$

$$R_M = I + \frac{1}{2}(e(R) - e(R)^T) = \begin{pmatrix} R_{xx}(R) & R_{xy}(R) \\ R_{yx}(R) & R_{yy}(R) \end{pmatrix} \tag{3.22}$$

The strain image and rotation have been plotted, for the sake of testing and discussing the desired outcome with the already implemented computer algorithm for GPA [13]. More on this can be seen in Chapter 4.

4 Results and Discussion

This chapter is dedicated to presenting and discussing the obtained results through the computational method performed in Chapter 3. I start by examining some of the images in the previous chapter, then I introduce new results regarding the strain computation which have not yet been made visible. I then proceed by discussing the strain results one would obtain by using the software ”Strain++ measure strain in TEM images”, an algorithm implemented in C++ by J. J. P. Peters and detailed in [13], and argue why these are the images one should expect. The last part of this chapter is dedicated to the presentation of part of my own results which does not lead to the desired effect and why this might be. Moreover, advice for how to improve and bring future addition to the Python code is presented throughout this chapter.

4.1. Discussion of results for sections 3.1-3.6

Up to and including the computation of the reduced phase pattern (section 3.6) one obtains patterns which completely coincide with what the literature suggests [18].

With other words, the creation of the desired lattice patterns, as one can see in section 3.1, is performed as wanted. This can be checked by several means such as:

- counting the atoms by hand (as the resolution of the images in this section is quite high): however, this can be quite cumbersome and inefficient
- changing lattice parameters: for instance, one could set $r_k = 0.05$, which is half of the value used in creating all the latices in section 3.1. Seeing as $\vec{r_k}$ is defined in reciprocal space, one expects for the new lattices to contain atoms that are twice as large. This can indeed be checked in figure 4.1 and its Fourier transform, figure 4.2. The Bragg peaks, which one would expect, are also located at half the distance with respect to the center (249,249) of the lattice of what they were before, with the use of the initial value of r_k.
- checking the rotation of the tilted lattice: in the Python program, the tilt has been performed with an angle of $\theta = 6^{\circ} \approx 1$ rad. This can be visualised in the lattices which contain a tilted layer, but a better check is to verify the concrete values of the rotation matrix defined in equation 3.22. This check is broadly discussed in the next sections and I can already state that it agrees with what one would expect.

The initial selection of the four Bragg peaks (see table 3.1) is also done correctly. First of all, the GPA method states that it is not mandatory, especially in the early stages, to select an optimal peak location. Moreover, the reduced phase image must contain a smooth region on the side of the lattice corresponding to a particular g-vector. Seeing as one is still working in these sections with no additional shift in the positions of the atoms, just with the presence of tilt of the whole lattice, the expected homogeneous region is (almost) one of the sides of the lattice (left or right) itself. Some inhomogeneity (figure 3.19) can still be expected (in the form of an addition waves), but this has already been verified and explained in the previous chapter.

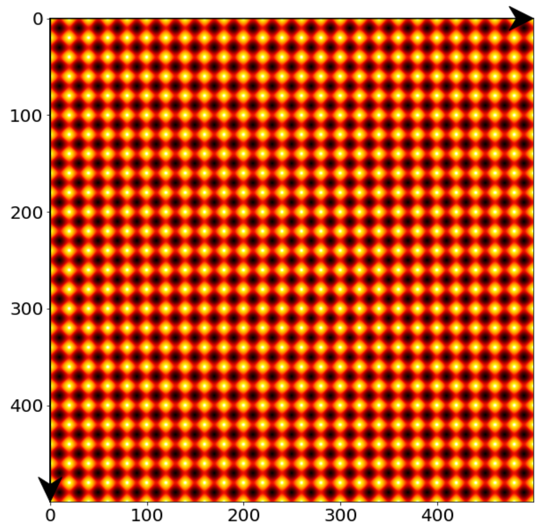

Figure 4.1: Original square lattice, defined with a size of 500 by 500 pixels, corresponding to $r_k = 0.05$. One can see the x axis pointing downwards and the y axis pointing to the right. This lattice presents no noise, no shift and no tilt. The number of atoms in this image is 25x25. This lattice serves to check whether the defined lattices in Chapter 3 are correctly computed and plotted. One would expect that by making the k-vector in the reciprocal space twice as small, the dimensions of the atoms in the real space image become twice as large. This is indeed the case in this figure, which confirms the correctness of the implementation of the lattices in the code

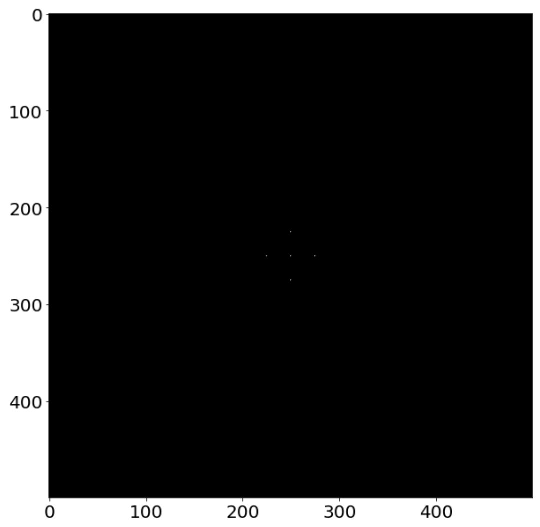

Figure 4.2: Fourier transform of figure 4.1. This image is defined in the Fourier domain and contains 500 by 500 pixels. It is used to verify whether the Fourier transform introduced in Chapter 3 has as output the correct expected values. One can see the expected 4 Bragg spots, which, however, now have a different coordinate- than before when $r_k = 0.1$: the distance between each peak and the center (249, 29) has halved with respect to the way it was in the case of the Fourier transform of a lattice with $r_k - 0.1$ This is indeed the case in this figure, which confirms the correctness of the implementation of the Fourier transform in the code

The raw phase image also corresponds to what one would expect (see literature [18]). The abrupt jumps in the phase gradient (portrayed by the jump from red to blue) is to be expected since the phase is forced to lie in the interval $[-\pi, \pi]$. With other words, the discontinuities in the raw phase have no correlation at all with possible discontinuities in the original lattice. However, the abrupt change in the orientation of the phase from one side to the other is due to the fact that that is the layer boundary which is, indeed, a discontinuity in the crystal and also the place where one would later on expect to see peaks in the value of the strain.

4.2. Discussion and results for sections 3.7 and 3.8

The step corresponding to the computation of accurate g-vectors is, unfortunately, the place where my code does not return the expected results and which prohibits me from displaying a correct strain image at this moment in time.

As I have explained in Chapter 3, I have so far attempted three main methods of computing the optimal g-vectors corresponding to:

- the ”by hand” approach in which I have performed the minimisation myself and the Python code just needs to solve a system of equations
- using the minimisation function already implemented in Python. Since this function uses a Simplex algorithm, I think that, amongst other reasons, the reason for which the output of the optimal g-vectors is mistaken, is the possible incompatibility of the Simplex algorithm with the function one desires to minimize
- using the curve-fit function, which is also implemented in Python and which, just as the previous two methods, gives a wrong result

Further in this section I will present and discuss some of my outputs by using the curve-fit algorithm. I have chosen to give more details on this particular method because it takes less amount of time to compile than the minimisation function and because it avoids the possibility of human error to hide itself in the process (as is the case with the ”by hand” method). The second two methods, are of course also available in the code.

In order to shine some light on this particular section of the code, I want to break down the lines of which it is composed and explain in better detail the method used to compute the optimal values through means of the curve-fit. For this part of the report especially it can be helpful for the reader to also consult with the appendix while reading (see A).

The function to be fitted is defined as:

$$func(g_x, g_y, c) = (2 \cdot \pi \cdot (g_x \cdot x + g_y \cdot y - c - \pi))mod(2\pi) + \pi \tag{4.1}$$

In equation 4.1 g_x, g_y and c are the respective optimal values which need to be given as output by the curve-fit. Before trying to calculate the matrix $G(R)$ (see equation 3.14) for all values of possible regions R, a test code is performed which I am going to discuss now (the operations are then redone in the context for a recursive loop in order to compute what is done in the test code for one R, for all R's of the lattice).

Since strain is computed with respect to a reference region, first the above mentioned minimisation is done for for the reference region O. The x and the y data given to the fit function are, respectively, the x and y coordinates of pixel in region O. A variable Z is defined as the phase values corresponding to the specific part of the lattice one is looking at (left or right or in another words raw phase image obtained through masking a vector corresponding to the straight half of the lattice or the vector corresponding to the tilted half). This is done because one would like to fit $func$ on Z in every pixel in the desired pre-selected region. In any case, the fitting must be performed twice, in order to obtain two sets of optimal values for $\vec{g}$ and c. The initial guesses needed by the algorithm are the selected peaks in table 3.1. By running the code, one can see that the closer the initial guess is to the optimal value which one expects, the better the result is. Unfortunately, having to depend on this condition is incorrect, seeing as the algorithm should be able to compute the optimal values regardless of the input values. This dependency on needing to insert already optimal values as first guesses in the fitting function is bound to result into wrong results.

The process is then repeated for an arbitrary area R. For the sake of testing the code, I defined it in the right side of the lattice (where one expects the strain to be seeing as the origin is chosen on the left side, which is uniform and smooth throughout).

In order to check the above made statements, I have plotted three figures which depict the following images, defined on a 21 by 21 pixels. The vertices of this region , which I will call $\mathcal{R}$, have the following coordinates: $(x_1, x_2, y_1, y_2) = (270, 290, 270, 290)$

1. figure 4.3 and figure 4.3 depict Z at every point in the defined region,
2. figure 4.4 depicts the value of $func$ in the defined region, by plugging in the optimal values for g_x, g_y, c when the initial guess is $g_{3select}$. The optimal values are supposed to be close to the values of $g_{3select}$
3. figure 4.5 depicts the value of $func$ in the defined region, by plugging in the optimal values for g_x, g_y, c when the initial guess is $g_{1select}$. The optimal values are supposed to be close to the values of $g_{3select}$

As one can see from figures 4.3, 4.4 and 4.5, for the particular region $\mathcal{R}$ which has been used in this test, the obtained values are very accurate. However, as soon as the initial guesses start changing, one notices that the image 4.5 drastically changes for the worse. From figures 4.4 and 4.5 it is clear that the output is exactly the initial guess: $g_{3select}$ and $g_{1select}$, respectively. The error in the code lies in this section corresponding to the curve-fitting: the fact that the optimal values, regardless of the used method, depend on the initial guesses used in the minimisation algorithms.

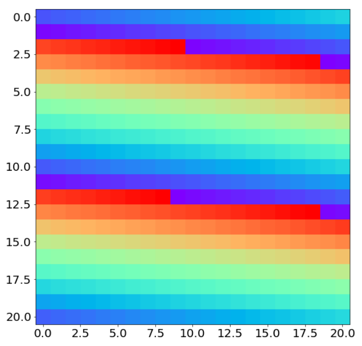
Figure 4.3: Raw phase image plotted for the region $\mathcal{R}$. With other words, this is a 21 by 21 pixels snapshot from a raw phase figure similar to 3.12, but which one would have to obtain through masking the peak $g_{3select}$ instead of $g_{1select}$

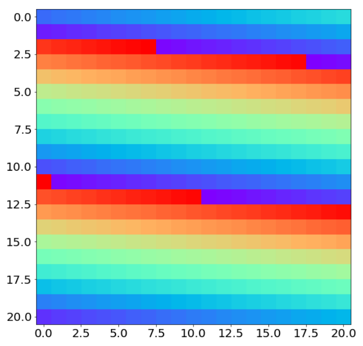
Figure 4.4: Mapping of the values from region $\mathcal{R}$ of function $func$ by plugging in the optimal values for g_x, g_y, c when the initial guess is $g_{3select}$. The optimal values are supposed to be close to the values of $g_{3select}$. This is indeed the case

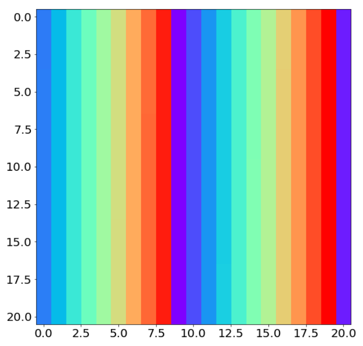
Figure 4.5: Mapping of the values from region $\mathcal{R}$ of function $func$ by plugging in the optimal values for g_x, g_y, c when the initial guess is $g_{1select}$. The optimal values are supposed to be close to the values of $g_{3select}$. Here lies the error of the code

4.3. Expected strain images and discussion

As has been mentioned at the start of this chapter, it is of interest to check and analyse what one should obtain through means of my Python code. I do so by using the algorithm implemented in C++ by J. J. P. Peters detailed in [13]. In this section I will not only explain how to obtain the strain, but also a short description of the steps.

The algorithm requires the upload of a grayscaled image of the lattice. In my Python code one can easily obtain these images through simply changing the coloring of the figures already obtained in section 3.1. The next step is to select the g-vectors and to mask around them. This is done almost identically to what I have explained in section 3.3, with a slight modification: the axes and their orientation (x and y) are defined differently from the way it is done in this project. Therefore, in order to identify the location of the g-vectors with what my project suggests, one must use the respective g-vectors mentioned in table 3.1, but rotated with $90°$ counterclockwise. It is important to mention that this is also the reason why the components of the strain matrix 3.21 differ from the ones I introduced in this current chapter. The xy component is actually the yx component and the other way around. The xx component is actually the yy component and other way around. In this section have therefore re-named the strain images according to this later definition, which the C++ algorithm also implements [13].

Next, I will enumerate some of the lattice patterns introduced in section 3.1 and show the respective plots for strain and rotation. I have chosen the lattice patterns which I believe to be illustrative for a complete discussion regarding the lattice strain. In the first case (of a simple straight lattice) I also inserted the intermediate Fourier transform plot performed with the algorithm implemented in C++. In the other presented patterns I have left out the images corresponding to the Fourier transform, but inserted the rotation matrices and the strain matrices, since the later is the main focus of the project and the rotation matrix is a measure of checking the results. I want to point out the fact that in order to be able to nicely analyse the rotation and strain plots, I have chosen a color gradient which has the role of emphasizing the fact that color light green corresponds with a value of zero.

1. Figure 4.6 is the simple straight lattice: This lattice is the grayscaled version of image 3.1 and represents the benchmark test in order to see whether the results concerning the obtained are correct. Since in this case one has a perfect straight square lattice, it is clear that the strain matrices and the rotation matrices should all be equal to zero. Indeed, this is visible in figures 4.8, 4.9 and 4.10, as the values of all the numbers of which these three matrices are made of are (almost) zero. All the other components of the strain and rotation ($_{yy}$, $_{y}x$) are left out since they all are equal to the zero matrix and introducing them would be redundant.
In reality, there is a very small imperfection, as the color which is indicated is a bit darker/lighter than the reference in some cases, but this can be due to the approximation errors lying in the C++ algorithm and in the fact that the created lattice in Python is not entirely perfect, also due to

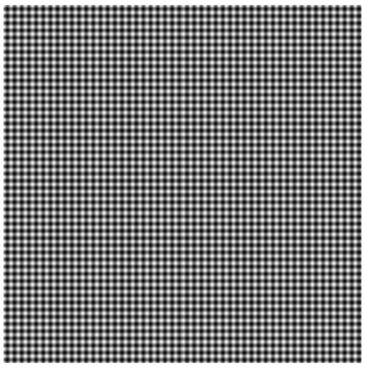
Figure 4.6: Grayscaled version of image 3.1

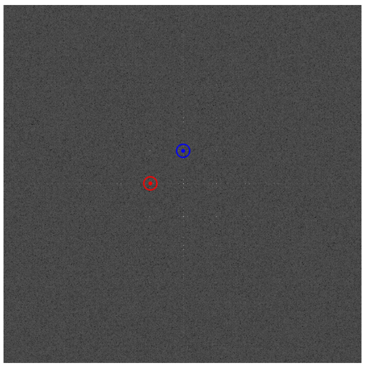
Figure 4.7: Fourier transform of image 3.1

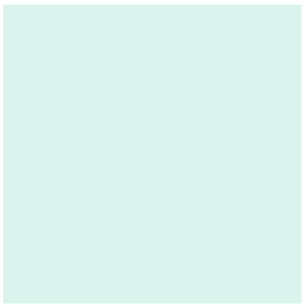
Figure 4.8: Image portraying ϵ_{xx}, the xx component of the strain matrix corresponding to image 4.6 which has only zero values due to the fact that the lattice in question has a perfect crystal structure

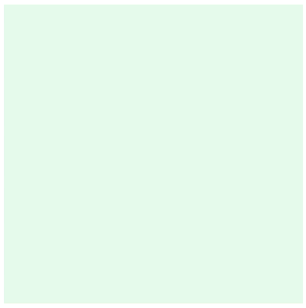
Figure 4.9: Image portraying ϵ_{xy}, the xy component of the strain matrix corresponding to image 4.6 which has only zero values due to the fact that the lattice in question has a perfect crystal structure

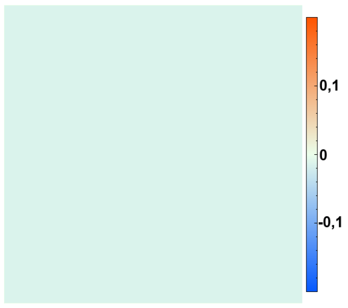

Figure 4.10: Image portraying the parameter w_{xy}, which is the xy component of the rotation matrix corresponding to image 4.6 which has only zero values due to the fact that the lattice in question has a perfect crystal structure

small approximation errors. Moreover, it is important to state the fact that since this lattice is finite, it cannot be completely perfect, by default. Nevertheless, this benchmark test clearly states the fact that the used algorithm gave correct results as output. In figure 4.7 one can see the Fourier transform (together with the selected peaks) of figure 4.6.

2. Figure 4.11 depicts a lattice containing one layer corresponding to a simple straight lattice and one layer corresponding to a tilted lattice: this is the grayscaled image of the pattern shown in 3.2. Figures 4.12-4.14 are the xx, xy and yy components of the corresponding strain, figures 4.15 and 4.16 are the xy and yx components of the corresponding rotation matrix. By looking at these figures, one can notice that they get what they expect: peaks corresponding to extreme values of the strain at the boundary of the two layers. The series of peaks which can be observed in some of the pictures close to the right side of the images might correspond to the fact that the algorithm interprets the inputted lattice as being part of a period structure. Intuitively, seeing as other layers might be present on the boundaries of the lattice, the appearance of peaks at the edges is completely justified and will not be mentioned again, not even in images similar to this one. The rotation value seen in figure 4.21 is approximately equal to 0.1, which the amount of $6°$ in radians. Therefore, one can confirm that the righter half is rotated with $6°$ with respect to the left one, while also displaying strain peak values at the boundary. All these aspects lead to the correctness of the used algorithm. The rotation corresponding to the yx component is the one of the xy component, but with a minus sign. This occurs due to the definition the algorithm in C++ uses. In the case of my code R_M is defined in equation 3.22. This formula dictates the fact that the xy and yx components of the rotation should be equal to each other. However, in the C++ algorithm, they leave the identity matrix out of the formula, fact which only influences the sign, not the absolute value. The other matrix components xx and yy are equal to zero. Moreover, in over to avoid being redundant, the plot corresponding to the yx of the strain matrix is left out by

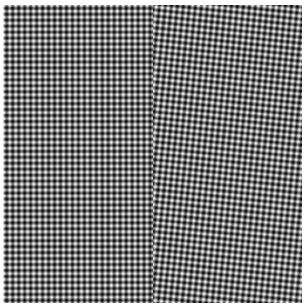
Figure 4.11: Grayscaled image of the pattern shown in 3.2 needed as input for the C++ algorithm.

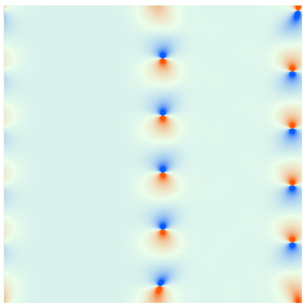
Figure 4.12: ϵ_{xx}, the xx component of the strain matrix computed for figure 4.11 through the C++ algorithm

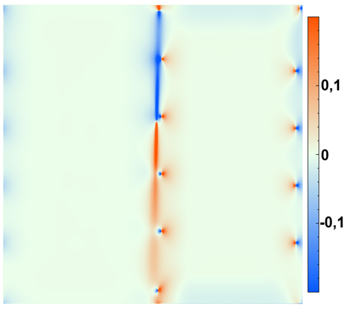

Figure 4.13: ϵ_{xy}, the xy component of the strain matrix which is computed for figure 4.11 through the C++ algorithm

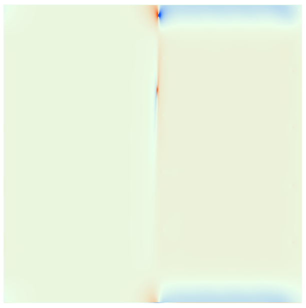
Figure 4.14: ϵ_{yy}, the yy component of the strain matrix which is computed for figure 4.11 through the C++ algorithm

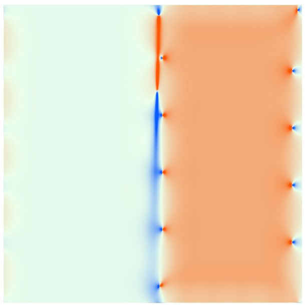
Figure 4.15: w_{xy}, the xy component of the rotation matrix which is computed for figure 4.11 through the C++ algorithm

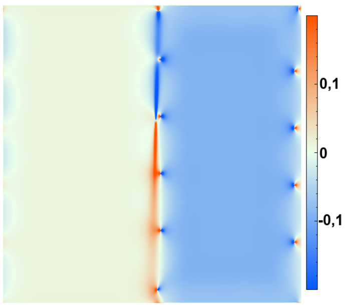

Figure 4.16: w_{yx}, the yx component of the strain matrix which is computed for figure 4.11 through the C++ algorithm

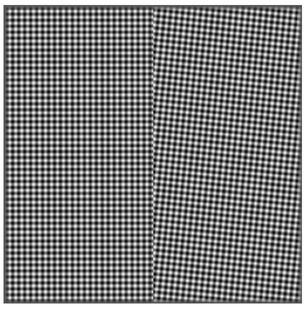
Figure 4.17: Grayscaled image of the pattern shown in 3.5 needed as input for the C++ algorithm.

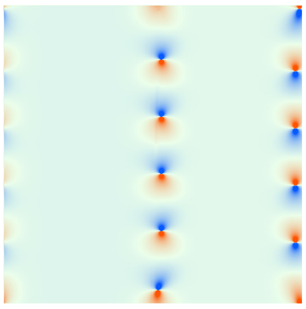
Figure 4.18: ϵ_{xx}, the xx component of the strain matrix which is computed for figure 4.17 through the C++ algorithm

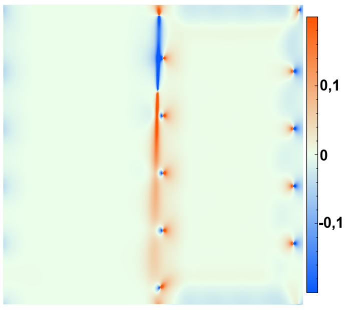

Figure 4.19: ϵ_{xy}, the xy component of the strain matrix which is computed for figure 4.17 through the C++ algorithm

the algorithm, due to symmetry.

3. Figure 4.17 consists of one layer corresponding to the simple straight lattice and one layer to the tilted lattice, with the window function (introduced in equation 3.2) around the entire lattice. This is the grasycaled version of the pattern shown in figure 3.5. Figures 4.18 and 4.20 are the xx, xy and yy components of the corresponding strain, figures 4.21 and 4.22 are the xy and yx components of the corresponding rotation matrix. The rotation and the strain matrices do not differ significantly from the previous case which analysed. This is acceptable, since attaching a smoothening window to the outside of the layers should not significantly influence the strain felt at the boundary in between the layers.

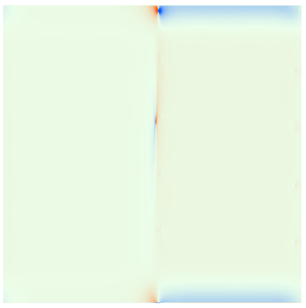
Figure 4.20: ϵ_{yy}, the yy component of the strain matrix which is computed for figure 4.17 through the C++ algorithm

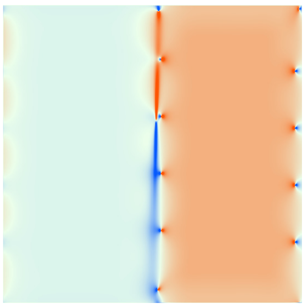
Figure 4.21: w_{xy}, the xy component of the rotation matrix which is computed for figure 4.17 through the C++ algorithm

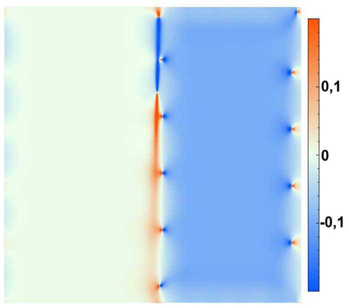

Figure 4.22: w_{yx}, the yx component of the rotation matrix which is computed for figure 4.17 through the C++ algorithm

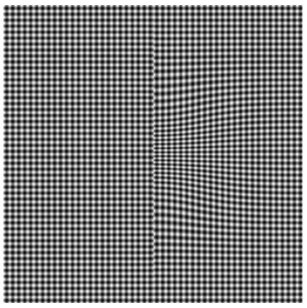
Figure 4.23: Greyscaled lattice corresponding of (left) perfect square layer, right shifted through Gaussian displacement

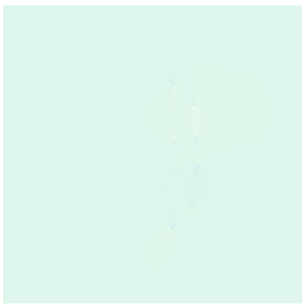
Figure 4.24: ϵ_{xx}, the xx component of the strain matrix which is computed for figure 4.23 through the C++ algorithm

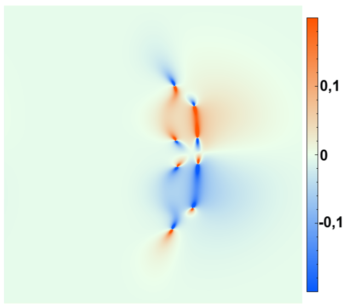

Figure 4.25: ϵ_{xy}, the xy component of the strain matrix which is computed for figure 4.23 through the C++ algorithm

4.4. Results for additional lattice patterns and filters

In this section I have introduced the lattice which have a shift different from zero in their components. Due to this shift, one already expects for the rotation and strain images to look different than the ones in the previous section.

1. Figure 4.23: This particular lattice has not been introduced in section 3.1 in order to avoid repetition, but I am stating in this section that it is formed out of two straight layers of lattices (both have $\theta = 0°$), the atoms in the right half being shifted with the amount given in equation 3.1. It is important to keep in the back of one's head the different axes notation and the fact that this is not a x shift anymore, but a y shift seeing as y is the vertical axis in the definition of [13]. From the strain figures one can see that now there peaks not only at the boundary between the layers, but also at coordinates where the shift of an atom is extreme. The rotation, outside of the shift, is also zero and this verifies what one expects, seeing as the right half as a whole is not rotated with respect to the left one. Just local rotation belonging to the shift in atoms is visible. Concerning the strain, once can observe that the peaks are located toward the center of the vertical axis which is the boundary of the two layers, since that is where the shift seems to have a high value. The images corresponding to the components of the strain and rotation matrices are depicted in figures 4.24 and 4.28, respectively. This is a very important lattice to test the strain: seeing as the displacements are defined only in the vertical region, the xx component of the strain matrix must the equal to zero, which is indeed clearly visible in figure 4.24.

2. Figure 4.29 is the grayscaled version of figure 3.4. It represents a lattice formed by two layers: the left half is a straight lattice, while the right half corresponds to a tilted lattice in which shift has been added as well. Seeing as, with respect to the last lattice which I introduced, this one has also rotation present globally. Again, the values of the rotation identify with the ones suggested by the initial tilt of the right layers (as always, $\theta = 6°$) One can see in the center of figure 4.27 how the local shift in atoms can annihilate the global rotation of the entire right layer and therefore

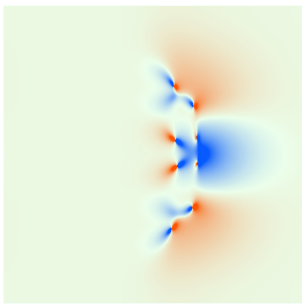
Figure 4.26: ϵ_{yy}, the yy component of the strain matrix which is computed for figure 4.23 through the C++ algorithm

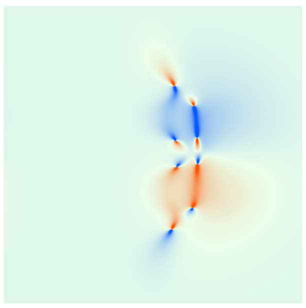
Figure 4.27: w_{xy}, the xy component of the rotation matrix which is computed for figure 4.23 through the C++ algorithm

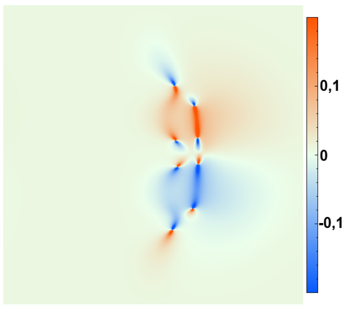

Figure 4.28: w_{yx}, the yx component of the rotation matrix which is computed for figure 4.23 through the C++ algorithm

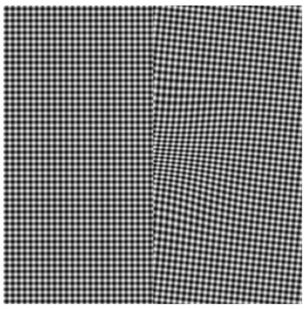
Figure 4.29: Grayscaled image of the pattern shown in 3.4 needed as input for the C++ algorithm.

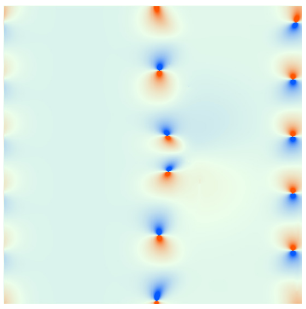
Figure 4.30: ϵ_{xx}, the xx component of the strain matrix which is computed for figure 4.29 through the C++ algorithm

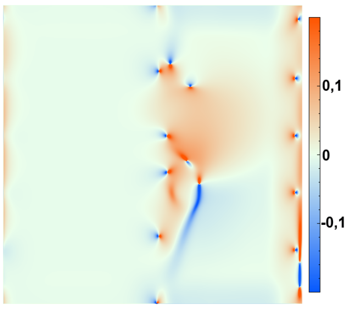

Figure 4.31: ϵ_{xy}, the xy component of the strain matrix which is computed for figure 4.29 through the C++ algorithm

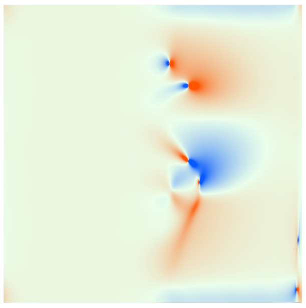
Figure 4.32: ϵ_{yy}, the yy component of the strain matrix which is computed for figure 4.29 through the C++ algorithm

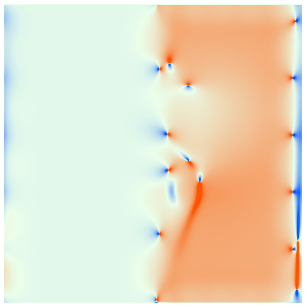
Figure 4.33: w_{xy}, the xy component of the rotation matrix which is computed for figure 4.29 through the C++ algorithm

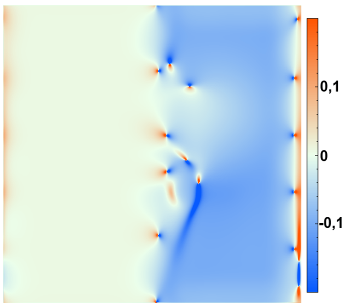

Figure 4.34: w_{yx}, the yx component of the rotation matrix which is computed for figure 4.29 through the C++ algorithm

created a rotation close to zero in that region. However, looking at the corresponding area in one of the strain components (see figure 4.26) one notices that in that region the strain has an extreme value, due to the fact that the contributions of the two operations (shift and tilt) add up in case of the strain. In figure 4.24 and 4.28 one can see the components of the strain matrix and the rotation matrix corresponding to figure 4.29.

3. Figure 4.35 is figure 4.29 with the smoothing window applied around its edges. This particular image is the grayscale version of the lattice pattern shown in figure 3.6. The corresponding strain and rotation values and figures (4.36 4.40) are almost identical to the ones obtained for figure 4.29. This is also to be expected, since adding a window to a lattice should not (significantly) change the strain which is felt at the boundary between the layers. The only influence it can have is related to the strain at the edges, with adjacent lattice patterns, of the same type, or different, depending on the analysed material.

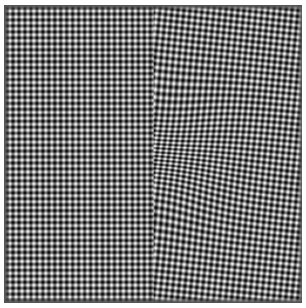
Figure 4.35: Grayscaled image of the pattern shown in 3.6 needed as input for the C++ algorithm

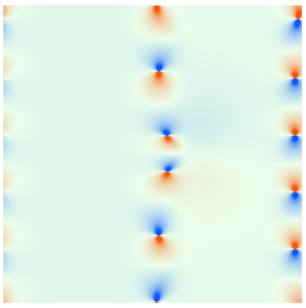
Figure 4.36: ϵ_{xx}, the xx component of the strain matrix which is computed for figure 4.35 through the C++ algorithm

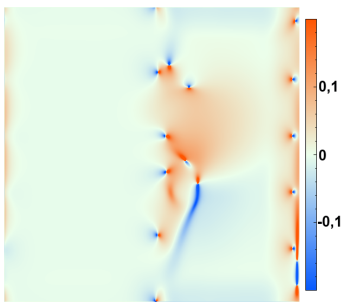

Figure 4.37: ϵ_{xy}, the xy component of the strain matrix which is computed for figure 4.35 through the C++ algorithm

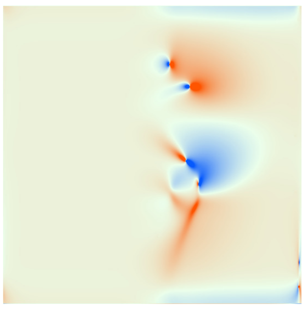
Figure 4.38: ϵ_{yy}, the yy component of the strain matrix which is computed for figure 4.35 through the C++ algorithm

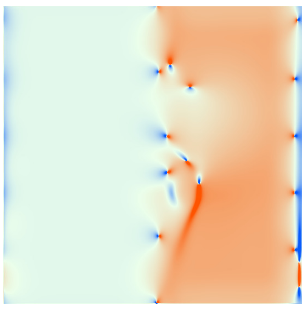
Figure 4.39: w_{xy}, the xy component of the strain matrix which is computed for figure 4.35 through the C++ algorithm

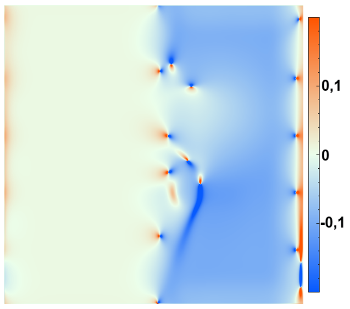

Figure 4.40: w_{yx}, the yx component of the strain matrix which is computed for figure 4.35 through the C++ algorithm

5 Conclusions

The main work that has been presented within this project revolves around computing the strain which arises at the atomic level of two-dimensional HRTEM images of several lattice patterns, due to local distortion or defects present in the crystal structure of the material in question. Having information about the strain is very important for an accurate understanding of the composition of the nanoscale of certain materials, with a wide range of application in the context of semiconductors, and it is computed by following steps outlined by the GPA method, a computational tool which requires the use of a computer algorithm and operations applied in the Fourier domain.
Within this project the HRTEM images are virtually simulated, together with the rest of the results, through a code written in the programming language Python. The steps of which the GPA method consists have been carefully introduced and explained in the context of my chosen lattice patterns, which vary from being perfect square lattices which contain no strain, to being lattices formed through shifting the atoms and/or rotating part of the lattice itself.

Once the lattice patterns are defined, the Bragg peak selection is explained in depth, followed by the process of masking around the selected peak. The masked image in the Fourier space is then converted into the real space. The phase of the latter is taken and the called raw phase image, which is followed by the computation of the reduced phase image, that is used to define the smooth reference region in the lattice pattern with respect to which the strain must be later computed. The step at which my Python code stops producing the desired and expected results is the one concerning the calculation of the optimal spacial frequencies (also called g-vectors) which must be done at every pixel of the lattice, in order to later be able to compute and map the strain at every pixel with respect to the chosen reference region. I expect, due to various reasons which have been introduced in the body of the thesis, that the explanation why my code does not return the desired output has to do with the incompatibility between the mathematical function which needs to be minimised or fitted and the minimisation or curve fit functions used in Python, respectively. This belief takes root in the fact that if not given as initial guess to the Python fitting function the exact desired output as initial guess, the code returns wrong g-vectors.

Even though the results which I have presented so far through means of my code have not yet been expanded to their maximal capacity, I still proceeded to analysing the strain mapping of my defined lattices through means of a pre-existing algorithm written in C++. In this way I am able to obtain the desired results and to discuss the outputs which my code should also provide. The peaks which one expects to obtain at the boundary between the two layers forming the lattice corresponding to extreme values of the strain are perfectly visible. Moreover, it turns out that by applying both a local shift in the positions of the atoms and a rotation of a whole part of the lattice one can obtain zero total rotation in some parts of the lattice, but which correspond to an extreme value of the strain. This can be explained due to the fact that, by definition, strain is the symmetric part of the local distortion, while the rotation is the antisymmetric part.

With the detailed background of theoretical information and discussions on strain, lattices and the GPA method, accompanied by concrete examples and applications which the reader cannot only use, but also modify according to their preferences, I consider that the goal of my project has been fulfilled, so far. The next steps regarding future projects involve not only creating a fully-functioning algorithm implemented in Python which analyses lattice strain through the GPA method and contains plenty of details and the possibility of being adapted at free will by the user, but also one which investigates alternative methods, such as the Peak Pair Algorithm and compares them amongst each other. This is not only a very exciting academic challenge which I am looking forward to, but also a computational application opportunity not to be missed.

A

Appendix

A.1. Tables with symbols and variables

A.1.1. Table containing the important mathematical symbols and their meaning

Mathematical symbol	Meaning
$a + b, a - b, a \cdot b, \frac{a}{b}, a = a$	Basic operators: summation, subtraction, multiplication, division, identity
a^b	a to the power b
$a > b, a < b, a \leq b, a \geq b$	a greater, less, less or equal, greater or equal to b
$\vec{a}$	Vector notation
$\vec{a} \cdot \vec{b}$	scalar product of vectors
$\sum$	Summation sign
∞	Infinity
e	The Euler’s number
i	The unit imaginary number
$\int$	Integral
π	Pi Greek
ω	Omega Greek,
α	Alpha Greek
β	Beta Greek
$f(x, y)$	Function which depends on variables x and y
$\frac{\partial f}{\partial x}$	Derivative of f with respect to x (partial differential equation)
(x, y)	x and y, Coordinates of a vector
mod	Modulo
$\|\|\vec{r}\|\|$	Absolute value of a two-dimensional vector height

A.1.2. Table containing important variables defined in the computational and coding space, respectively. The term ”(used in method)” means that the variable in question also appears in the computational method.

Code variable	Meaning
S	half size of square side lattice (used in method)
r	Value of vector in reciprocal space (used in method)
θ	Angle of tilt for lattice (used in method)
xshift	Value of shift of atoms vertically (used in method)
original	Name of the original straight simple lattice
fourier_original	FT original lattice
tilted	Original and tilted lattice with θ
left_half_original	Takes left vertical half of original lattice
right_half _original	Takes right vertical half of original lattice
fourier_combined	FT concatenated halves from layers
fourier_combined_edges	FT concatenated halves and windows
pks	The 4 peaks returned by the python function (also in code of form g_{select})
fourier_combined_filtered	The masked FT around a selected peak
fourier_combined_filtered_new	Masked FT around the second selected peak
angle_fourier_combined_iift	The raw phase image for first peak
angle_fourier_combined_iift _	The raw phase image second peak
reduced reduced_new	Reduced phase image corresponding to first peak Reduced phase image corresponding to second peak
f_summation	Definition of summation function for minimisation
O_left	Reference region (also used in method)
func	Function to be curve-fitted
r_G	Dictionary which contains optimal values of $\vec{g}$ and $\vec{c}$
coord_list	The region R on which min is performed (taking values in all lattice pixels)
G_all_pixels	Matrix containing only $\vec{g}$ values for all pixels
A_all_pixels	Matrix containing only $\vec{a}$ values for all pixels
c_all_pixels	Matrix containing only $\vec{c}$ values for all pixels
u_E	Displacement vector in all pixels (also in method)
u_L	Displacement vector in all pixels (also in method)
e_E	Displacement vector in all pixels (also in method)
e_L	Displacement vector in all pixels (also in method)
x_0	x coordinate center reference region (also in method)
y_O	x coordinate center reference region (also in method)
A_O	A matrix for center of region O (also in method)
G_O	G matrix for center of region O (also in method)
e_x, e_y	x and y components of distortion vector (also in method)
R_xx, R_xy, R_yx, R_yy	components of rotation matrix (also in method)
$\epsilon_{xx}, \epsilon_{xy}, \epsilon_{yx}, \epsilon_{yy}$	components of strain matrix (also in method)

A.2. Code availability

```
[ ]: #here I import the used libraries, but also the pre-existing Pyhton
      ↪functions cited in my report
     import dask.array as da
     import matplotlib
     import matplotlib.pyplot as plt
     import numpy as np
     import os
     #import cv2
     from PIL import Image, ImageEnhance
     import colorcet

     import cmath
     import scipy
     from skimage.feature import peak_local_max
     from skimage.io import imread, imshow
     from skimage.color import rgb2hsv, rgb2gray, rgb2yuv
     from skimage import color, exposure, transform
     from skimage.exposure import equalize_hist

     import scipy.ndimage as ndi

     from dask.distributed import Client, LocalCluster

     from moisan2011 import per
     from pyGPA.phase_unwrap import phase_unwrap
     from pyGPA.imagetools import fftplot, gauss_homogenize2,
      ↪gauss_homogenize3
     from pyGPA.mathtools import wrapToPi, standardize_ks
     import pyGPA.geometric_phase_analysis as GPA
     from copy import deepcopy

     from latticegen import combine_ks, squarelattice_gen
     %matplotlib inline
     from turtle import Screen, Turtle
     #import test_geometric_phase_analysis.py as test_GPA
     #from .imagetools import gauss_homogenize2, fftbounds, fftplot, trim_nans2

     from scipy.ndimage.filters import gaussian_filter
```

```
[ ]: #this function adds nice arrows on axis due to the peculiar way Pyhton
      ↪defines them
     def arrowed_spines(fig, ax):

         xmin, xmax = ax.get_xlim()
         ymin, ymax = ax.get_ylim()





         # get width and height of axes object to compute
         # matching arrowhead length and width
         dps = fig.dpi_scale_trans.inverted()
         bbox = ax.get_window_extent().transformed(dps)
         width, height = bbox.width, bbox.height
```

```
    # manual arrowhead width and length
    hw = 1./20.*(ymax-ymin)
    hl = 1./20.*(xmax-xmin)
    lw = 1. # axis line width
    ohg = 0.3 # arrow overhang

    # compute matching arrowhead length and width
    yhw = hw/(ymax-ymin)*(xmax-xmin)* height/width
    yhl = hl/(xmax-xmin)*(ymax-ymin)* width/height

    # draw x and y axis
    ax.arrow(xmin, 0, xmax-xmin, 0., fc='k', ec='k', lw = lw,
             head_width=hw, head_length=hl, overhang = ohg,
             length_includes_head= True, clip_on = False, color="gray")

    ax.arrow(0, 0,0, 500, fc='k', ec='k', lw = lw,
             head_width=yhw, head_length=-yhl, overhang = ohg,
             length_includes_head= True, clip_on = False, color="gray")
```

```
[ ]: #this function changes the color gradiant of a particular image
def plot_color_gradients(category, cmap_list):
    # Create figure and adjust figure height to number of colormaps
    nrows = len(cmap_list)
    figh = 0.35 + 0.15 + (nrows + (nrows - 1) * 0.1) * 0.22
    fig, axs = plt.subplots(nrows=nrows + 1, figsize=(6.4, figh))
    fig.subplots_adjust(top=1 - 0.35 / figh, bottom=0.15 / figh,
                        left=0.2, right=0.99)
    axs[0].set_title(f'{category} colormaps', fontsize=14)

    for ax, name in zip(axs, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=plt.get_cmap(name))
        ax.text(-0.01, 0.5, name, va='center', ha='right', fontsize=10,
                transform=ax.transAxes)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
    for ax in axs:
        ax.set_axis_off()

    # Save colormap list for later.
    cmaps[category] = cmap_list
```

```
[ ]: #This section has been commented out since it serves as the series of
  ↪commands used to plot nice figures with nice axis, size etc
#plt.figure(figsize=(10,10))
#plt.xticks(fontsize=20)
#plt.yticks(fontsize=20)

#plt.imshow((angle_fourier_combined_new_iift), cmap='rainbow')
#fig = plt.gcf()
#fig.set_facecolor('white')
#ax = plt.gca()

#arrowed_spines(fig, ax)
```

This first section of the code is dedicated towards modelling the lattice, computing its Fourier Transform.

```
[ ]: S = 250 #half of the size of the suqare lattice side
     r_k = 0.1    #length of reciprocal lattice vector
     theta1=0     #angle of rotation of lattice, which can take multiple values
     theta2=6
     theta2_prime=-6
     psi= 0.00             #these are parameters which are not important in
      ↪the definition of the square lattice, they have more
                           #importance in cases such as the hexagonal lattice
     shift=np.array((0, 0))
     kappa=1
     a_0 = 0.246

     order = 1

     original = squarelattice_gen(r_k, theta1, order, size=2*S, kappa=kappa,
      ↪shift=shift).compute()
     original_prime = squarelattice_gen(r_k, theta1, order, size=2*S,
      ↪kappa=kappa, shift=shift).compute()



     #Gaussian blurring:
     #original = gaussian_filter(original_perfect_lattice, sigma=2)

     #adding distortion in the y direction:


     xp, yp = da.meshgrid(np.arange(-S,S), np.arange(-S,S), indexing='ij')
     xshift = 0.5*xp*np.exp(-0.5 * ((xp/(2*S/8))**2 + 1.2*(yp/(2*S/6))**2))
     plt.imshow(abs(xshift))
     noise = np.stack((xshift,np.zeros_like(xshift)), axis=0)

     #original = squarelattice_gen(r_k, theta1, order, size=2*S, kappa=kappa,
      ↪shift=noise).compute()
     c=plt.imshow(original.T,
                  cmap='cet_fire_r',
                 extent=[-S*r_k*a_0, S*r_k*a_0, S*r_k*a_0, -S*r_k*a_0])
```

```
[ ]: #computing the Fourier transform of the original lattice

     fourier_original = np.fft.fftshift(np.fft.fft2(original))
     plt.figure(num=None, figsize=(8, 6), dpi=80)
     plt.imshow((abs(fourier_original)), cmap='gray');
```

```
[ ]: #plotting different tilted matrices, with or without noise
     tilted = squarelattice_gen(r_k, theta2_prime, order, size=2*S,
      ↪kappa=kappa, shift=noise).compute()
     d2=plt.imshow(tilted,
                  cmap='cet_fire_r',
                 extent=[-S*r_k*a_0, S*r_k*a_0, S*r_k*a_0, -S*r_k*a_0])

     tilted_simple= squarelattice_gen(r_k, theta2_prime, order, size=2*S,
      ↪kappa=kappa, shift=shift).compute()
```

```
[ ]: #taking halves of lattices
     left_half_original = original[:,:250]
     right_half = original[:,250:]
     plt.imshow(left_half_original)
```

```
[ ]: left_half_tilted= tilted[:,:250]
     right_half_tilted = tilted[:,250:]
     plt.imshow(right_half_tilted)

     left_half_simple=tilted_simple[:,:250]
```

```
[ ]: #defining the combined lattices by combining the two halves
     combined=[]
     combined=np.concatenate((left_half_original,left_half_simple),axis=1)
     plt.figure(figsize=(10,10))
     plt.xticks(fontsize=20)
     plt.yticks(fontsize=20)

     plt.imshow((combined))
     plt.axis('off')
```

```
[ ]: #defining the window function
     def window_function(x,delta):
         if np.abs(x)<delta or np.abs(x)>500-delta :
             return (np.sin(np.pi*x/(2*delta)))**2
         else:
             return 1
```

```
[ ]: Delta=10
     combined_edges=np.zeros((500,500))

     for i in range (np.shape(combined)[1]):
         for j in range (np.shape(combined)[0]):
             combined_edges[i][j]=combined[i][j]*window_function(i,
       ↪Delta)*window_function(j, Delta)
```

```
[ ]: #defining the ft of the final combined lattices, window applied to lattice

     fourier_combined_edges = np.fft.fftshift(np.fft.fft2(combined_edges))
     plt.figure(num=None, figsize=(8, 6), dpi=80)
     plt.imshow((abs(fourier_combined_edges)), cmap='gray');


     plt.figure(figsize=(10,10))
     plt.xticks(fontsize=20)
     plt.yticks(fontsize=20)

     plt.imshow(abs(fourier_combined_edges), cmap="gray")
     fig = plt.gcf()
     fig.set_facecolor('white')
     ax = plt.gca()

     arrowed_spines(fig, ax)
```

```
[ ]: #some more ft of lattices
     fourier_combined = np.fft.fftshift(np.fft.fft2(combined))
     plt.figure(num=None, figsize=(8, 6), dpi=80)
     plt.imshow((abs(fourier_combined)), cmap='gray');
```

The second section of the code is dedicated towards finding the peaks in the Fourier Transform.

```
[ ]: h1, w1 = np.shape(fourier_combined) #the lattice environment is divided
      ↪into rows and columns components
     Y1, X1 = np.ogrid[:h1, :w1]
     row1,col1 = np.unravel_index(np.argsort(fourier_combined.
      ↪ravel()),fourier_combined.shape)


     fig_a, ax = plt.subplots()
     draw_circle = plt.Circle(tuple(reversed((250, 300))), 10, alpha = 0.3)
      ↪#a circle is drawn around the given coordinates
     ax.add_artist(draw_circle)
     ax.imshow((abs(fourier_combined)), cmap='gray');
     import pyGPA.geometric_phase_analysis as GPA
     from pyGPA.geometric_phase_analysis import fftbounds, peak_local_max,
      ↪remove_negative_duplicates, fftplot,
      ↪_decrease_threshold,select_closest_to_triangle
     #importing the function from the Github repository
     def extract_primary_ks(image, plot=False, threshold=0.7,
      ↪pix_norm_range=(2, 200),
                            sigma=1, NMPERPIXEL=1., DoG=True):
         """Attempt to extract primary k-vectors from an image from a smoothed
         version of the Fourier transform.

         Recursively adapts parameters until a satisfactory solution is found.

         Parameters
         ----------
         image : ndarray
             The realspoace input image
         plot : bool
            Whether to plot a debug plot containing the FFT, the detected peaks
             and the selected peaks. Also prints more debug info
         threshold : float, default: 0.7
             relative threshold for peak height.
         pix_norm_range : 2-tuple of int, default: (2, 200)
         sigma : float, default = 1
             width of the gaussian smoothing of the FFT before peaks are
      ↪extracted.
         NMPERPIXEL : float
         DoG : bool
             (Difference of Gaussians)
             Whether to divide the smoothed FFT by a sigma=50 smoothed version

         Returns
         -------
         primary_ks : ndarray (N, 2)
             list of main k-vectors
         all_ks : ndarray (N+M, 2)
```

```
        list of all found k-vectors
    """
    image = image - image.mean()
    pd, _ = per(image, inverse_dft=False)
    fftim = np.abs(np.fft.fftshift(pd))
    kxs, kys = [fftbounds(n) for n in fftim.shape]
    smooth = ndi.filters.gaussian_filter(fftim, sigma=sigma)
    if DoG:
        smooth -= ndi.filters.gaussian_filter(fftim, sigma=50)

    center = np.array(smooth.shape)//2
    # min_distance=5, threshold_rel=np.quantile(smooth, threshold))
    cindices = peak_local_max(smooth, threshold_rel=threshold)
    coords = cindices - center
    selection = np.logical_and((np.linalg.norm(coords, axis=1) <
  ↪pix_norm_range[1]),
                               (np.linalg.norm(coords, axis=1) >
  ↪pix_norm_range[0]))
    cindices = cindices[selection]
    coords = coords[selection]  # exclude low intensity edge area and
  ↪center stigmated dirt spots

    # coords = np.vstack((coords, [0,0])) # reinclude center spot
    all_ks = np.array([kys[cindices.T[0]], kxs[cindices.T[1]]]).T
    # Select only one direction for each pair of k,-k
    all_ks = remove_negative_duplicates(all_ks)
    newparams = False
    if len(all_ks) < 4:
        newparams = True
        if len(all_ks) == 0:
            # print(f"no ks at: {threshold:.4f}")
            if threshold > _decrease_threshold(threshold):
                threshold = _decrease_threshold(threshold)
            else:
                print("No ks found at minimum threshold!")
                newparams = False
        else:
            coordsminlength = np.linalg.norm(coords, axis=1).min()
            if coordsminlength < 5 * sigma:
                sigma = coordsminlength / 6
            elif threshold > 0.2*np.max([smooth[cindex[0], cindex[1]] for
  ↪cindex in cindices]):
                threshold = 0.2*np.max([smooth[cindex[0], cindex[1]] for
  ↪cindex in cindices])
            elif threshold > _decrease_threshold(threshold):
                threshold = _decrease_threshold(threshold)
            else:
                print("Can't find enough ks!")
                newparams = False
        if newparams:
            primary_ks, all_ks = extract_primary_ks(image, plot=False,
                                                    threshold=threshold,
                                                    sigma=sigma,

  ↪pix_norm_range=pix_norm_range)
```

```
        else:
            primary_ks = all_ks.copy()

    knorms = np.linalg.norm(all_ks, axis=1)
    if not newparams:
        primary_ks = all_ks.copy()

    if len(primary_ks) != 4:
        if len(primary_ks) > 4:
            # print(f"Too many primary ks {len(primary_ks)}")
            primary_ks = select_closest_to_triangle(all_ks)
        elif len(all_ks) > 8:
            # print("all_ks > 3, selecting 3 with most similar length")
            # primary_ks = all_ks[np.argpartition(np.abs(knorms-knorms.
↪mean()), 3)[:3]]
            if plot:
                print("all_ks > 3 but not enough primary_ks, selecting
↪closest to triangle")
            primary_ks = select_closest_to_triangle(all_ks)
       elif threshold > _decrease_threshold(threshold) and not newparams:
            if plot:
                print(f"pks<3, all_ks < 6, decreasing threshold
↪{threshold:.3f}")
            threshold = _decrease_threshold(threshold)
            primary_ks, all_ks = extract_primary_ks(image, plot=False,
                                                    threshold=threshold,
                                                    sigma=sigma,
                                                   
↪pix_norm_range=pix_norm_range)
        else:
            if plot:
                print("pks < aks=3", len(all_ks), len(primary_ks))
            primary_ks = all_ks.copy()
    if plot:
        fig, ax = plt.subplots(ncols=2, figsize=[12, 8])
        fftplot(np.transpose(smooth), d=NMPERPIXEL, ax=ax[0],
                levels=[smooth.max()*threshold*0.8],
                contour=False, pcolormesh=False, origin='upper')
        ax[0].set_xlabel('k (periods / nm)')
        ax[0].set_ylabel('k (periods / nm)')
        ax[0].scatter(*(all_ks/NMPERPIXEL).T, color='red',
                      alpha=0.2, s=50)
        ax[0].scatter(*(primary_ks/NMPERPIXEL).T, color='black',
                      alpha=0.7, s=50, marker='x')

        circle = plt.Circle((0, 0), 2.*knorms.min()/NMPERPIXEL,
                            edgecolor='y', fill=False, alpha=0.6)
        ax[0].add_artist(circle)
        axlim = kxs[min(center[0] + pix_norm_range[1], len(kxs)-1)]
        ax[0].set_xlim(-axlim, axlim)
        ax[0].set_ylim(-axlim, axlim)
        ax[1].imshow(image, origin='upper')
        for r in [kxs[center[0]+s] for s in pix_norm_range]:
            circle = plt.Circle((0, 0), r/NMPERPIXEL,
                                edgecolor='w', fill=False, alpha=0.6)
            ax[0].add_artist(circle)
        plt.title(plot)
    return primary_ks, all_ks

pks,_ = extract_primary_ks(combined, pix_norm_range=(2,70), plot=True, )
```

```
[ ]: print(pks)
```

```
[ ]: #to check whether the right peaks have been chosen
fig_a, ax = plt.subplots()
draw_circle = plt.Circle(tuple(reversed((250, 300))), 10, alpha = 0.3)
ax.add_artist(draw_circle)
ax.imshow((abs(fourier_combined)), cmap='gray');
```

This third section is dedicated towards plotting masks around the selected peaks

```
[ ]: #h1, w1 = np.shape(fourier_combined_bounds)
#Y1, X1 = np.ogrid[:h1, :w1]
#row1,col1 = np.unravel_index(np.argsort(fourier_combined_bounds.
  ↪ravel()),fourier_combined_bounds.shape)

center = [249, 249]
peak1=[250,300]
peak2=[199, 254]
dist_from_peak1 = np.sqrt((X1 - peak1[1])**2 + (Y1 - peak1[0])**2)    
  ↪#Y1-corresponds to row coordinate here
#! changed row and col line above, influences mask

dist_from_peak2 = np.sqrt((X1 - peak2[1])**2 + (Y1 - peak2[0])**2)
g1y_test=(peak1[1]-center[0])/500 #we hereby define the values of the x
  ↪and y components of the g vector computed by hand
g1x_test=(peak1[0]-center[1])/500

g1_test=[g1x_test, g1y_test] #the vector composed of the previous 2
  ↪components of g1

g2y_test=(peak2[1]-center[0])/500
g2x_test=(-center[1]+peak2[0])/500

g2_test=[g2x_test, g2y_test]
mask1 = (dist_from_peak1 <= 10)
mask2 = (dist_from_peak2 <= 10)


fourier_combined_edges_masked1 = deepcopy(fourier_combined_edges)
fourier_combined_edges_masked2 = deepcopy(fourier_combined_edges)
fourier_combined_edges_masked1[~mask1] = 0
fourier_combined_edges_masked2[~mask2] = 0

wc1 = np.hanning(500)
wr1 = np.hanning(500)
wc1, wr1 = np.meshgrid(wc1,wr1)
mask = [wc1,wr1]
window1 = mask[0]*mask[1]
```

```
#applying mask on ft
fourier_combined_filtered= fourier_combined_edges_masked1*window1
#getting the ift of the masked ft
fourier_combined_iift = np.fft.ifft2(np.fft.
 ↪ifftshift(fourier_combined_filtered))

#applying mask on ft
fourier_combined_filtered_new = fourier_combined_edges_masked2*window1
#getting the ift of the masked ft
fourier_combined_new_iift = np.fft.ifft2(np.fft.
 ↪ifftshift(fourier_combined_filtered_new))
```

```
[ ]: plt.imshow(abs(fourier_combined_filtered))
     plt.imshow(abs(fourier_combined_filtered_new))
```

```
[ ]: plt.imshow(abs(fourier_combined_filtered_new))
```

```
[ ]: #computing the first raw phase image
     angle_fourier_combined_iift=np.angle(fourier_combined_iift)


     plt.figure(figsize=(10,10))
     plt.xticks(fontsize=20)
     plt.yticks(fontsize=20)

     plt.imshow((angle_fourier_combined_iift), cmap="rainbow")
     fig = plt.gcf()
     fig.set_facecolor('white')
     ax = plt.gca()

     arrowed_spines(fig, ax)
     #abs(angle_fourier_combined_iift)
     #print(np.min(angle_fourier_combined_iift))

     print(np.max(angle_fourier_combined_iift))
     print(np.min(angle_fourier_combined_iift))
```

```
[ ]: #computing the second raw phase image
     angle_fourier_combined_new_iift=np.angle(fourier_combined_new_iift)


     plt.figure(figsize=(10,10))
     plt.xticks(fontsize=20)
     plt.yticks(fontsize=20)

     plt.imshow((angle_fourier_combined_new_iift), cmap="rainbow")
     fig = plt.gcf()
     fig.set_facecolor('white')
     ax = plt.gca()

     arrowed_spines(fig, ax)
```

Peak 1 and g1, peak 2 and g2:

```
[ ]: #computin the images needed to be subtracted in order to obtain reduced
      ↪phases
```

```
g_1_select = pks[3]
g_2_select = -pks[0]

g_3_select=[-0.1, 0.01]
g_4_select=[0.01, 0.1]
subtract=np.
 ↪zeros((len(angle_fourier_combined_iift),len(angle_fourier_combined_iift[0])))

plt.figure(figsize=(15,15))

for i in range (len(angle_fourier_combined_iift)):
    for j in range (len(angle_fourier_combined_iift[0])):
        subtract[i][j]=((2*np.pi*np.matmul(g_4_select,[i,j]))+np.
 ↪pi)%(2*np.pi)-np.pi    #here we use the by hand computated vector g

plt.imshow(subtract, cmap='rainbow')

print(subtract)
```

[ ]:
```
subtract_new=np.
 ↪zeros((len(angle_fourier_combined_iift),len(angle_fourier_combined_iift[0])))

plt.figure(figsize=(15,15))

for i in range (len(angle_fourier_combined_iift)):
    for j in range (len(angle_fourier_combined_iift[0])):
        subtract_new[i][j]=((2*np.pi*np.matmul(g_3_select,[i,j]))+np.
 ↪pi)%(2*np.pi)-np.pi    #here we use the by hand computated vector g

plt.imshow(subtract_new, cmap='rainbow')

print(subtract)
```

[ ]:
```
#plotting the two reduced phases
reduced=(angle_fourier_combined_iift-subtract +np.pi)%(2*np.pi)-np.pi
plt.figure(figsize=(10,10))
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

plt.imshow((reduced), cmap="rainbow")
fig = plt.gcf()
fig.set_facecolor('white')
ax = plt.gca()

arrowed_spines(fig, ax)

reduced_filtered=(angle_fourier_combined_iift-subtract +np.pi)%(2*np.pi)-
np.pi
```

[ ]:
```
reduced_new=(angle_fourier_combined_new_iift-subtract_new +np.pi)%(2*np.
 ↪pi)-np.pi
plt.figure(figsize=(10,10))
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.imshow((reduced_new), cmap="rainbow")
fig = plt.gcf()
fig.set_facecolor('white')
ax = plt.gca()

arrowed_spines(fig, ax)

reduced_filtered=(angle_fourier_combined_iift-subtract +np.pi)%(2*np.pi)-
np.pi
```

```
[ ]: #defyning region for testing O:

     O=[]
     O=reduced[50:70,50:70]

     #manually introduce the boundaries of defined region O
     Ox_1=0
     Ox_2=499
     Oy_1=0
     Oy_2=499



     #c1_test=angle_fourier_combined_iift[324][]

     print(np.max(O))
     print(np.min(O))

     c1_test=-(np.max(O)+np.min(O))/(2*np.pi)     #c1 computed by hand as
       ↪weighted sum of max and min values in defined origin
```

```
[ ]: O_new=[]
     O_new=reduced_new[50:70,50:70]

     #manually introduce the boundaries of defined region O
     Ox_new_1=50
     Ox_new_2=150
     Oy_new_1=50
     Oy_new_2=150



     #c1_test=angle_fourier_combined_iift[324][]

     print(np.max(O_new))
     print(np.min(O_new))

     c2_test=-(np.max(O_new)+np.min(O_new))/(2*np.pi)     #c2 computed by hand
       ↪as weighted sum of max and min values in defined origin
```

```
[ ]: O_prime=O+2*np.pi*c1_test



     print(np.sum(O_prime*O_prime))
print(np.shape(O_prime*O_prime))
```

```
[ ]: O_prime_new=O+2*np.pi*c2_test



     print(np.sum(O_prime_new*O_prime_new))

     print(np.shape(O_prime_new*O_prime_new))
```

```
[ ]: #attampt to obtain minimum through summation
     def f_summation (x, pattern, O_b):
         g_x, g_y, c = x
         sum_value = 0
        # for i in range (O_list[1]-O_list[0]):
         for x_c in np.arange(O_b[0],O_b[1] + 1):
             for y_c in np.arange(O_b[2],O_b[3] + 1):

                 #print("[", pattern[x_c][y_c],image[x_c][y_c] ,"]")
                 sum_value += (pattern[x_c][y_c] - np.angle(np.exp(np.
      ↪complex(0,(2 * np.pi * ((g_x*x_c + g_y * y_c) - c))))))**2
         return sum_value
```

```
[ ]: O_list_new = [Ox_new_1, Ox_new_2,Oy_new_1,Oy_new_2]
     O_list = [Ox_1, Ox_2, Oy_1, Oy_2]
```

```
[ ]: #attampt to obtain minimum through summation
     O_right=[350,450, 350,450]
     from scipy.optimize import fmin

     min_parameters = fmin(func = f_summation, x0 = [g_4_select[0],
      ↪g_4_select[1], c1_test], args = (angle_fourier_combined_iift,
      ↪O_right), xtol = 1e-1, ftol = 1e-1, disp = False)

     g1_x_min, g1_y_min, c_min = min_parameters
     print(min_parameters)
     print(g1_x_min)
     print(g1_y_min)
     print(c_min)
     print(f_summation([g1_x_min, g1_y_min, c_min],
      ↪angle_fourier_combined_iift, O_right))
     print(f_summation([g_1_select[0], g_1_select[1], c1_test],
      ↪angle_fourier_combined_iift, O_right))
     g_matrix_test=np.zeros((500,500))
     g_difference=np.zeros((500,500))
     g_sum_squares=0
     for x_c in range (0,500):
         for y_c in range (0,500):
             g_matrix_test[x_c][y_c]=np.angle(np.exp(np.complex(0,(2 * np.pi *
      ↪((g1_x_min*x_c + g1_y_min * y_c) - c_min)))))
             g_difference[x_c][y_c]=angle_fourier_combined_new_iift[x_c][y_c]
      ↪- g_matrix_test[x_c][y_c]

     for i in range (O_right[0],O_right[1]+1):
```

```
    for j in range (O_right[2],O_right[3]+1):
        g_sum_squares+=g_difference[i][j]**2
print("this is the sum of squares", g_sum_squares)


plt.figure(figsize=(10,10))
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.imshow( g_difference, cmap='RdBu', vmin=-2*np.pi, vmax=2*np.pi)
#plt.imshow((g_matrix_test), cmap='RdBu', vmin=-np.pi, vmax=np.pi)
fig = plt.gcf()
fig.set_facecolor('white')
ax = plt.gca()

arrowed_spines(fig, ax)
print(np.min(g_difference))
print(np.max(g_difference))
```

```
[ ]: O_left=[100,450,50,200] #reference
     O_right=[270,290,270,290] #region R for testing strain wrt O_left
       ↪(reference)
     print(g_1_select)
```

```
[ ]: #attempt to compute minimum through means of a curve fit
     %%time
     from scipy.optimize import curve_fit
     def func(var, g_x,g_y,c):
         x,y=var
         return (2*np.pi*(g_x*x + g_y*y - c -np.pi))%(2*np.pi) + np.pi
     x_O=np.linspace(O_left[0], O_left[1], O_left[1]- O_left[0]+1, dtype=int)
     y_O=np.linspace(O_left[2], O_left[3], O_left[3]- O_left[2]+1, dtype=int)

     XO,YO=np.meshgrid(x_O,y_O)
     dataO=np.vstack((XO.ravel(), YO.ravel()))
     #print(X1.shape)
     #print(data1)

     Z1_O = np.zeros(XO.shape)

     Z1_O=angle_fourier_combined_iift[O_left[0]:O_left[1]+1, O_left[2]:
       ↪O_left[3]+1]
     Z2_O = np.zeros(XO.shape)

     Z2_O=angle_fourier_combined_new_iift[O_left[0]:O_left[1]+1, O_left[2]:
       ↪O_left[3]+1]
     p0_1=2*np.pi*g_1_select[0],2*np.pi*g_1_select[1], c1_test
     p0_2=g_2_select[0],g_2_select[1], c2_test
     popt1_O, pcov1_O = curve_fit(func, dataO, Z1_O.ravel(), p0_1)
     popt2_O, pcov2_O = curve_fit(func, dataO, Z2_O.ravel(), p0_2)


     x_R=np.linspace(O_right[0], O_right[1], O_right[1]- O_right[0]+1,
       ↪dtype=int)
     y_R=np.linspace(O_right[2], O_right[3], O_right[3]- O_right[2]+1,
       ↪dtype=int)
```

```
XR,YR=np.meshgrid(x_R,y_R)
dataR=np.vstack((XR.ravel(), YR.ravel()))
Z1_R = np.zeros(XR.shape)
Z1_R=angle_fourier_combined_iift[O_right[0]:O_right[1]+1, O_right[2]:
 ↪O_right[3]+1]

Z2_R = np.zeros(XR.shape)
Z2_R=angle_fourier_combined_new_iift[O_right[0]:O_right[1]+1, O_right[2]:
 ↪O_right[3]+1]


pR_1=g_4_select[0],g_4_select[1], c1_test
pR_2=g_3_select[0],g_3_select[1], c2_test

popt1_R, pcov1_R = curve_fit(func, dataR, Z1_R.ravel(), pR_1)
popt2_R, pcov2_R = curve_fit(func, dataR, Z2_R.ravel(), pR_2)
#print(Z1.shape)
#print(Z1)
#print(x_w)
#print(popt1_O)
#print(popt2_O)
#print(popt1_R)
print(popt2_R)

plt.imshow(Z2_R)

#p0_2=g_3_select[0],g_3_select[1], c2_test
#print(curve_fit(func, (x1_w, y1_w),z_1,p0_1))
#print(curve_fit(func, (x2_w, y2_w),z_2,p0_2))
print(c1_test)
print(c2_test)


image_test=np.zeros((O_right[1]- O_right[0]+1,O_right[1]- O_right[0]+1))
for x_c in range (O_right[1]- O_right[0]+1):
    for y_c in range (O_right[1]- O_right[0]+1):
        image_test[x_c][y_c]= func((x_R[x_c], x_R[y_c]),
 ↪g_3_select[0],g_3_select[1],c2_test)
plt.imshow(image_test)

image_test1=np.zeros((O_right[1]- O_right[0]+1,O_right[1]- O_right[0]+1))
for x_c in range (O_right[1]- O_right[0]+1):
    for y_c in range (O_right[1]- O_right[0]+1):
        image_test1[x_c][y_c]= func((x_R[x_c], x_R[y_c]),
 ↪popt2_R[0],popt2_R[1],popt2_R[2])
```

```
[ ]: #testing intermediary operations to see where the error lies
#plt.imshow(image_test1)
plt.figure(figsize=(10,10))
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

plt.imshow((image_test1), cmap='rainbow')
fig = plt.gcf()
```

```
fig.set_facecolor('white')
ax = plt.gca()
```

[ ]:
```
plt.figure(figsize=(10,10))
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)

plt.imshow((Z2_R), cmap='rainbow')
fig = plt.gcf()
fig.set_facecolor('white')
ax = plt.gca()
```

[ ]:
```
#constructing a dictionary where to store the optimal values for gx1,
 ↪gx2, gy1,gy2,c1,c2
square_length = 500
dictionary_square_length = 480
r_G = [[{1:0, 2:0, 3:0, 4: 0, 5:0, 6:0} for x in range(square_length)]
 ↪for y in range(square_length)]
```

[ ]:
```
#creating a for loop to compute the needed parameters for all the pixels
 ↪15 pixels away from the edges
#the same I attempted to do through using the minimisation function, but
 ↪that did not work either
for x_c in range (15, 486):
    for y_c in range (15, 486):
        coord_list=[x_c-10, x_c+10, y_c-10, y_c+10]
        xnew_R=np.linspace(coord_list[0], coord_list[1], coord_list[1]-
 ↪coord_list[0]+1, dtype=int)
        ynew_R=np.linspace(coord_list[2], coord_list[3], coord_list[3]-
 ↪coord_list[2]+1, dtype=int)
        XRnew,YRnew=np.meshgrid(xnew_R,ynew_R)
        dataR=np.vstack((XRnew.ravel(), YRnew.ravel()))
        Z1new_R = np.zeros(XRnew.shape)
        Z1new_R=angle_fourier_combined_iift[coord_list[0]:
 ↪coord_list[1]+1, coord_list[2]:coord_list[3]+1]
        Z2new_R = np.zeros(XR.shape)
        Z2new_R=angle_fourier_combined_new_iift[coord_list[0]:
 ↪coord_list[1]+1, coord_list[2]:coord_list[3]+1]


        pRnew_1=g_4_select[0],g_4_select[1], c1_test
        pRnew_2=g_3_select[0],g_3_select[1], c2_test

        poptim1_R, pcovnew1_R = curve_fit(func, dataR, Z1new_R.ravel(),
 ↪pRnew_1)
        poptim2_R, pcovnew2_R = curve_fit(func, dataR, Z2new_R.ravel(),
 ↪pRnew_2)
        #print(curve_fit(func, (x1_w, y1_w),z_1,p0_1))
        #print(curve_fit(func, (x1_w, y1_w),z_2,p0_2))
        r_G[x_c][y_c]={1:poptim1_R[0],2:poptim2_R[0], 3:poptim1_R[1], 4:
 ↪poptim2_R[1], 5:poptim1_R[2], 6:poptim2_R[2]}
```

[ ]:
```
#printing some test values
print(r_G[10][11])
print(r_G[12][13])
print(r_G[470][470])
```

```
[ ]: #making for each pixel a G matrix with the corresponding g-vectors
G_all_pixels = np.zeros((500,500,2,2))
#Defining the A matrix as the inverse of the G matrix, for every pixel of
 ↪the lattice
A_all_pixels=np.zeros((500,500,2,2))
A_all_pixels_plain=np.zeros((500,500,2,2))
for x_c in range (15, 486):
    for y_c in range (15, 486):
        G_all_pixels[x_c][y_c][0][0]=r_G[x_c][y_c][1]
        G_all_pixels[x_c][y_c][0][1]=r_G[x_c][y_c][2]
        G_all_pixels[x_c][y_c][1][0]=r_G[x_c][y_c][3]
        G_all_pixels[x_c][y_c][1][1]=r_G[x_c][y_c][4]
        A_all_pixels_plain[x_c][y_c]= np.linalg.
 ↪inv(G_all_pixels[x_c][y_c])
        A_all_pixels[x_c][y_c]=np.transpose(A_all_pixels_plain[x_c][y_c])



print(G_all_pixels[300][300])
print(A_all_pixels[300][300])        #next to be transposed!!!
```

```
[ ]: #defining the c vector containing the optimal c values, again for each
 ↪pixel of the lattice
c_all_pixels=np.zeros((500,500,2))
for x_c in range (15, 486):
    for y_c in range (15, 486):
        c_all_pixels[x_c][y_c][0]=r_G[x_c][y_c][5]
        c_all_pixels[x_c][y_c][1]=r_G[x_c][y_c][6]


print(c_all_pixels[20][30])
```

```
[ ]: #defining the displacement vector , the distortion, the strain, the
 ↪reference matrix A_O, G_O
u_E=np.zeros((500,500,2,2))
u_L=np.zeros((500,500,2,2))
e_E=np.zeros((500,500,2,2))
e_L=np.zeros((500,500,2,2))
eps=np.zeros((500,500,2,2))
R=np.zeros((500,500,2,2))

A_1_x=np.zeros((500,500))
G_1_x=np.zeros((500,500))
G_1_y=np.zeros((500,500))
A_1_y=np.zeros((500,500))
A_2_x=np.zeros((500,500))
A_2_y=np.zeros((500,500))
a_1_x=np.zeros((500,500))
a_1_y=np.zeros((500,500))
A_product_1=np.zeros((500,500,2,2))
A_product_2=np.zeros((500,500,2,2))
A_O=np.zeros((2,2))
c_O=np.zeros((2,2))
```

```
x_O=(50+150)//2
print(x_O)

y_O=(50+150)//2
print(y_O)

A_O=np.linalg.inv(G_O)
c_O=c_all_pixels[x_O][y_O]

for x_c in range (15, 486):
    for y_c in range (15, 486):

        A_1_x[x_c][y_c]=A_all_pixels[x_c][y_c][0][0]
        A_1_y[x_c][y_c]=A_all_pixels[x_c][y_c][1][0]
        A_2_x[x_c][y_c]=A_all_pixels[x_c][y_c][0][1]
        A_2_y[x_c][y_c]=A_all_pixels[x_c][y_c][1][1]
        G_1_y[x_c][y_c]=r_G[x_c][y_c][3]
        G_1_x[x_c][y_c]=r_G[x_c][y_c][1]
#plt.imshow(a_1_x, cmap="RdBu", vmin=-0.1, vmax=0.1)
#plt.colorbar(extend="max",cmap='RdBu')
#print (np.max(a_1_x))
#print (np.min(a_1_x))
#plt.imshow(A_O)
#print(A_O)
```

[ ]:
```
#plotting g1y for testing, must be zero left side and -0.1 right side
plt.figure(figsize=(10,10))

plt.imshow(G_1_y, cmap="RdBu", vmin=-0.1, vmax=0.1)
plt.colorbar(extend="max",cmap='RdBu')
print (np.max(G_1_y))
print (np.min(G_1_y))
```

[ ]:
```
plt.figure(figsize=(10,10))

plt.imshow(G_1_x, cmap="RdBu", vmin=-0.03, vmax=0.03)
plt.colorbar(extend="max",cmap='RdBu')
print (np.max(G_1_x))
print (np.min(G_1_x))
```

[ ]:
```
#plotting a1x for testing
plt.figure(figsize=(10,10))
plt.imshow(A_1_x, cmap="RdBu", vmin=-0.5, vmax=0.5)
plt.colorbar(extend="max",cmap='RdBu')
print (np.max(A_1_x))
print (np.min(A_1_x))
```

[ ]:
```
#plotting a1y for testing
plt.figure(figsize=(10,10))
plt.imshow(A_1_y, cmap="RdBu", vmin=-10, vmax=10)
plt.colorbar(extend="max",cmap='RdBu')
print (np.max(A_1_y))
print (np.min(A_1_y))
```

```
[ ]: #filling in the values for the rotation matrix, strain, displacement at
      ↪each point
     for x_c in range (15, 486):
         for y_c in range (15, 486):
             A_product_1[x_c][y_c]=np.dot(A_O, np.linalg.
      ↪inv(A_all_pixels[x_c][y_c]) )
             A_product_2[x_c][y_c]=np.dot(A_all_pixels[x_c][y_c],np.linalg.
      ↪inv(A_O))
             u_E[x_c][y_c]=np.dot((np.identity(2)-
     A_product_1[x_c][y_c]),[x_c,y_c])+ np.dot(A_O, (c_all_pixels[x_c][y_c]-
     c_O))
             #u_L[x_c][y_c]=np.dot(np.dot(A_product_2[x_c][y_c],np.
      ↪identity(2)),[x_O,y_O]+np.
      ↪dot(A_all_pixels[x_c][y_c],(c_all_pixels[x_c][y_c]-c_O)))
             e_E[x_c][y_c]=np.identity(2)-A_product_1[x_c][y_c]
             e_L[x_c][y_c]=A_product_2[x_c][y_c]-np.identity(2)
             eps[x_c][y_c]=(e_E[x_c][y_c]+np.transpose(e_E[x_c][y_c]))/2
             R[x_c][y_c]=np.identity(2)+ 0.5*(e_L[x_c][y_c]-(e_L[x_c][y_c]).T)
```

```
[ ]: #plotting a2x
     plt.imshow(A_2_x, cmap="RdBu", vmin=-344, vmax=344)
     plt.colorbar(extend="max",cmap='RdBu')
     print (np.max(A_2_x))
     print (np.min(A_2_x))
```

```
[ ]: #plotting a2y
     plt.imshow(A_2_y, cmap="RdBu", vmin=-1, vmax=1)
     plt.colorbar(extend="max",cmap='RdBu')
     print (np.max(A_2_y))
     print (np.min(A_2_y))
```

```
[ ]: #testing part of the code to see whether they work to find the bug
     np.dot((np.identity(2)-A_product_1[x_c][y_c]),(np.array([x_c,y_c])))[1]
```

```
[ ]: #testing again
     np.dot(A_O, (c_all_pixels[x_c][y_c]-c_O))[0]
```

```
[ ]: #testing some values of displacement
     print(u_E[25][34])
```

```
[ ]: #displacement=np.zeros((500,500))
```

```
[ ]: #plotting components of strain matrix (which is 2x2 but each term in
      ↪itself is a matrix)
     displacement=np.zeros((500,500))
     e_xx=np.zeros((500,500))
     e_xy=np.zeros((500,500))
     e_yx=np.zeros((500,500))
     e_yy=np.zeros((500,500))
     R_yx=np.zeros((500,500))
     for x_c in range (15, 486):
         for y_c in range (15, 486):
             e_xx[x_c][y_c]=eps[x_c][y_c][0][0]
```

```
        e_xy[x_c][y_c]=eps[x_c][y_c][1][0]
        e_yx[x_c][y_c]=eps[x_c][y_c][0][1]
        e_yy[x_c][y_c]=eps[x_c][y_c][1][1]
        R_yx[x_c][y_c]=R[x_c][y_c][0][1]
```

[ ]:
```
#plotting e_xx
plt.imshow(e_xx, cmap='RdBu', vmin=-2.3, vmax=2.3)

#print(count(np.min(e_xx)))
plt.colorbar(extend="max",cmap='RdBu')
print(np.max(e_xx))
print(np.min(e_xx))
```

[ ]:
```
##plotting R_yx
plt.imshow(R_yx, cmap='RdBu', vmin=-0.3, vmax=0.3)

#print(count(np.min(e_xx)))
plt.colorbar(extend="max",cmap='RdBu')
print(np.max(R_yx))
print(np.min(R_yx))
```

[ ]:
```
#plotting e_xy
plt.imshow(e_xy, cmap='RdBu', vmin=-1, vmax=1)
print(e_xy)
```

[ ]:
```
#plotting e_yx
plt.imshow(e_yx)
```

[ ]:
```
##plotting e_yy
plt.imshow(e_yy)
```

[ ]:
```
#checking some strain values
print(np.max(e_yy))
print(np.min(e_yy))
```

Bibliography

[1] N. David Mermin A. Neil W. Ashcroft, ed. *Solid State Physics*. 1976.

[2] S. Hamid Nawab Alan V. Oppenheim Alan S. Willsky, ed. *Systems Signals*. Prentice-Hall, Inc, 1997, pp. xxx+948. ISBN: 0-13-814757-4.

[3] J. Cowley L. Eyring P. Buseck, ed. *High-Resolution Transmission Electron Microscopy and Associated Technique*. Oxford University Press, 1989. ISBN: 0-19-504275-1.

[4] W. Chang. “Determination of strain fields on two-dimensional images using the STC method”. In: *Computational Materials Science* (2019). DOI: 10.1103/PhysRevB.99.161117.

[5] S. Conesa-Boj. “Plastic and Elastic Strain Fields in GaAs/Si Core−Shell Nanowires”. In: *NANO letters* (2014). DOI: 10.1021/nl4046312.

[6] O. M. Essenwanger. *Elements of statistical analysis*. Elsevier, 1986.

[7] Pedro L. Galindo. “Strain mapping from HRTEM images”. In: ().

[8] Pedro L. Galindo. “The Peak Pairs algorithm for strain mapping from HRTEM images”. In: ().

[9] D. C. Gijswijt, ed. *Algebra 1*. Delft University of Technology, 2021.

[10] van Gog. “Thermal stability and electronic and magnetic properties of atomically thin 2D transition metal oxides”. In: *npj 2d materials and applications* (2019). DOI: 10.1038/s41699-019-0100-z.

[11] Jeff Hecht, ed. *Understanding Lasers: An Entry Level Guide*. 2018.

[12] Hu and Hau. “S4 Symmetric Microscopic Model for Iron-Based Superconductors”. In: *Physical review* (2012). DOI: 10.1103/PhysRevX.2.021009. URL: https://doi.org/10.1103/PhysRevX.2.021009.

[13] M.J. Hÿtch. “Quantitative measurement of displacement and strain fields from HREM micrographs”. In: *Ultramicroscopy* (1998). DOI: 10.1016/S0304-3991(98)00035-7.

[14] et al. de Jong T.A. “Imaging moiré deformation and dynamics in twisted bilayer graphene.” In: *Nature Communication* 13.70 (2022). DOI: doi.org/10.1038/s41467-021-27646-1. URL: https://doi.org/10.1038/s41467-021-27646-1.

[15] Nayak. “Rigid-band electronic structure of scandium nitride across the n-type to p-type carrier transition regime”. In: *Physical review* (2019). DOI: 10.1103/PhysRevB.99.161117.

[16] *Optimize.CurveFit*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html.

[17] *Optimize.Minimize*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html?highlight=minimize.

[18] J.L. Rouvière and E. Sarigiannidou. “Theoretical discussions on the geometrical phase analysis”. In: *Ultramicroscopy* 106.1 (2005), pp. 1–17. ISSN: 0304-3991. DOI: https://doi.org/10.1016/j.ultramic.2005.06.001. URL: https://www.sciencedirect.com/science/article/pii/S0304399105001038.

[19] Steven H. Simon. *The Oxford Solid States Basics*. First. Oxford University Press, 2013, pp. xiii+290. ISBN: 978–0–19–968077–1.

[20] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[21] A. K. Zak. “X-ray analysis of ZnO nanoparticles by Williamson-Hall and size-strain plot methods”. In: *Solid State Science* (2019). DOI: 10.1038/s41699-019-0100-z.