NTNU

Norwegian University of
Science and Technology

# Developing Quantitative Image Processing of Scanning Electron Microscopy Data Sets to Evaluate Nanowire Growth

**Steinar Myklebost**

Nanotechnology
Submission date: July 2017
Supervisor: Antonius Theodorus Johann Van Helvoort, IFY

Norwegian University of Science and Technology
Department of Physics

# Abstract

Semiconductor nanowires have promising future applications in optical and electronic components. Automated computer routines are developed to accurately, objectively, and efficiently evaluate methods for initiating local nanowire growth. The routines are created using Python and open source libraries, and are able to detect nanowire catalyst droplets on scanning electron microscopy (SEM) images, and produce data about yield, droplet diameter and nanowire displacement from an ideal lattice.

The developed routines are employed to analyze SEM images of three different nanowire samples, all containing self-catalyzed GaAsSb nanowires grown using molecular beam epitaxy (MBE). The first case shows a set of nanowire arrays patterned using a focused ion beam (FIB) with varying patterning diameter and ion fluence, in order to efficiently study the effect these parameters have on nanowire growth. The second case consists of a large area analyzed using stitched SEM imaging. In this case nanoimprint lithography was used to pattern a mask prior to MBE nanowire growth. Images of regions larger than 0.047 $mm^2$ are assembled by stitching multiple adjacently acquired images. Studying a large amount of nanowires (> 50 000) enables the acquisition of highly detailed data. Finally, the nanowire density and droplet size is analyzed for randomly positioned nanowires initiated by FIB.

For the FIB patterned arrays, patterning with lower ion fluence and patterning diameter results in better single nanowire yield (up to 84 %), and higher placement uniformity. Higher fluence and patterning diameter results in the growth of 2D-crystals or multiple nanowires per hole. Displacement analysis shows that nanowires tend to nucleate along edges of patterned holes. Analysis of the NIL sample show that large numbers of nanowires can be analyzed. Imaging and image stitching can have severe effects on accuracy of the analysis. Nanowires near empty regions have larger catalyst droplets, and nanowires with other growth at the same hole tend to be displaced with threefold symmetry. For both samples, it is shown that single nanowires have larger catalyst droplets with higher size uniformity. Analysis of the random growth dataset shows that the low effective fluence experienced outside of areas directly patterned by FIB gives good conditions for high density non-position controlled nanowire growth.

# Sammendrag

Nanotråder av halvledermateriale har lovende potensiale for bruk i optiske og elektroniske komponenter. Automatiserte databehandlingsrutiner utvikles for å evaluere prosesser for initiering av nanotråd-vekst på en nøyaktig, objektiv og effektiv måte. Rutinene er laget med språket Python og støttebiblioteker med åpen kildekode, og er i stand til å detektere katalysatordråper på nanotråder på bilder tatt i sveipeelektronmikroskop (SEM), og produsere data om utbytte, dråpediameter og nanotrådenes forskyvning fra et ideelt gitter.

De utviklede rutinene blir brukt til å analysere SEM-bilder av tre ulike prøver, som alle inneholder selvkatalyserte GaAsSb-nanotråder grodd med molekylstråleepitaksi (MBE). Det første tilfellet viser et sett med nanotråd-arrays mønstret med en fokusert ionestråle (FIB) med varierende diameter og ion-fluens, for effektivt å studere virkningen disse parameterne har på nanotråd-veksten. Det andre tilfellet består av et stort område som er analysert ved hjelp av sammenføyde SEM-bilder. I dette tilfellet ble nanoimprint-litografi brukt til å mønstre en maske før nanotråder ble grodd med MBE. Bilder av områder større enn 0,047 $mm^2$ skapes ved å føye sammen flere overlappende bilder. Å studere en stor mengde nanotråder (> 50 000) gjør det mulig å skaffe seg svært detaljerte data. Til slutt analyseres tettheten av nanotråder og dråpestørrelsen for tilfeldig plasserte nanotråder initiert av FIB.

For FIB-mønstrede arrays resulterer mønster med lavere ion-fluens og mønstringsdiameter i en høyere andel av vellykkede enkeltnanotråder (opptil 84 %) og mer regulær posisjonering. Høyere fluens og mønstringsdiameter fører til vekst av 2D-krystaller eller flere nanotråder per hull. Forskyvningsanalyse viser at nanotråder har en tendens til å nukleere langs kantene av mønstrede hull. Analyse av NIL-prøven viser at det er mulig å analysere et stort antall nanotråder. Kvaliteten på bildetaking og -sammenføying kan ha alvorlig innvirkning på nøyaktigheten av analysen. Nanotråder i nærheten av tomme områder har større katalysatordråper, og nanotråder med annen vekst i samme hull blir ofte trisymmetrisk forskjøvet. For begge prøvene vises det at enkeltstående nanotråder har større kata-lysatordråper med mer regulær størrelse. Analyse av de ikke-posisjonerte nanotrådene viser at den lave effektive fluensen som oppnås utenfor områder direkte mønstret av FIB, gir gode betingelser for ikke-posisjonert nanotråd-vekst med høy tetthet.

# Preface

The work presented in this Master's thesis has been done at the Norwegian University of Science and Technology (NTNU) during the spring of 2017. The project has been supervised by Professor Antonius T. J. van Helvoort, and co-supervised by Aleksander Mosberg. All FIB patterning described in this work, and acquisition of all SEM images used, has been done by Aleksander Mosberg. NIL patterning and MBE nanowire growth has been done by Dingding Ren. The work done in this Master's thesis is a continuation of the work for my project thesis, performed during the fall of 2016.

The work done in this Master's thesis has been presented in a poster shown at Nanowire week 29 May-2 June, Lund, Sweden and EMAG, 3-6 July, Manchester, UK, and in a conference paper for the EMAG conference, which will be published in *Journal of Physics: conference series*. Both of these works are presented as appendices to this thesis.

I would like to thank everyone involved in the project, especially Prof. van Helvoort, who has been an excellent supervisor, always doing his best to help me get the most out of my Master's thesis, through frequent follow-up meetings and thorough feedback. I would like to thank Aleksander Mosberg for contributing with ideas, help, and support. I would like to thank my mother, for supporting me through tough times. Finally, I would like to thank all the fantastic people of Timini, for making my years as a university student an amazing experience.

Steinar Myklebost

# List of Acronyms

**BSE** backscattered electrons

**CVD** chemical vapor deposition

**FIB** focused ion beam

**LMIS** liquid metal ion source

**MBE** molecular beam epitaxy

**MOCVD** metalorganic chemical vapor deposition

**px** pixel(s)

**SE** secondary electrons

**SEM** scanning electron microscope

**VLS** vapor liquid solid

# Contents

# Chapter 1

# Introduction and motivation

Semiconductor nanowires have the potential to be useful for many different applications due to their optical and electronic properties. Among the possible applications are use in sensors, lasers, transistors, and solar cells [1, 2]. The viability of usage of nanowires for these applications depends on the ability to develop processes to reliably synthesize large amounts of nanowires with the desired properties. To optimize nanowire properties, it is necessary to accurately control their size, position, morphology and composition.

In order to develop these processes, it is necessary to analyze the quality of nanowires grown using different methods, and determine the effects of varying process parameters on nanowire properties. The counting and measurement necessary for such analysis is commonly done manually [3], and thus only for a small number of nanowires, even though samples contain thousands or even millions of nanowires. This leads to the collected data being inaccurate, as it is based on a small sample size, and limited by the restrictions of manual measurement. The collected data is also subject to human bias, as the researcher manually selects the nanowires to be analyzed. Improved methods for analyzing nanowire growth would thus be helpful for the development of good nanowire growth processes.

This thesis aims to develop automated computer routines to objectively and efficiently analyze micrographs of nanowire growth, and extract useful data. Feature detection techniques

are employed to detect position and size of nanowire catalyst droplets, and the the resulting data is processed and analyzed to provide useful insights about nanowire growth. To demonstrate their viability, the developed routines are deployed for three different tasks. All tasks consist of studying self catalyzed GaAsSb nanowires grown using molecular beam epitaxy (MBE) on a patterned substrate consisting of Si covered by a $SiO_x$ thin film. The differences lie in the patterning of the samples, and the purpose of the studies.

Firstly, a matrix of $8 \times 8$ nanowire growth arrays patterned using a focused ion beam (FIB) is studied. The FIB has been used to pattern holes in the substrate using varying ion fluence and patterning diameter for each array. Nanowires growing at each array is analyzed in order to efficiently gain an overview of how the two patterning parameters affect nanowire growth.

Secondly, images of large areas of a nanowire growth sample patterned using nanoimprint lithography (NIL) are studied. The images have been created by stitching together adjacently acquired scanning electron microscopy (SEM) images. As the combined images cover areas of up to 47 000 $\mu m^2$, containing more than 50 000 nanowires, large amounts of data can be collected, enabling high statistical accuracy.

Lastly, another area of the aforementioned FIB patterned sample is analyzed. This part of the sample contains eight patterned squares, showing non-position controlled nanowire growth inside the squares, and in an area surrounding the squares. Images of this area is analyzed in order to gain insight into the conditions for of non-position controlled nanowire growth.

The developed routines are written in the Python programming language, utilizing publicly available open source libraries. The created code is made available online as an open source repository. This enables other researchers to utilize the routines in their own projects, inspect the source code, and undertake further development of the project. As opposed to commonly used proprietary black-box software packages, open source code ensures the transparency of methods used in research, enabling researchers to know exactly how any result is found, increasing the scientific integrity of their work.

# Chapter 2

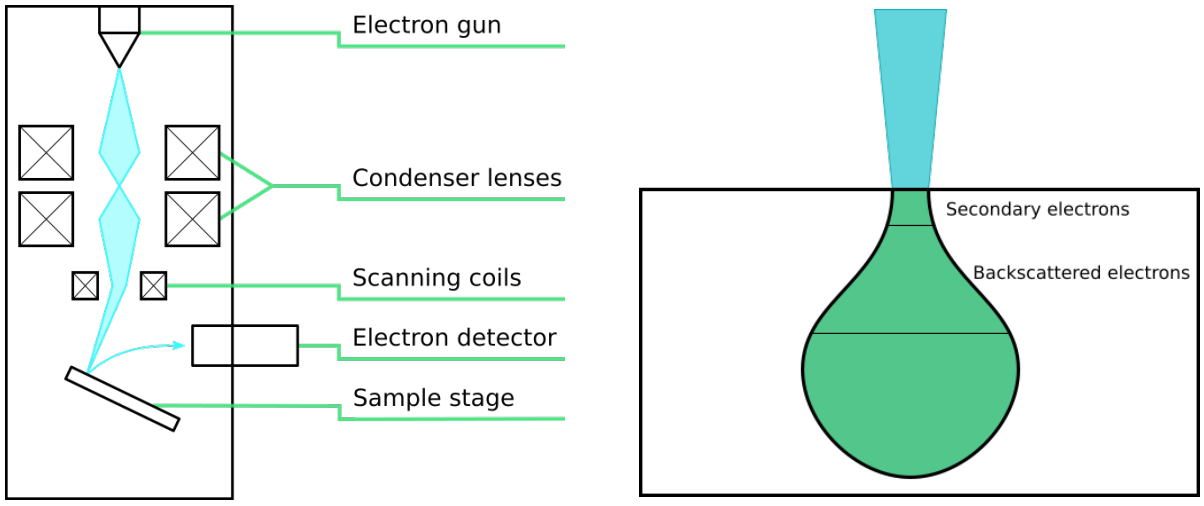
# Theory

## 2.1 Fabrication and characterization methods

### 2.1.1 Scanning electron microscope (SEM)

The scanning electron microscope (SEM) [4] is a microscope that creates an image of the specimen by raster scanning an electron beam across its surface, causing the emission of electrons from the sample. The amount and energy of the electrons emitted varies with the properties of the sample. The amount of detected electrons when the electron beam is hitting a certain point on the sample is the source of the contrast in the obtained images.

The layout of a typical SEM column is shown in figure 2.1(a). Electrons are emitted by an electron gun, which might utilize the principles of thermionic emission, field emission or Schottky emission. The beam then passes through a column, where it is focused by a series of magnetic lenses, and then deflected towards the desired spot on the sample by scanning coils. There are normally two sets of scanning coils, for deflecting the beam in the $x$- and $y$-directions. The voltage applied to the coils follows a sawtooth pattern, with one coil having a longer period, and the other a shorter period. This results in a raster pattern which scans across every part of the image.




(a) (b)

Figure 2.1: (a) Schematic of an SEM column, showing the most important components. (b) Diagram showing the interaction volume in an SEM sample. SE will only escape from the topmost part of the interaction volume, while the BSE might originate from somewhat deeper in the sample.

When the electron beam hits the sample, the electrons will penetrate into the sample, and interact with it in various ways. The volume of the sample with which most of the incoming electrons interact is called the interaction volume. The typical shape of the interaction volume is shown in figure 2.1(b). Near the surface of the sample, the width of the interaction volume is similar to that of the incoming electron beam. As the electrons penetrate deeper into the sample, they have a greater chance of scattering on nuclei or other electrons, redirecting their momentum, and thus the interaction volume widens. The overall size of the interaction volume depends on the energy of the incoming electrons, and the composition of the sample. Higher energy electrons will penetrate deeper, and have a larger interaction volume. Larger atoms in the sample will lead to more frequent scattering of the electrons, reducing the penetration depth, and leading to a smaller interaction volume.

### Types of emissions

As the incoming electrons interact with the sample, they will cause the emission of secondary electrons (SE), backscattered electrons (BSE), auger electrons, and x-rays. The latter two will not be further explained in this text.

When electrons in the sample are hit by the incoming electron beam, they might gain enough energy to escape from the sample. These are called secondary electrons (SE). Each incoming electron can interact with several electrons in the sample, causing more than one SE emission. The energy of the SE is normally less than 100 eV. Due to this low energy, SE generated deep within the sample will lose their energy to scattering, and will not escape from the sample. Only SE generated within the upper few nm of the sample will escape, and be detected.

Electrons from the incoming beam may also elastically scatter on the nuclei in the sample. A portion of these electrons will escape the sample, and can be detected. These are called backscattered electrons (BSE). Since they are elastically scattered, the BSE lose very little energy, so the energy of the BSE is approximately equal to the energy of the incoming beam, which is typically several keV. The BSE thus have a larger escape depth than the SE, and will provide information about a deeper section of the sample. Since the width of interaction volume increases with depth, BSE also originate from a wider region, meaning that BSE images have a lower spatial resolution than SE images. BSE are usually less abundant than the SE, and constitute a smaller part of the overall signal.

### SE contrast

Images obtained by the detection of SE mostly exhibit topological contrast. This is due to two factors. Firstly, the volume of the sample emitting detectable SE (the volume both exposed to the electron beam, and within escape depth of the surface), will depend on the local topology of the sample. This is illustrated in figure 2.2. Since the depth of the volume emitting detectable SE is determined by the shortest distance to the sample surface, which is not necessarily along the same axis as the incoming beam, this volume will vary in size with the local

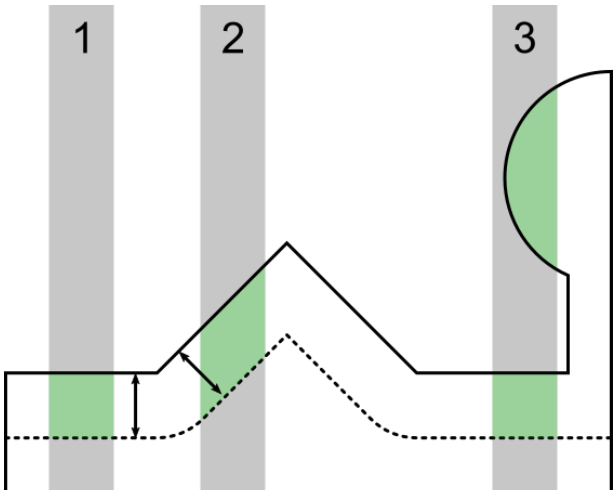



Figure 2.2: Interaction volumes for emission of SE on different topologies. Beam 1 hits a flat surface, and has the smallest interaction volume. Beam 2 has a larger interaction volume due to the angled surface, and the fact that the lower limit of the interaction volume is determined by the shortest distance to the sample surface. Beam 3 meets an overhanging structure, and has an even larger interaction volume.

topology. Surfaces which are not normal to the incoming beam will thus have a larger volume, and more SE will be emitted. Overhanging structures will lead to even larger emissions of SE.

SE images are also subject to a shadowing effect. If there are features between the site of emission of the SE and the electron detector, a portion of the SE will collide with these features, and less electrons will reach the detector. If the detector is placed at an angle, this leads to protruding features obscuring the areas behind them, creating what looks like shadows on the images. These two effects lead to a topological contrast that gives the observer an intuitive understanding of the sample's 3D geometry.

**BSE contrast**

BSE exhibit different kinds of contrast than SE. The larger escape depth and wider interaction volume of the BSE means that they are much less subject to the kinds of topological contrast that affects SE. The amount of BSE emitted is however highly dependent on the atomic num-

ber of the atoms in the sample. Larger nuclei are more likely to scatter the incoming electrons, leading to a larger amount of BSE. The BSE signal thus shows compositional contrast.

The BSE signal is also subject to an effect called channeling. The penetration depth of the BSE is affected by the crystallographic orientation of the sample relative to the incoming electrons. This leads to a crystallographic contrast, where differing crystal structure or grain orientation leads to differences in brightness.

#### Artifacts and distortions

If a sample examined with SEM is not conductive, it will build up a negative charge when hit by the electron beam. If the sample becomes negative enough, it will deflect the incoming electron beam. This will lead to an effect called charging, where the image becomes distorted or obscured. SEM images can also be distorted due to vibrations or thermal drift, although this is not a common problem.

### 2.1.2 Focused ion beam (FIB)

The focused ion beam (FIB) [5] is an instrument that creates a focused beam of ions which can be used to either image or pattern a sample. The ions are generated by a liquid metal ion source (LMIS), consisting of a tungsten tip supplied with liquid metal, often gallium. The gallium wets the tungsten tip, and an extractor electrode applies a strong electric field. This ionizes the gallium, and extracts and accelerates the ions. The beam of ions is focused by a series of lenses and apertures, and directed by deflection coils. The beam can be raster scanned across the sample, or scanned in a user determined pattern. In many modern FIB systems, it is common to have an electron column in addition to the ion column, as illustrated in figure 2.3(a). This allows for less destructive SEM imaging along with FIB milling. Such an instrument is called a dual-beam or FIB-SEM system.

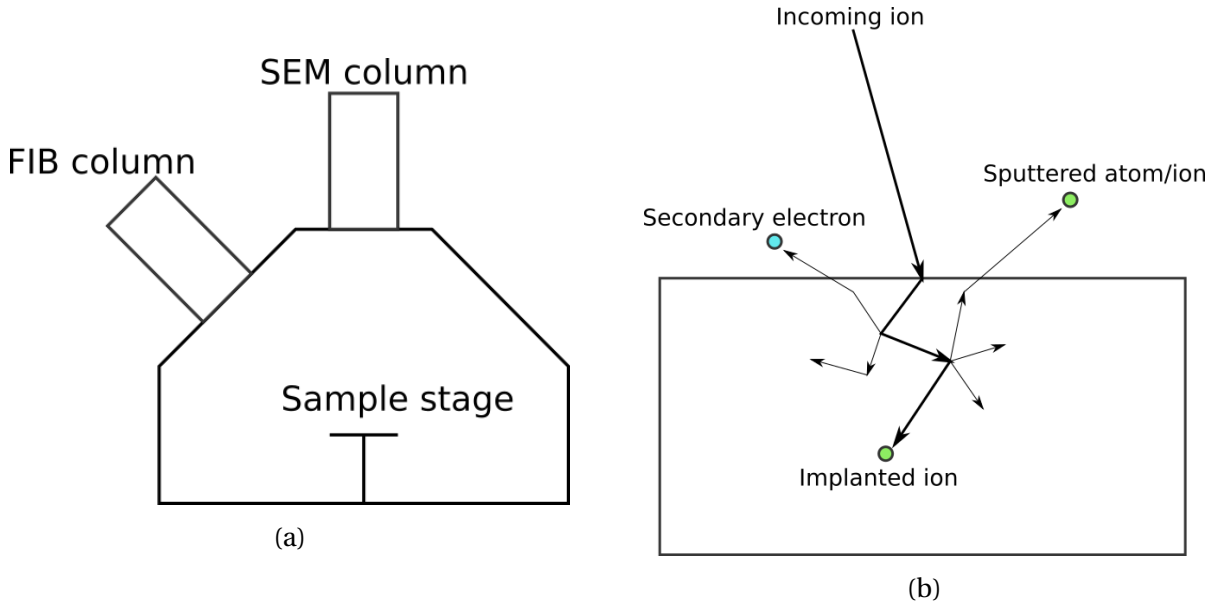



Figure 2.3: (a) Schematic showing the setup of a dual-beam FIB-SEM system (b) Illustration of the collision cascade

As the ion beam hits the sample, the ions will collide with the atoms in the sample. These collisions will transfer energy to the atoms, and might cause them to escape from their position in the sample material, and collide with further atoms. This process is called a collision cascade, and is illustrated in figure 2.3(b). If the collision cascade leads to atoms near the surface gaining enough energy to escape from the sample, they will be sputtered away. The collision might also ionize the atoms, causing them to leave the sample as ions. Electrons in the topmost layer of the sample may also be scattered by the collision cascade, and escape the sample.

Since the collision cascade move atoms away from their place, the material might become amorphous. If the incoming ions are not scattered back out of the sample, they will be implanted. These effects apply to the topmost layer of the sample, down to the penetration depth of the ions. This local alteration of the sample's properties might cause the areas exposed to the beam to react differently to later processing, for instance by changing the etch rate.

### Imaging

The ions, atoms and electrons that are ejected from the sample by the collision cascade can be detected, and if the beam is raster scanned, this can be used to image the sample in much the same way as with an SEM. The mechanisms of contrast will be different for ion images and electron images, and thus these images provide complementary information.

### Milling

Since the collision cascade causes atoms to sputter from the surface, the FIB can be used for milling the sample. As the beam can be scanned in an arbitrary pattern, a wide variety of microstructures can be created. When milling, a much higher beam current is typically used, than when imaging. The amount of milled material scales linearly with the amount of incoming ions. A desired depth of milled features is achieved by exposing the areas to be milled to a given fluence. This can be achieved either by leaving the beam over the areas for a certain amount of time, or by performing multiple passes over the pattern until the desired fluence has been reached.

Patterns milled with the FIB can have a spatial resolution down to 10 nm. The pattern will however not be perfectly sharp, as the beam has a Gaussian profile. This means that any sidewalls of milled features will be somewhat sloped. The area around milled features will also be lightly exposed to the ion beam, and might have its properties changed, even if it is not subject to significant milling.

The depth of milled features can be increased by increasing fluence, and their width can be increased by increasing the beam diameter. However, the Gaussian profile of the beam means that increasing fluence also will increase feature width somewhat. An increase in beam diameter, while keeping the fluence constant, will also increase the feature depth. Thus feature width and depth are not adjusted independently by simply varying either beam diameter or fluence.

### 2.1.3 Nanowire growth

One of the most commonly utilized methods for growing nanowires is the vapor-liquid-solid method (VLS) [6]. The method is named after the phases the source material goes through. The material is supplied in the vapor phase. It enters liquid catalyst droplets on the substrate surface, either directly, or by diffusing along the surface. As a droplet absorbs more and more of the source materials, it will eventually become saturated, and at this point, crystal growth will initialize at the droplet-substrate interface. As the growth continues, the droplet will be pushed along the top of the growing crystal. This leads to one-dimensional growth, which results in a nanowire structure.

The catalyst droplets used often consist of Au [6], but this can lead to the nanowire being contaminated with the Au [7]. For semiconductor nanowires, this might severely affect their electronic properties, and is thus undesirable. Contamination can be avoided by the utilization of self-catalyzed VLS growth, where the catalyst is the same as one of the growth species, for instance using Ga as the liquid catalyst when growing GaAs nanowires [8].

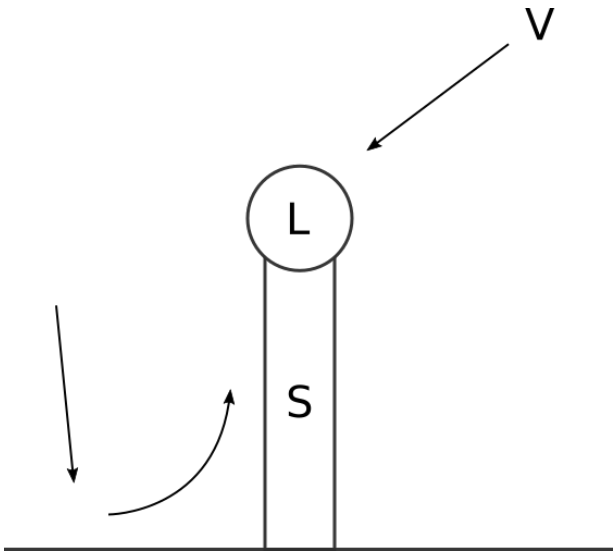



Figure 2.4: Illustration of VLS-growth. Vapor phase precursor material is adsorbed into the liquid phase catalyst droplet either directly or through diffusion along the substrate and wire. The solid nanowire grows beneath the droplet, pushing it upwards as it grows.

The source materials in the vapor phase can be supplied in several different ways, one of which is by molecular beam epitaxy (MBE) [9]. In an MBE setup, the sample is placed in

a chamber with ultra-high vacuum, and the material to be deposited is supplied in vapor phase by effusion cells connected to the chamber. The sample is often rotated to avoid shadowing and other directional effects. The MBE instrument allows for precise control of process parameters and deposition rates.

When growing nanowires through VLS, the positions and sizes of the nanowires can be controlled through control of the catalyst droplets on the growth substrate [10]. The position and size of the droplets determine the position and size of the nanowires grown. The droplets can be positioned by patterning a substrate so that droplets will only form in certain locations. For instance, a substrate of Si can be masked with a layer of $SiO_x$. Droplets will not form on the oxide, and thus nanowire growth will be limited to the areas where the Si is exposed.

## 2.2 Digital image processing and computer vision

Computer vision [11] is the subject of processing digital images to extract information. The human brain is able to easily extract information from what we see. To replicate this in a computer, we need to implement a variety of methods and algorithms. If successful, the computer can analyze images in an automated fashion. This is useful in many fields, for instance in scientific research, to analyze images acquired.

Digital image processing [12] is simply the processing of images using a computer. Image processing has applications in many fields, and is among other things an important component of computer vision. A variety of functions and algorithm are employed to alter images in order to enhance desired features to facilitate the extraction of the desired information.

To discuss image processing, an understanding of the digital representation of images is needed. A digital image is represented by an array of pixels, where each pixel can have a single brightness value, for greyscale images, or a vector of intensity values for each color, and opacity if relevant. For a binary image, each pixel has a value of 0 or 1. In greyscale images, each pixel can have one of a range of integer values, typically ranging from 0 to 255.

### 2.2.1 Spatial filtering

Spatial filtering [12] is the process of performing an operation on an input image, which generates an output image based on some function of the pixels in the input image. Each pixel of the output image is defined by applying an operator to a neighborhood of pixels around the corresponding pixel in the input image. The neighborhood can have any shape desired, but is typically defined as a square region centered on the given pixel, with the simplest neighborhood being a $3 \times 3$ square of pixels. If an operation is performed with a neighborhood of $n \times n$ pixels, it is said to have a kernel size of $n$. Since the kernel must have a pixel in its center, the kernel size is always an odd integer. Any spatial filtering where the operation performed is a linear combination of the pixels in the input neighborhood is called linear spatial filtering. Spatial filtering where this is not the case is called non-linear spatial filtering.

#### Median filtering

Median filtering is a type of non-linear spatial filtering, where the output pixel is the median value of the pixels in the input neighborhood [12]. Median filtering is able to remove noise, while preserving sharp edges, as opposed to averaging filters like a simple box blur, or a Gaussian blur, which will blur any sharp edges [13]. This makes median filtering useful for preparing images for feature detection [14].

Median filtering does however affect the shape of features in the image [14], and this must be considered when using median filtering as a preprocessing step for feature detection. While straight edges are not distorted by median filtering, any corners will be rounded, and features smaller than the kernel size might disappear entirely. Circular features are less affected by this, as they have no corners, but their size might be reduced. This effect is most pronounced when the radius of the circle is small compared to the median filter kernel size. If the kernel size used is much smaller than the circle radius, the size reduction is small or zero [15].

#### Erosion and dilation

Erosion and dilation [12] are the two primitive operations of morphological image processing. They can be defined in several equivalent ways, for instance as a non-linear spatial filtering operation, where the value of every pixel is replaced with the lowest (erosion) or highest (dilation) value of the pixels in its neighborhood.

#### Morphological reconstruction by erosion as a hole filling algorithm

Morphological reconstruction [12] is the process of repeatedly performing erosion or dilation on one image, the *seed*, limited by another image of the same dimensions, the *mask*. Morphological reconstruction by erosion can be used to "fill holes" in an image, that is, to flatten out any dips or valleys in the brightness function not connected to the edge of the image. In this case, the mask used is a copy of the original image, while the seed image consists of a 1 px thin edge copied from the original image, with the rest of the image set to the highest brightness value found in the original image. Erosion is performed repeatedly, with a cross shaped kernel with a width of 3 px. This causes darker values along the edge to "spread" and fill in the rest of the image, but since the mask is a copy of the original image, no pixel will obtain a value lower than in the original image. Any local minima in the brightness function will be flattened out due to the mask image setting a lower limit on the brightness value obtainable by erosion to that of its neighboring pixels.

### 2.2.2 Feature detection

Feature detection [11] is a process within computer vision where features, or points of interest, are located within an image. For more advanced computer vision tasks, such as recognition of complex objects like buildings or faces, feature detection is one of the low-level steps, giving information to be used further in a larger algorithm. When the objects to be recog-

nized are simple, such as having a well defined shape or brightness profile, simple feature detection might be the only step needed to locate the desired objects.

### 2.2.3 Image segmentation

Image segmentation [11] is the process of segmenting the pixels of an image into regions, where each region consists of a set of adjacent pixels related in some way. This might be as simple as a similarity in brightness or color, or more complex, like looking more like a building or an animal. One of the simplest forms of image segmentation is thresholding, where a binary image is generated, by evaluating each pixel according to some condition, e.g. intensity less than a given value, and setting each pixel to 1 if it meets the criterion or 0 otherwise. The image is then segmented by grouping together adjacent pixels of the same value.

## 2.3 Overview of software methods

The routines developed in this work are written in the Python programming language, utilizing several pre-made methods from open source libraries to perform image processing and other tasks. This section gives an overview of the most important methods used, and explains what they do. The methods described come from three different libraries.

**SciPy** [16] is a library that implements a variety of functions, algorithms and data structures useful for scientific computing.

**scikit-image** [17] is an independently developed add-on package for SciPy, implementing a wide range of image processing algorithms.

**OpenCV** [18] (also referred to as cv2) is an open source computer vision library written in C++, but with bindings for Python.

Following is the list of methods:

`scipy.optimize.minimize`

This function seeks to find the input values which minimize the output of a given function. The function to minimize can be any function defined in the program. The `minimize` function finds the minimum of the given function numerically, using one of several numerical minimization methods. One of the choices for minimization method is the Nelder-Mead method [19], which while not in all cases performing as well as alternate methods, provides robust results in many applications.

`scipy.signal.medfilt2d`

Applies a median filter to the given image, with a given kernel size

`scipy.spatial.KDTree`

A data structure for storing coordinates of a set of points, which allows for efficient lookup of the nearest points to any given point

`skimage.morphology.reconstruction`

Performs morphological reconstruction using given seed and mask images, by either erosion or dilation as specified by input

`cv2.simpleBlobDetector`

A class implementing an algorithm for detecting features in an image. Further details are given in section 2.3.1.

### 2.3.1 The `cv2.simpleBlobDetector` class

The `simpleBlobDetector` class within the OpenCV library is a class that deals with the setup and execution of a simple feature detection algorithm. The feature detection is performed by segmenting the image using a range of thresholds, and then accepting or rejecting each segment based on a set of criteria. The segments meeting the criteria are in the end returned

as a list giving the center coordinates and size of each of the accepted segments, from now on referred to as *blobs*.

By default, the algorithm detects dark features surrounded by brighter pixels. By inverting the image before detection, features that were bright in the original non-inverted image, can be detected instead. The range of values used for thresholding is by default the entire brightness value range, from 0 to 255, but can be limited by setting a minimum and maximum threshold. The criteria for acceptance of each segment can also be manually adjusted, setting minimum and/or maximum limits for a set of properties. The available criteria are as follows:

**Area** The number of pixels in the blob

**Circularity** A measure for how closely the segment resembles a perfect circle, based on the ratio of its area to its circumference. A perfect circle has a circularity of 1.

**Convexity** The ratio of the area of the blob and the area of its convex hull

**Inertia ratio** A measurement of how elongated the shape of the segment is. Defined as the ratio between the moments of inertia about its minor and major axes.

# Chapter 3

# Experimental

This chapter presents the three cases studied using the image processing routines developed in this thesis. The first section describes the FIB patterned sample, containing the matrix of nanowire growth arrays. The second section describes the NIL patterned sample, and the two sets of tiled images showing large regions of this sample. The last section describes the area of the FIB patterned sample containing non-positioned controlled nanowire growth, hereby referred to as the random growth sample. As this region is part of the same sample as the FIB milled matrix of arrays, some of the sample preparation is described in the first section.

Each section details the preparation of the sample, the acquisition and properties of the images comprising each dataset, any further processing of the images, the computer vision based feature detection, and what kinds of data was obtained about each dataset.

## 3.1 FIB arrays

### 3.1.1 Sample

A (111) p-doped Si wafer cut 5° off axis, covered by a 40 nm thick $SiO_2$ film, was patterned using a FEI Helios 600 NanoLab DualBeam focused ion beam system. The ions used were $Ga^+$

ions accelerated over a voltage of 30 kV. Figure 3.1 shows a schematic of the milling pattern. The sample has several different features, two of which were studied in this work.

The first feature of interest is a matrix of $8 \times 8$ nanowire growth arrays, represented on the schematic as white squares in the top right part of the sample. Each array is patterned with a hexagonal lattice of $15 \times 18$ holes with a 1 µm pitch. The arrays are numbered for reference, in the manner shown on the figure. The ion fluence and diameter used to pattern the holes were varied for each array, increasing as shown by the arrows in the schematic. The applied ion fluence was increased linearly from 0.06 nC/µm$^2$ for the bottom row, to 0.53 nC/µm$^2$ for the top row (equivalent to 10 – 100 nm deep milling in standard Si, according to the FIB control software). The milled diameter for each hole was increased linearly from 10 nm for the leftmost column, to 80 nm for the rightmost column.

The second feature of interest is the area around the eight gray squares labeled as "Random Growth" on the schematic. Although this feature is part of the same sample as the matrix of arrays, it differs significantly in its nature, and will be treated separately for the rest of this thesis. The study of this feature is described further in section 3.3.

After patterning, the sample was etched in a 1 % HF solution for 2.5 minutes. $GaAs_{x-1}Sb_x$ nanowires were grown on the sample by self-catalyzed VLS in a Varian GEN II Modular MBE system. The growth parameters are given in table 3.1. Figure 3.2 shows the sample after nanowire growth.

Table 3.1: MBE growth parameters for FIB patterned sample (ML = monolayers)

| | |
|---|---|
| **Ga flux** | 0.02 ML/s |
| **$As_4$ flux** | $5.4 \cdot 10^{-6}$ mbar |
| **$Sb_4$ flux** | $1 \cdot 10^{-6}$ mbar |
| **Growth time** | 50 min |
| **Temperature** | 640 °C |

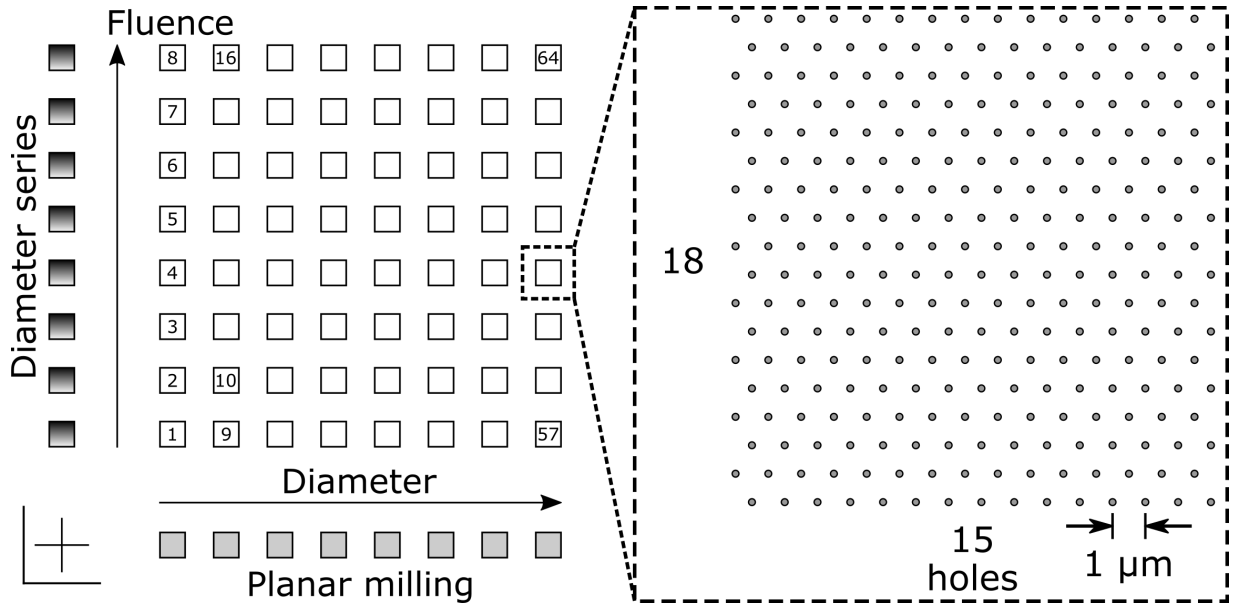



Figure 3.1: Schematic of the sample layout.

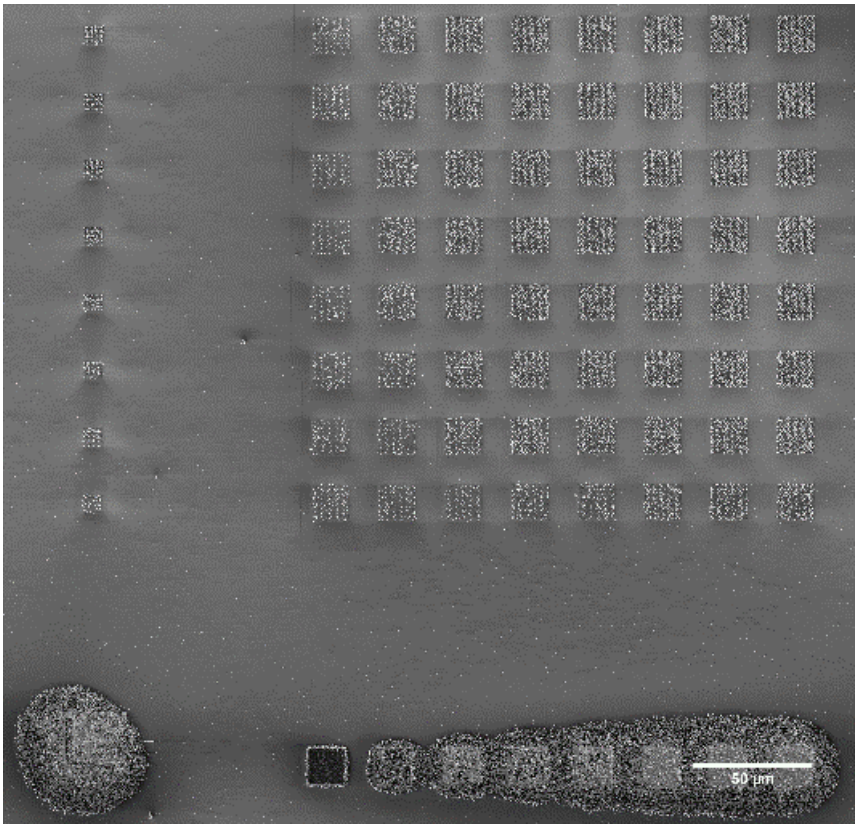
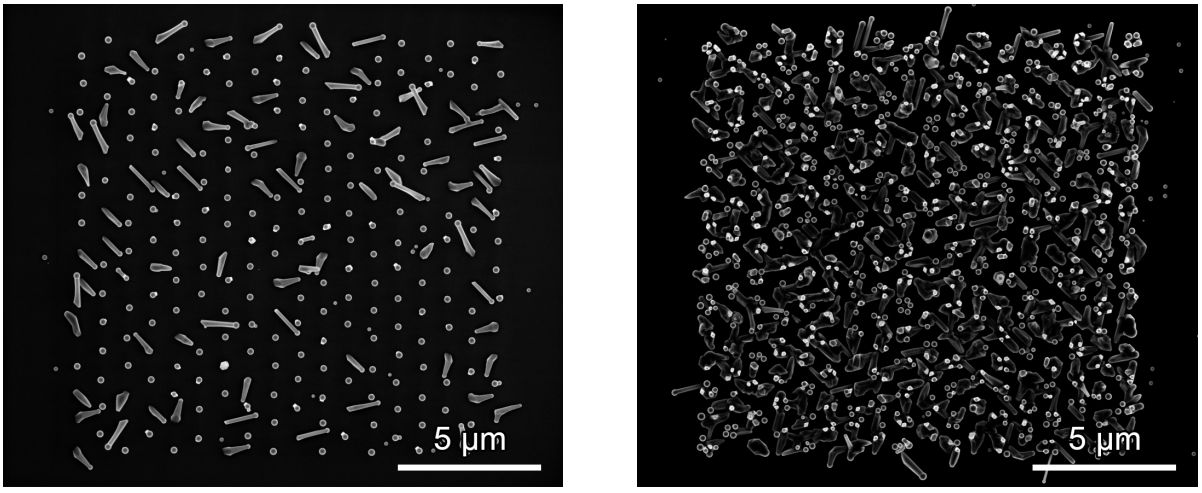



Figure 3.2: Micrograph of the sample after nanowire growth.

### 3.1.2 Dataset

The matrix of arrays was imaged as a set of
64 SE SEM images, showing a top-down view of each of the arrays. The images were captured using the electron beam in the Dualbeam used for patterning in immersion mode, using the in-lens secondary electron detector, with a voltage of 5 kV, a beam current of 86 pA and a pixel dwell time of 3 µs.

The images were captured at a resolution of 4096 × 3775 pixels. As shown in figure 3.3, the background was dark, and nanowires and 2D-crystals showed as bright outlines getting darker towards the center. The catalyst droplets on top of nanowires were distinguishable by their circular shape. The tip of nanowires without catalyst droplets showed as very bright. Before further analysis, the images were preprocessed to remove noise, by median filtering with a kernel size of 3 px.




(a) (b)

Figure 3.3: Images of two of the arrays. The arrays shown are the ones patterned using respectively the lowest and highest ion fluence and beam diameter. (a) array 1, and (b) array 64

### 3.1.3 Detection

Both the catalyst droplets on top of nanowires, and the tips of nanowires lacking catalyst droplets, were detected. The detection was performed using the `simpleBlobDetector` method. For detecting the catalyst droplets, the detection parameters were adjusted to detect the dark centers of the droplets, by looking for dark blobs with a high circularity. For detecting the dropletless nanowires, detection parameters were adjusted to look for bright blobs with less strict demands on circularity, but still high convexity. For both detection runs, bounds were set for the size of detected blobs. The exact detection parameters used are given in table 3.2. The detection yielded numbers for the size of each detected droplet/wire, and its position in the image.

Table 3.2: Parameters used for detection of droplets and dropletless wires in the FIB patterned arrays.

| | **Droplet detection** | **Dropletless wire detection** |
|---|---|---|
| **Invert** | False | True |
| **Minimum threshold** | 50 | - |
| **Maximum threshold** | - | 130 |
| **Minimum area** | 200 px | 20 px |
| **Minimum circularity** | 0.85 | 0.7 |
| **Minimum convexity** | - | 0.9 |
| **Minimum inertia ratio** | 0.8 | - |

### 3.1.4 Lattice of holes

To analyze the occurrences and positions of the detected nanowires in relation to the patterned holes, the coordinates of the holes are needed. The holes are not visible on the top-down images, but they are known to be positioned in a regular lattice. The coordinates of the holes were thus approximated by fitting an ideal lattice to the positions of the detected nanowires.

For each image, an initial guess for the lattice was defined by manually defining the corners of the array through mouse input. The lattice parameters were then numerically optimized to achieve the minimum sum of squared distances between each detected wire and its nearest lattice point. This was done using the method `scipy.optimize.minimize`, with the numerical method *Nelder-Mead*. Other numerical methods were tried as well, but the *Nelder-Mead* method was selected, as it provided robust results.

Each detected wire was assigned to its nearest lattice point. Wires further than half a lattice distance from the nearest lattice point were discarded as anomalies. For each array, the yield of holes with n wires were calculated, defined as the ratio of holes containing exactly n wires to the total number of holes. The displacement vector of each nanowire from its lattice point was found, and the magnitude and angle of displacement were calculated.

## 3.2 Large NIL array

### 3.2.1 Sample

This sample was produced for an earlier study by Ren et al.[20], where it was referred to as sample A. The sample consisted of a heavily p-doped Si wafer covered by a 40 nm thick $SiO_2$ film. The sample was patterned with a hexagonal lattice of holes with a pitch of 1µm, using wet etching and nanoimprint lithography (NIL). After patterning, $GaAs_{x-1}Sb_x$ nanowires were grown on the sample by self-catalyzed VLS in a Varian GEN II Modular MBE system. Growth parameters are given in table 3.3.

Table 3.3: MBE growth parameters for NIL sample (ML = monolayers)

| | |
|---|---|
| **Ga flux** | 0.7 ML/s |
| **$As_2$ flux** | $2.5 \cdot 10^{-6}$ Torr |
| **$Sb_2$ flux** | $2 \cdot 10^{-7}$ Torr |
| **Growth time** | 35 min |
| **Temperature** | 625 °C |

### 3.2.2 Datasets

Two sets of images were acquired from the NIL sample, using different imaging conditions. Both datasets were acquired with a FEI Apreo HiVac SEM using the in-lens SE detector. The FEI Maps 2.5 software present on the SEM was used to automatically acquire a set of overlapping images covering a large area, and stitch the images together to form one combined image covering the whole imaged region.

Dataset 1 was imaged with a voltage of 5 kV, a beam current of 25 pA and a pixel dwell time of 5.00 µs. 10 × 10 overlapping images with a resolution of 3 072 × 2 048 px were acquired, and stitched together to form a combined image of 27 778 × 18 100 px covering an area of 271.3 µm × 176.8 µm, or 0.048 $mm^2$, giving a scale of 9.77 nm/px. The combined image was provided as a set of 28 × 18 non-overlapping tiles, each with a resolution of 1024 × 1024 pixels. An overview of the entire imaged region is shown in figure 3.4.

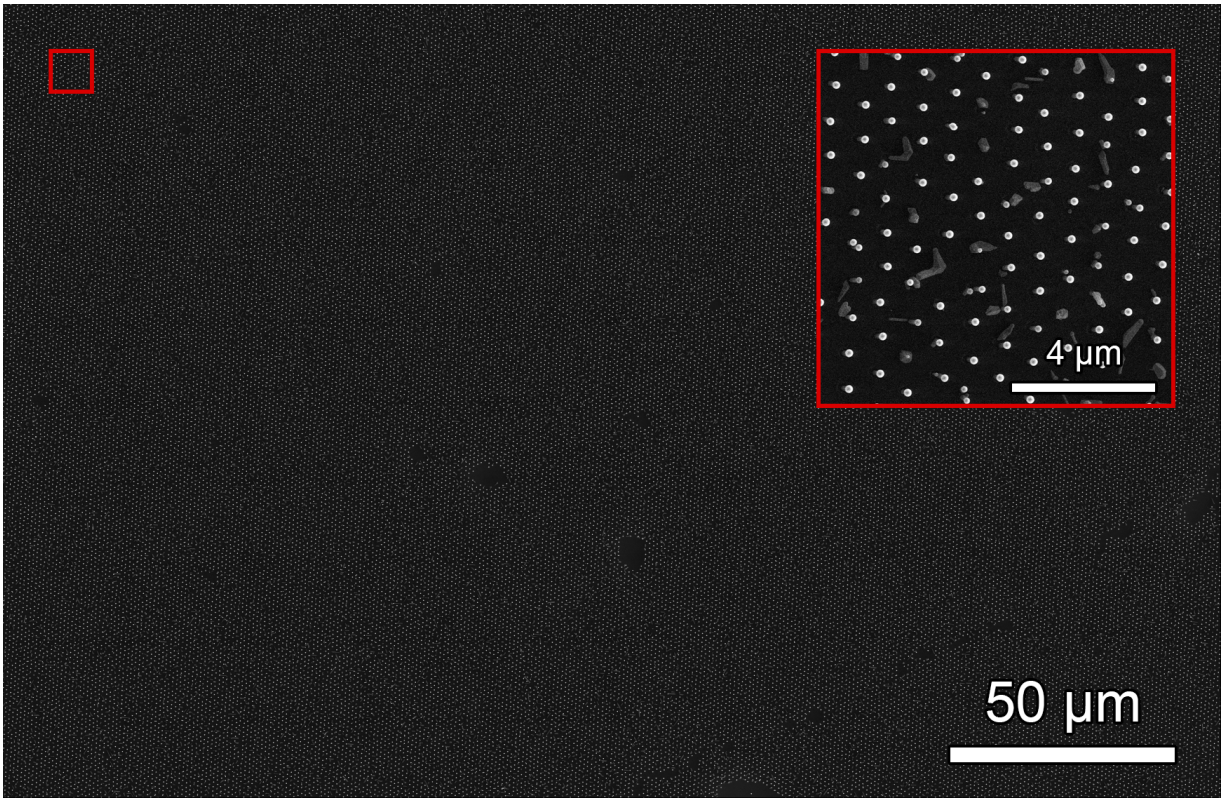



Figure 3.4: Overview of the imaged region of dataset 1. Inset: Enlarged view of the marked tile

Dataset 2 was imaged with a voltage of 15 kV, a beam current of 0.10 nA and a pixel dwell time of 1.00 µs. 5 × 5 overlapping images with a resolution of 6 144 × 4 096 px were acquired, and stitched together to form a combined image of 28 195 × 18 458 px covering an area of 91.78 µm × 60.08 µm, or 0.0055 $mm^2$, giving a scale of 3.26 nm/px. The combined image was provided as a set of 28 × 18 non-overlapping tiles, each with a resolution of 1024 × 1024 pixels. An overview of the entire imaged region is shown in figure 3.5.

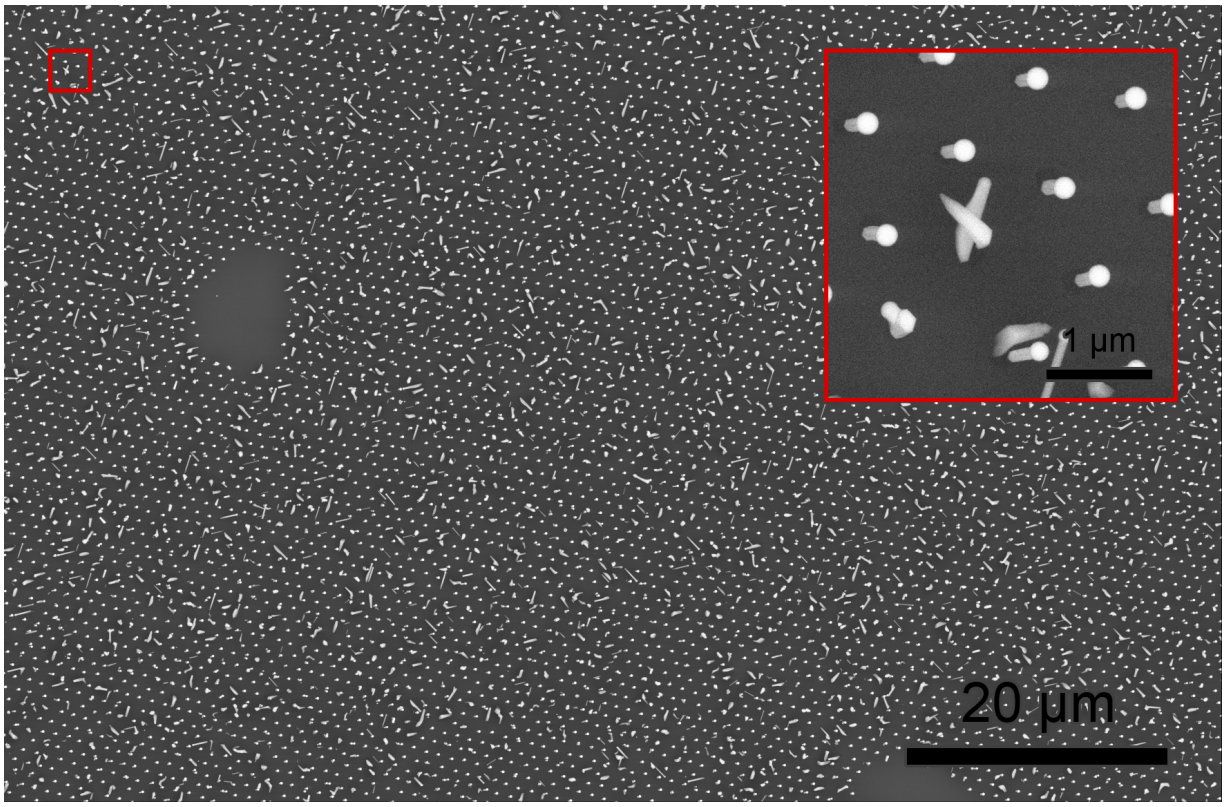



Figure 3.5: Overview of the imaged region of dataset 2. Inset: Enlarged view of the marked tile

### 3.2.3 Preprocessing

For each of the two datasets, the tiles were preprocessed to aid with detection. Erosion based reconstruction was used to fill in the dark centers of the droplets. To remove shot noise, and smooth out fuzzy edges, the images were median filtered with a kernel size of 5 px for dataset 1, and 7 px for dataset 2. Finally, another pass of the erosion based reconstruction was applied.

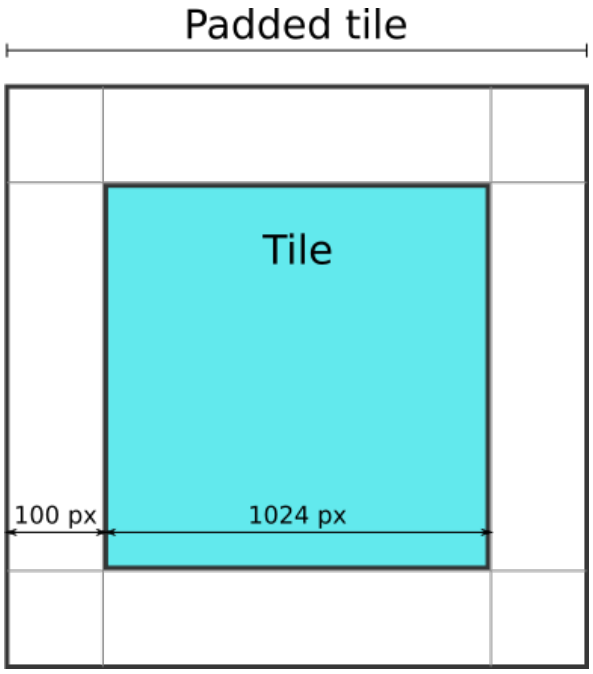



Figure 3.6: Illustration of a padded tile. The colored region represents the original tile, whereas the larger square is the padded tile, created by adjoining parts of neighboring tiles. The illustration is not to scale.

Padded tiles were used to avoid edge effects on the borders between tiles. For each tile, a padded tile was generated, using parts of neighboring tiles to extend the tile by 100 px in each direction, as illustrated in figure 3.6. The preprocessing was run on the entire padded tile. The padded region was then cropped away, returning the tile to its original size.

### 3.2.4 Detection

For both datasets, droplets were detected using the `simpleBlobDetector` method, with the parameters given in table 3.4. Detection was performed on one tile at a time, but the coordinates of detected blobs were stored with respect to the entire image. To avoid problems with the detection of droplets situated on the border between tiles, padded tiles were used for detection. Detection was performed on the entirety of the padded tile, but detected blobs whose centers were located outside the original tile were discarded, so as to avoid duplicate detections of the same blob.

Table 3.4: Parameters used for blob detection for each of the two NIL datasets

| | **Dataset 1** | **Dataset 2** |
|---|---|---|
| **Invert** | True | True |
| **Maximum threshold** | 200 | 200 |
| **Minimum area** | 40 px | 200 px |
| **Minimum circularity** | 0.8 | 0.85 |
| **Minimum convexity** | 0.0 | 0.9 |

### 3.2.5 Lattice of holes

As with the FIB arrays, the patterned holes were approximated with a lattice fitted to the detected droplets. An initial guess was obtained by a combination of user input and detected data. The lattice was then numerically fitted to the detected droplets using the same method described in section 3.1.4.

The lattice was first fitted to the detected droplets in the top left tile of each dataset. The resulting lattice was used as an initial guess for another round of fitting, using droplets from additional tiles along the top row. This was repeated until the lattice was fitted to all the tiles of the top row. The selection of droplets for the lattice to be fitted to was then expanded downwards with additional rows, until the lattice was finally fitted to droplets from the entire image. As the number of droplets in the region to be fitted exceeded 500, a random selection of 500 droplets from the given region was used instead. For a final pass a random selection of 4000 droplets from the entire image was used.

After a final lattice was obtained, each blob was assigned to its nearest lattice point, and the angle and magnitude of its displacement from the point was calculated.

## 3.3 Random growth area

### 3.3.1 Sample

The area of non-position controlled nanowire growth studied is part of the sample described in section 3.1.1. It is shown on the sample schematic in figure 3.2 as a set of gray squares along the bottom labeled "Random Growth". The squares each represent a square region uniformly FIB milled with a linearly increasing ion fluence, from 0.06 nC/µm$^2$ for the leftmost square, to 0.53 nC/µm$^2$ for the rightmost square. After patterning, the sample was further processed to grow nanowires as described in section 3.1.1.

### 3.3.2 Dataset

The images of the random growth area were acquired as a set of 8 overlapping images, each with a resolution of 4096 × 3156 px. The images were acquired using the same imaging conditions as for the matrix of arrays, as described in section 3.1.2. The images were manually stitched together, resulting in the image shown in figure 3.7. This image was cut up into a set of 19 × 4 non-overlapping tiles, each with a resolution of 1024 × 1024 pixels. The images had a contrast similar to the FIB array images, with both droplets, non-vertical nanowires and 2D growth showing as bright along the edges, and darker towards the middle. Before detection, the dataset was preprocessed in the same way as with the NIL datasets, but with a kernel size of 5 px for median filtering.

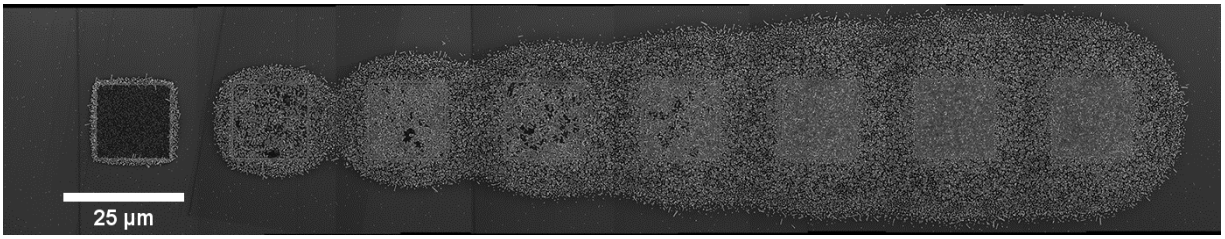



Figure 3.7: The entire stitched random growth image.

### 3.3.3 Detection

Detection of the droplets was performed for each tile, in the same way as for the NIL sample, as described in section 3.2.4. The detection parameters used are given in table 3.5.

Table 3.5: Detection parameters for random growth

| | |
|---|---|
| **Invert** | True |
| **Maximum threshold** | 200 |
| **Minimum area** | 40 px |
| **Maximum area** | 450 px |
| **Minimum circularity** | 0.7 |

# Chapter 4

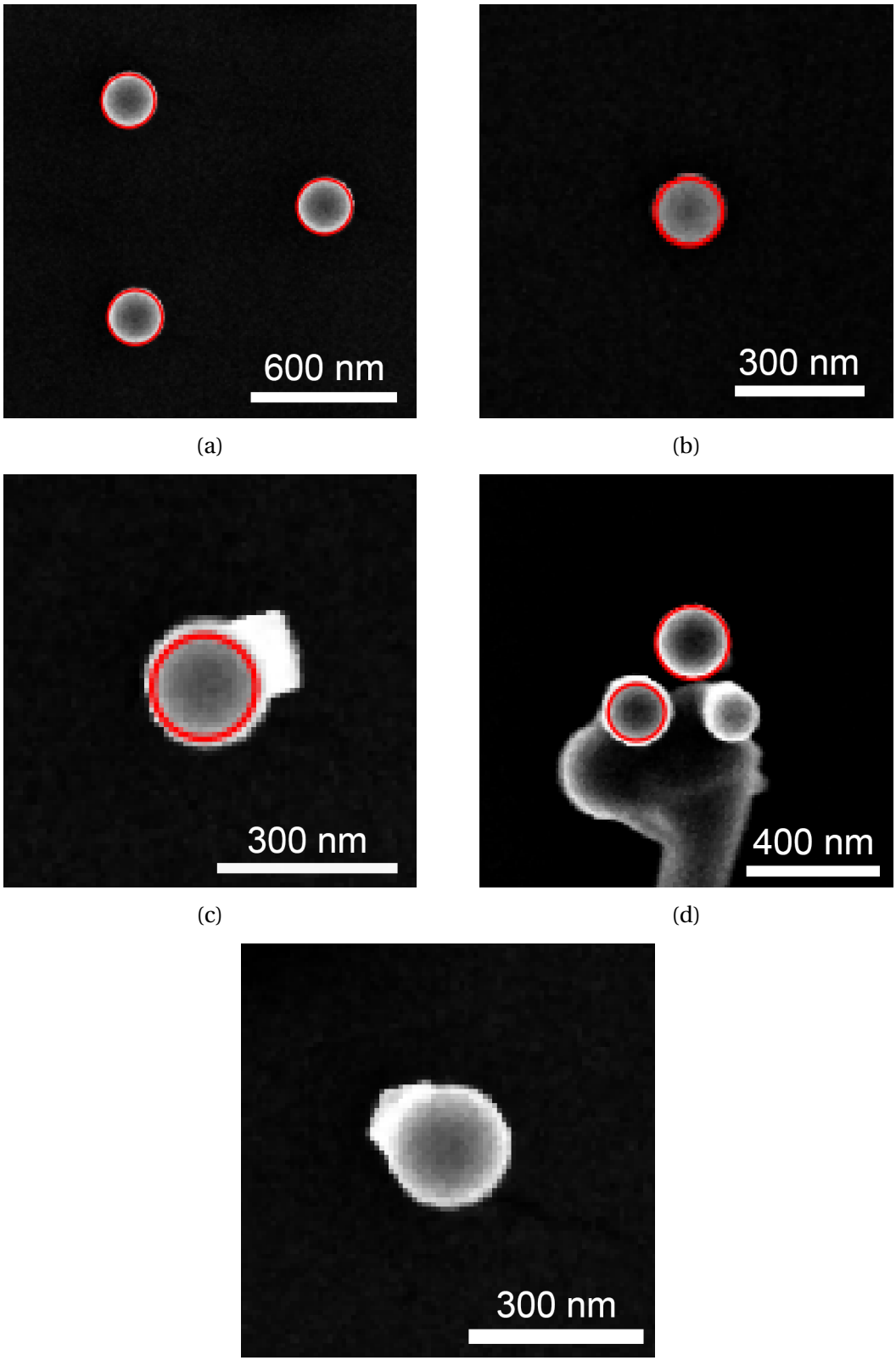
# Results

## 4.1 FIB arrays

### 4.1.1 Detection

The detection yielded a set of blobs largely corresponding to the nanowire catalyst droplets in position and size, as shown in figure 4.1(a). Some catalyst droplets on the oxide, without nanowire growth, were also detected by the algorithm, as shown in figure 4.1(b). Very occasionally, the algorithm detected features other than catalyst droplets, which also showed as a circular shaped decrease of intensity in the images.

Some of the detected nanowires were detected with a radius that was smaller than the actual radius. This often happened when other bright features were adjacent to the nanowire, as in figures 4.1(d) and 4.1(c). Figures 4.1(d) and 4.1(e) show how some nanowires, often but not always near 2D-crystals, failed to be detected. The detection of nanowires without droplets was not as accurate as the droplet detection, especially in size measurements. Thus, these detected nanowires were only used for yield calculations.




(a) (b) (c) (d) (e)

Figure 4.1: A selection of cases illustrating detection quality. (a) Well detected droplets (b) Droplet on substrate detected (c) Nanowires near 2D-crystals. Some are detected, others not. (d) Nanowire with protruding feature (e) Undetected droplet

### 4.1.2 Yields

Nanowires with catalyst droplets were detected with good accuracy. Nanowires without droplets were detected with decent accuracy. Lattices were defined and optimized successfully. By assigning the detected nanowires to lattice points, the percentage yield of holes containing exactly n nanowires, with n ranging from 0 to 5, was obtained for all arrays, and is plotted in figure 4.2.

A high yield of single nanowires (fig. 4.2(b)) was obtained in the arrays patterned with lower dose or diameter. The highest yields were observed in arrays 6, 17 and 5, with yields of 84.1 %, 83.0 % and 81.9 % respectively.

As dose and diameter increase, there is a band in the dose-diameter parameter space where a high amount of the holes have no nanowires at all (fig 4.2(a)). Array 26 has the largest amount of holes with no detected nanowires, 48.9 %. Further increasing dose and diameter leads to a high yield of 2, 3, or even more nanowires per hole.

### 4.1.3 Droplet size

Data on the size of all detected droplets was obtained. This data was however not perfectly accurate, as the size of the detected blobs did not always perfectly correspond to the size of the droplet, as explained in section 4.1.1.

Figure 4.3(a) plots the median droplet diameter in each of the 64 arrays. The parameter space is clearly divided into two regions: one with a high median diameter (~220 - 240 nm), and one with a lower median diameter (~150 - 200 nm). The high diameter region correlates well with the region with high yield of single nanowires, shown in figure 4.2(b).

The droplet diameter distributions look significantly different in the two regions. The diameter distributions in the high diameter region are similar to the one shown in figure 4.3(b). They have a sharp peak around ~230 nm, and fewer nanowires with lower diameters. The di-

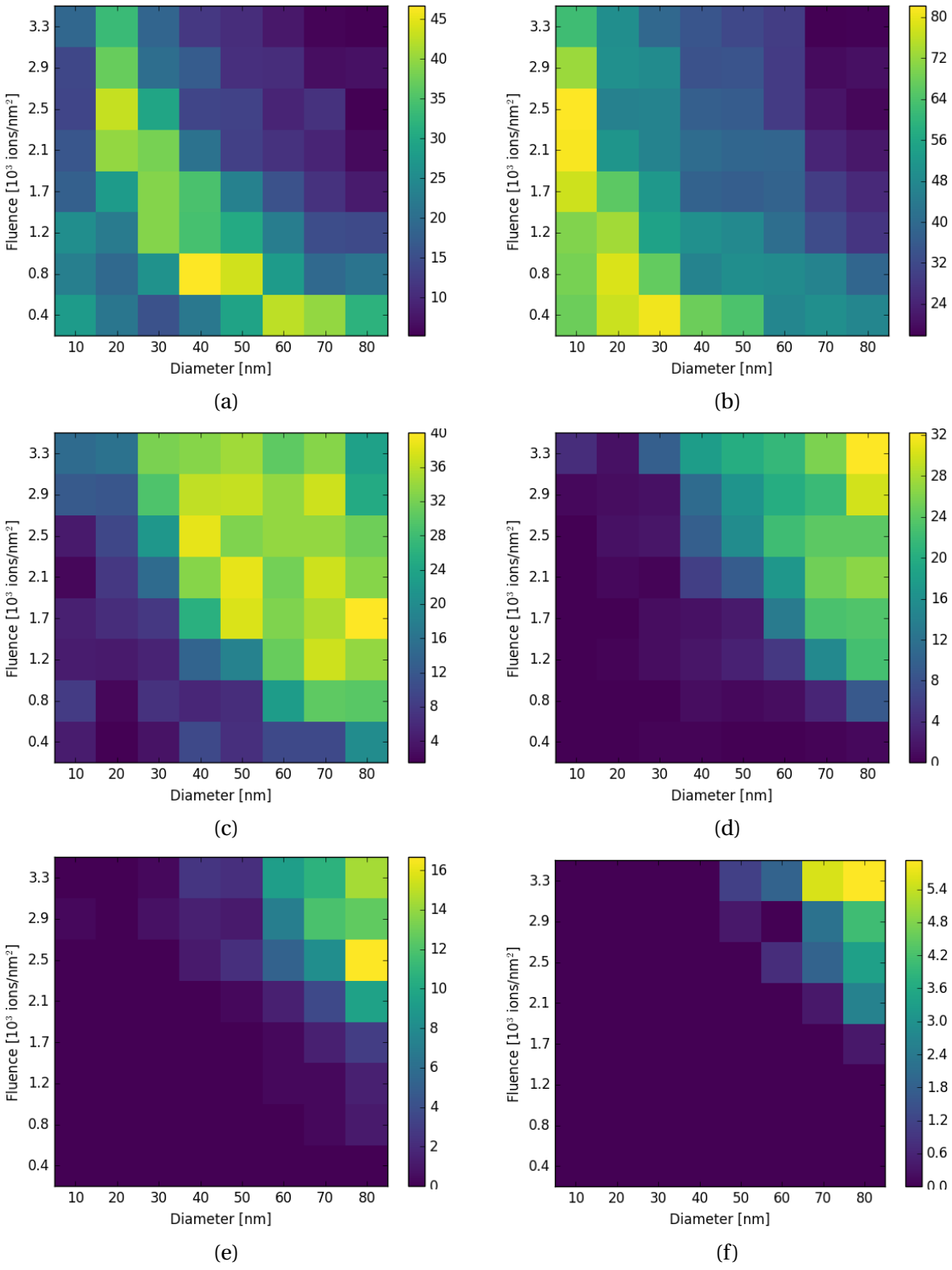
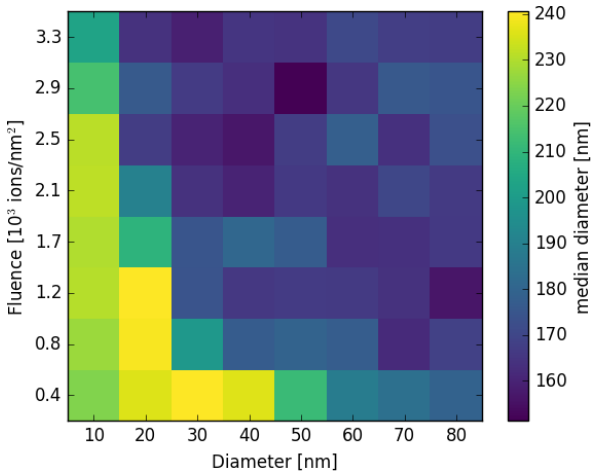
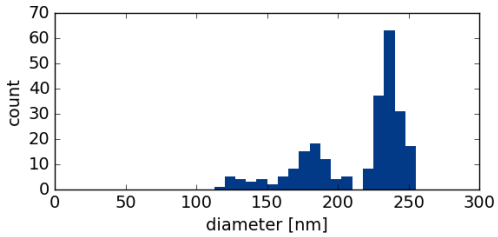


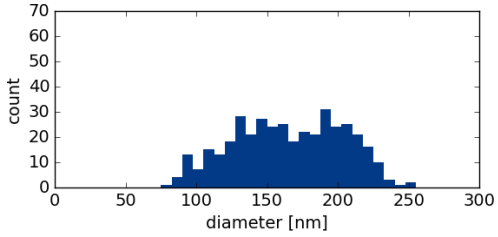

Figure 4.2: Plots showing the arrays colored by percentage yield of holes containing exactly 0(a), 1(b), 2(c), 3(d), 4(e) or 5(f) nanowires. Note that the color-bar scales vary.



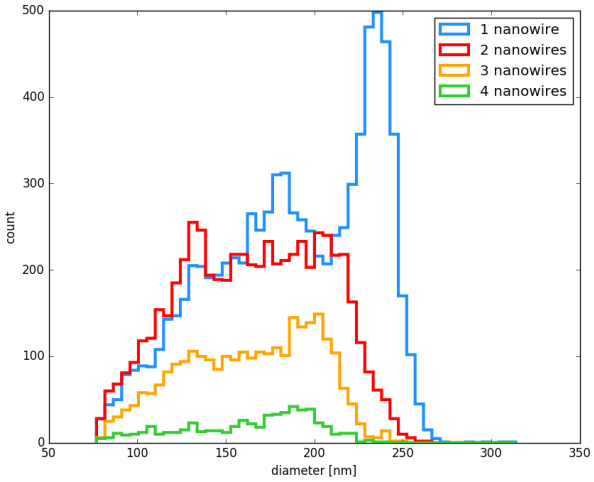

(a)




(b)




(c)




(d)

Figure 4.3: (a) Median diameter for the droplets detected in each array. (b) Histograms of the droplet diameters of droplets detected at lattice points containing 1, 2, 3 and 4 or more nanowires respectively. (c) A typical histogram of droplet diameters for arrays with high median diameter. (d) A typical histogram of droplet diameters for arrays with a low median diameter.

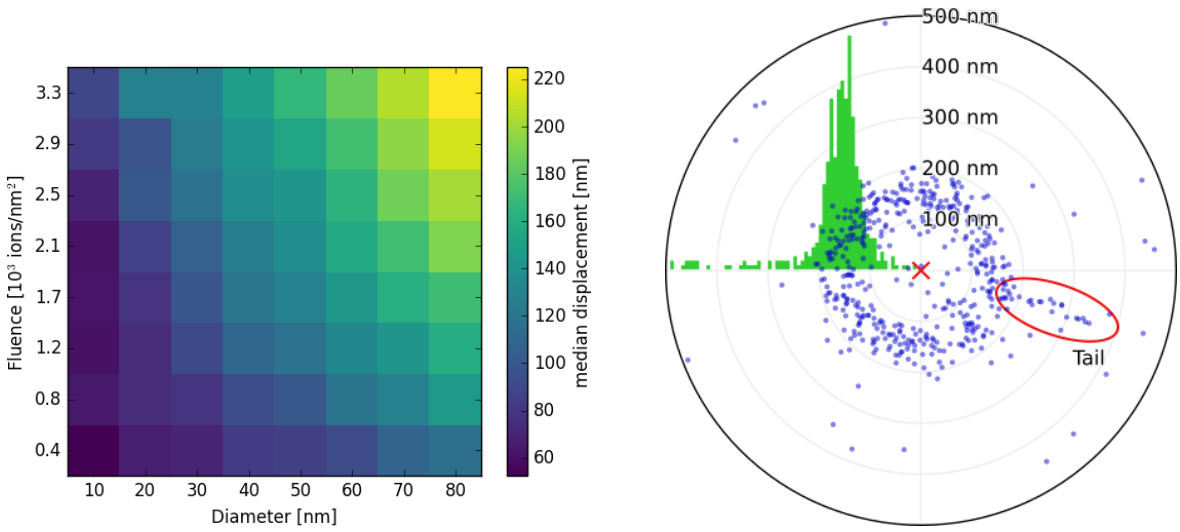
ameter distributions in the low diameter region are similar to the one shown in figure 4.3(b). They are broad, and without many droplets with a diameter of 230 nm and above.

Figure 4.3(d) shows that the ~230 nm peak in the diameter distribution is only present for single nanowires. It can also be seen that for non-single nanowires, the amount of nanowires at a given lattice point does not influence the droplet diameter significantly.

### 4.1.4 Displacements from lattice

Data on each droplet's displacement from its lattice point was successfully obtained. The magnitude of displacement increases as dose and diameter increase, as shown in figure 4.4(a). An intuitive overview of how the displacement of the nanowires varies between the different arrays can be obtained by scatter plots where each wire is represented by a dot whose position corresponds to the nanowires displacement from its nearest lattice point. An annotated example of one such scatter plot is shown in figure 4.4(b), and similar plots for all the arrays are shown in figure 4.5.

The figures 4.4(a) and 4.5 clearly show that the nanowires stray further from the hole centers as dose and diameter increase, however at the lowest diameter, increasing dose does not increase the spread much. Most displacement distributions are shaped like a circular band, with few nanowires growing in the center. For some of the arrays, particularly those with high dose, the displacement distribution has a "tail" going off to the right.




(a) (b)

Figure 4.4: (a) Median magnitude of displacement from lattice point for nanowires in each array. (b) Scatter plot of displacements for array 32. Each blue dot represents the displacement of one nanowire from its lattice point, marked by a red x. The green histogram shows the radial distribution of displacements. Some of the displacement distributions show a "tail" going off to the side, here marked by a red ellipsis.

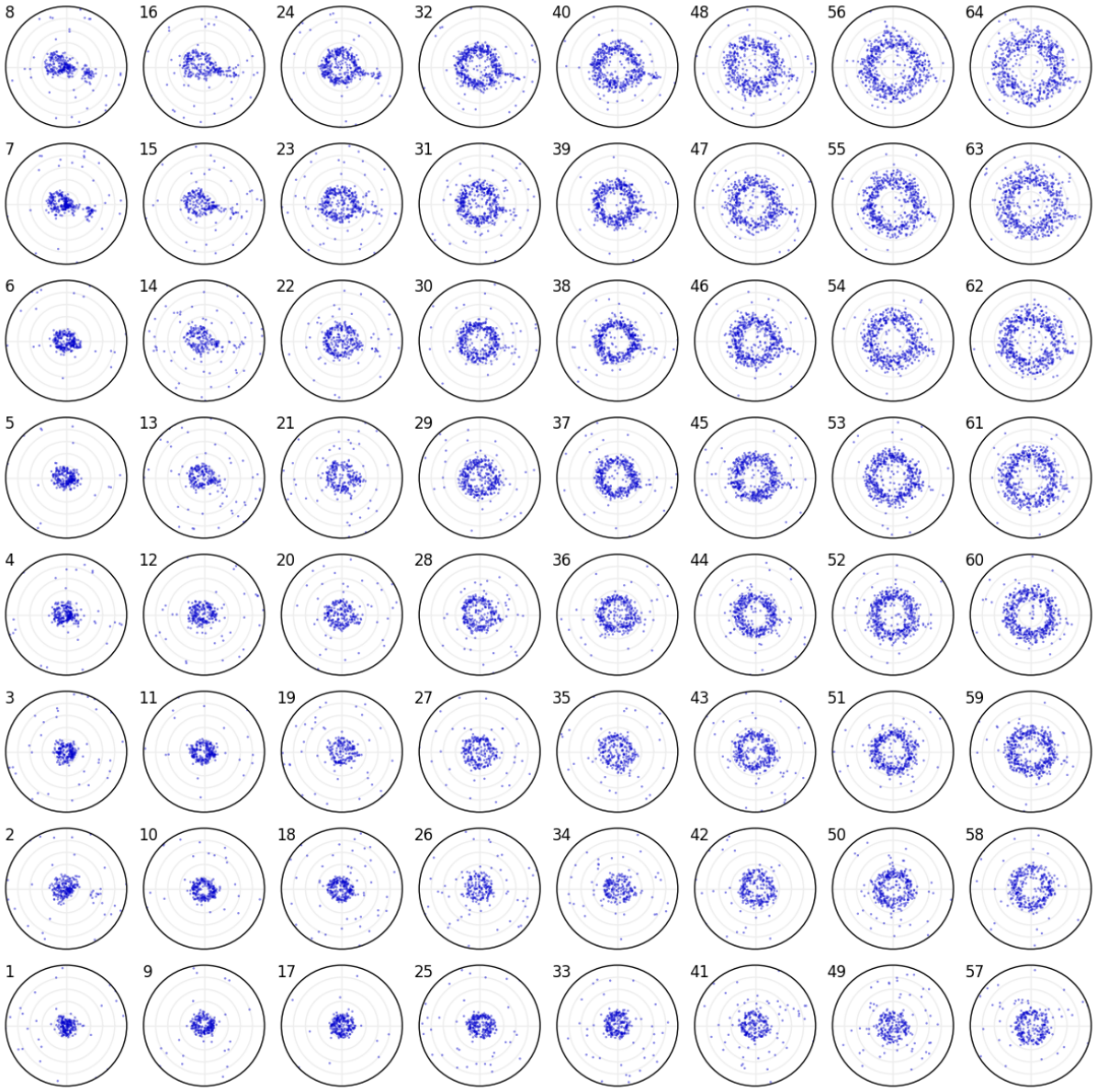



Figure 4.5: Scatter plots showing the displacement from the nearest lattice point for each nanowire on each of the 64 arrays. Axes are left unlabeled to avoid clutter, but the scale equal to that of figure 4.4(b).

## 4.2 Large NIL array

### 4.2.1 Detection

The developed routines were able to detect the nanowires on both samples. Since almost all standing nanowires were topped by catalyst droplets, the nanowires were easily detected by detecting the droplets. 53 494 nanowires were detected in dataset 1, and 6 543 in dataset 2. The two datasets were taken from the same sample, but imaging conditions differed, leading to differences in detection quality.

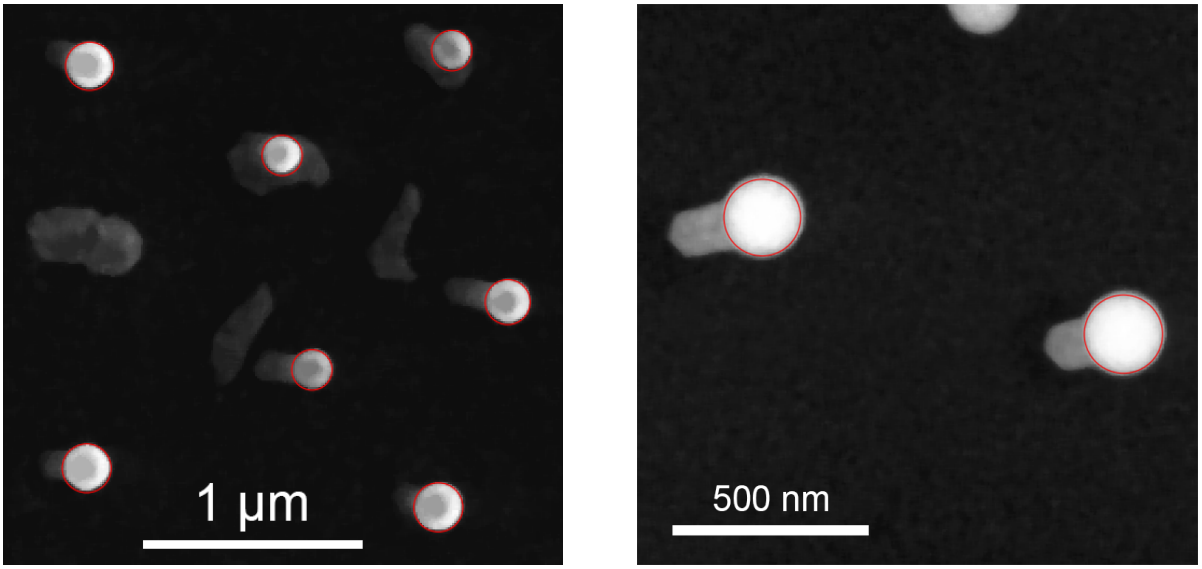
#### Dataset 1

The contrast in the images was highly favorable, as the catalyst droplets were much brighter than any other features, and the detection was thus not disturbed by non-standing nanowires or 2D crystals, as can be seen in figure 4.6(a). Some parts of the image showed artifacts resulting from erroneous image stitching, and these were occasionally picked up as large blobs.

#### Dataset 2

The original images were quite noisy, and thus median filtering with a high kernel size had to be applied. While median filtering usually preserves sharp edges, the high amount of noise resulted in features on the images having slightly blurred edges, as can be seen on figure 4.6(b). Since the sample was slightly tilted sideways during imaging, the images showed the nanowires as a feature protruding from the droplets. The blurring of edges lead to an unclear boundary between the droplets and the nanowires, and this in turn lead to only the inner part of the droplet being recognized as a circular feature. Thus all nanowires were detected with a size somewhat smaller than their actual size. When detection was performed with no size limitation, slight intensity variations in the background were picked up by the detector. This was avoided by setting a minimum size limit. Some droplets on the surface, without

nanowires, were detected. Very occasionally, 2D-crystals growing alongside the nanowire were detected as part of the droplet, leading to a blob of a larger size.



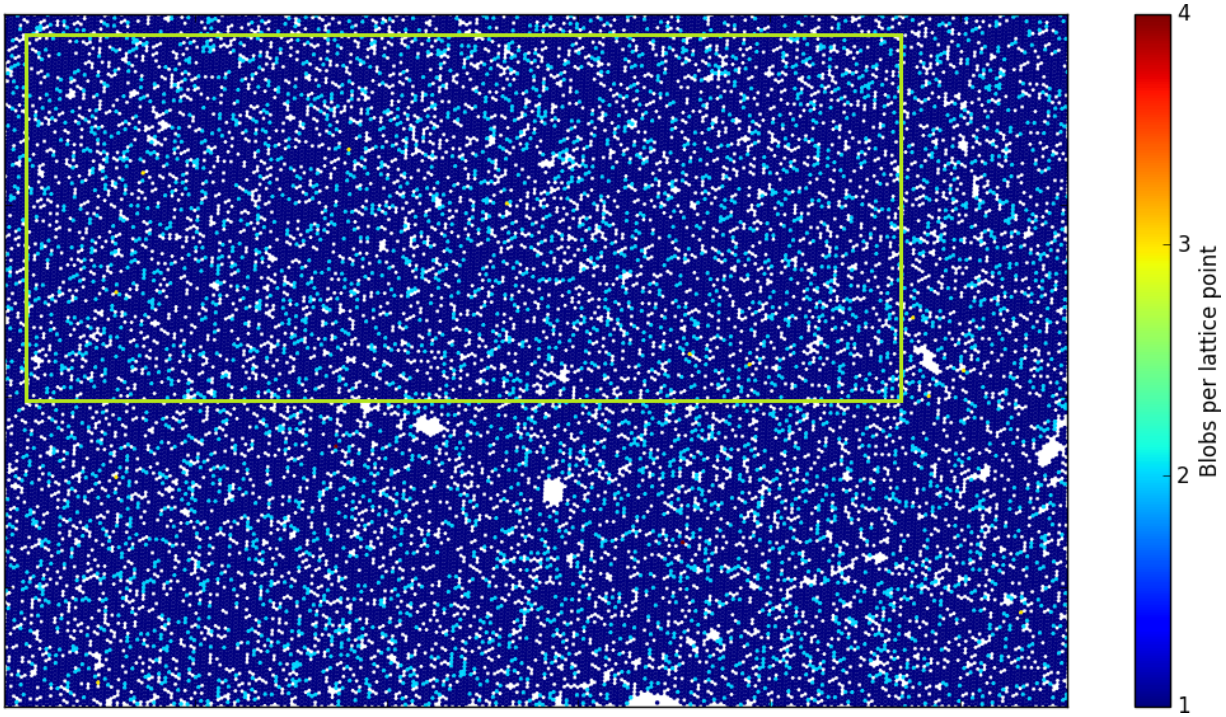

(a) (b)

Figure 4.6: Sections of NIL dataset 1 (a) and 2 (b) showing detected droplets outlined in red

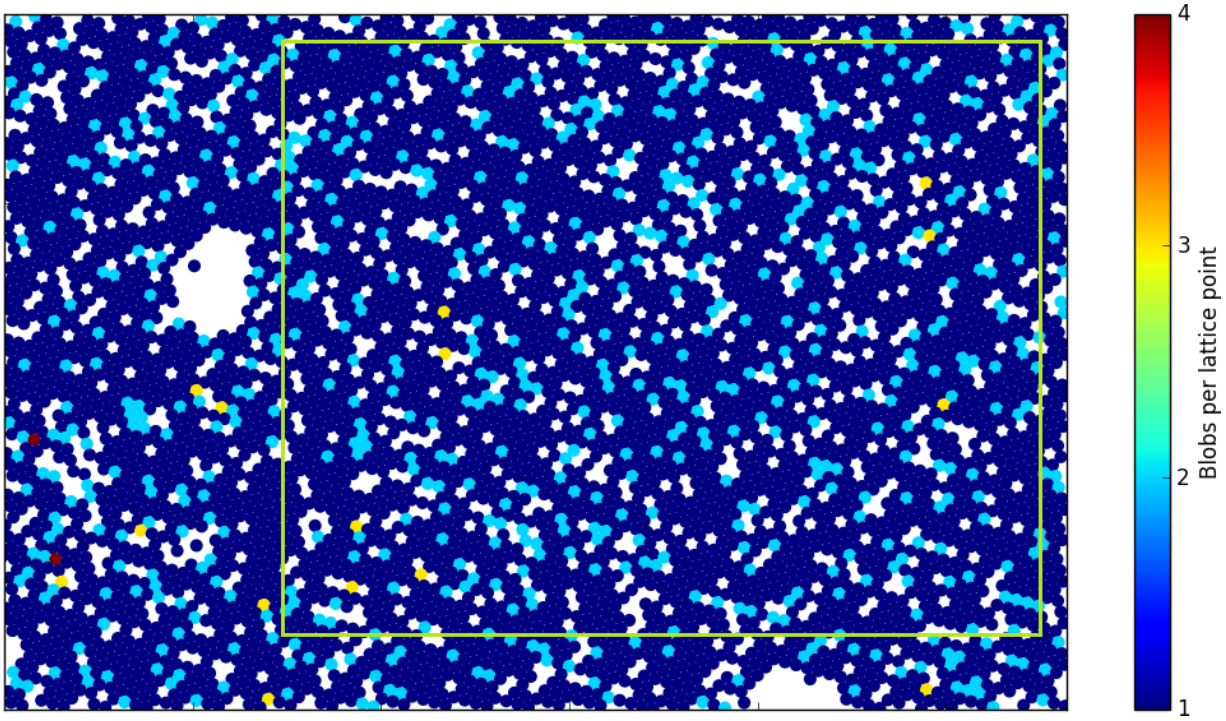
### 4.2.2 Yields

The yield numbers for single nanowires, as well as empty lattice points, and lattice points with multiple nanowires, were obtained, and are given in table 4.1. Both datasets contained areas spanning several lattice points completely devoid of nanowires, these can be seen as white areas in the plots in figure 4.7. To obtain yield numbers not affected by these anomalous areas, subregions of each dataset were selected, as shown in figure 4.7. Yields calculated for these subregions as well, and are also shown in table 4.1.

Single nanowire yields were higher in dataset 1 (~ 79 %) than in dataset 2 (~ 74 %), as the second dataset had a higher proportion of both empty lattice points, and double nanowires. The subregions had a slightly lower amount of empty lattice points, but an otherwise similar yield distribution.




(a)




(b)

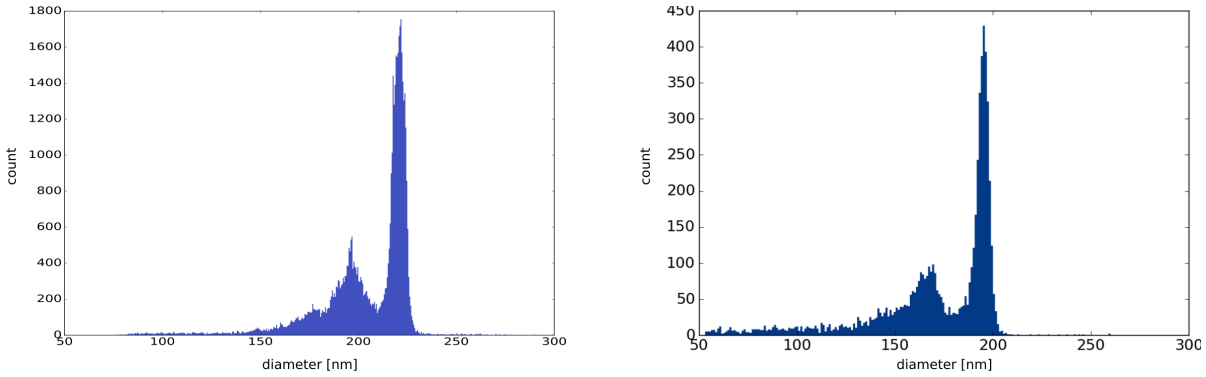
Figure 4.7: Plots of dataset 1 (a) and dataset 2 (b) where each lattice point is represented by a circle colored by the number of nanowires found at that lattice point. The green rectangles mark the subregions used for calculating yields while disregarding anomalous areas.

Table 4.1: Calculated percentage yields of the proportion of lattice points with a given number of nanowires for each of the datasets, and for subregions of the datasets as marked in figure 4.7.

| Nanowires | **Dataset 1** | **Dataset 1 subregion** | **Dataset 2** | **Dataset 2 subregion** |
|---|---|---|---|---|
| **0** | 13.26 % | 13.15 % | 15.52 % | 14.69 % |
| **1** | 78.77 % | 79.08 % | 74.09 % | 74.89 % |
| **2** | 7.93 % | 7.74 % | 10.13 % | 10.18 % |
| **3** | 0.02 % | 0.03 % | 0.23 % | 0.24 % |
| **4** | 0.004% | 0 | 0.03 % | 0 |
| **5+** | 0 | 0 | 0 | 0 |

### 4.2.3 Droplet size

Histograms showing the diameter distributions of the droplets detected on the two datasets are shown in figure 4.8. For the first dataset, most of the nanowires had a droplet diameter ranging from 150 nm to 230 nm. The diameter distribution showed two distinct peaks: A short wide peak at 196 nm, and a tall narrow peak at 221 nm. The second dataset has a similar diameter distribution, but shifted towards lower diameters, with the peaks appearing at ~170 nm and ~190 nm. The distribution for the second dataset also has a larger portion of the nanowires in the low diameter tail, and this tail stretches to smaller diameters than for the first dataset.




(a) (b)

Figure 4.8: Histograms showing the diameter distribution of the detected nanowire droplets in dataset 1 (a) and dataset 2 (b)

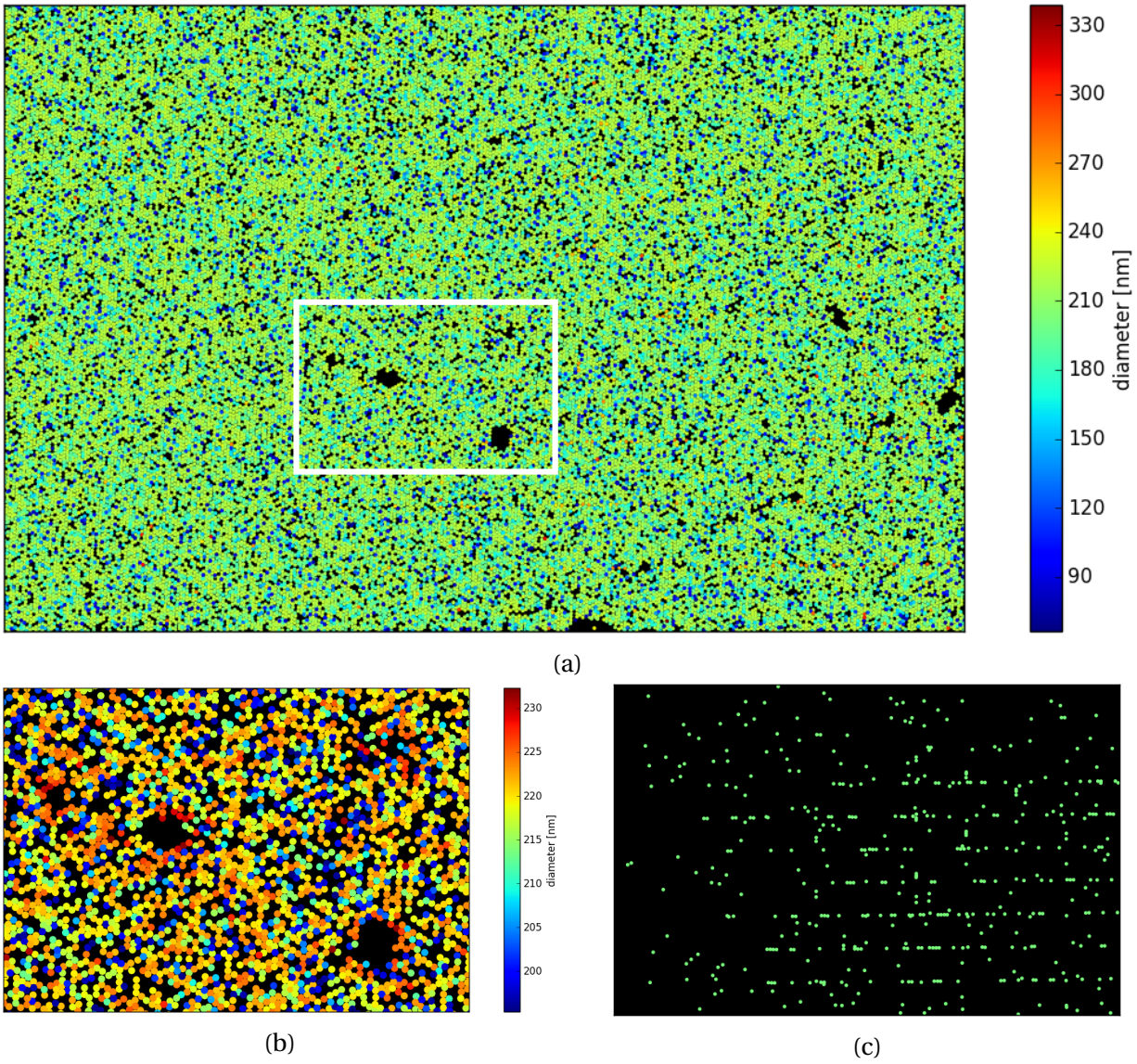
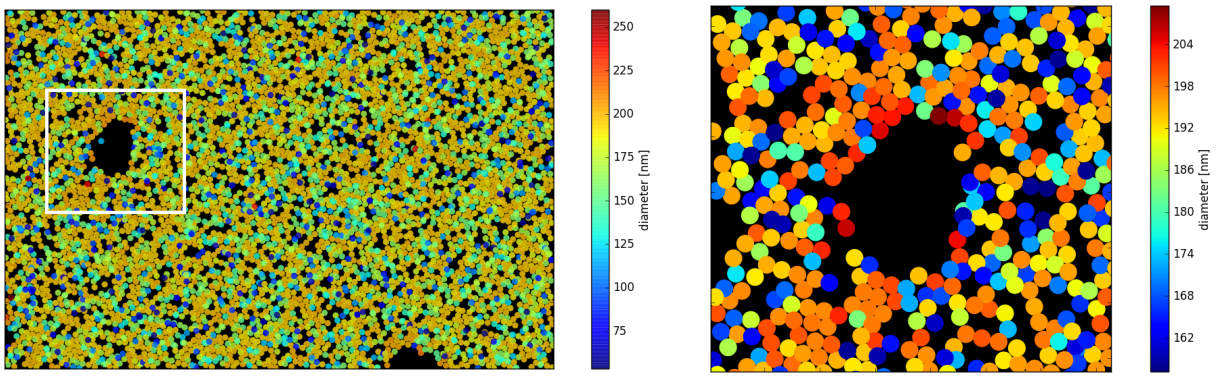



Figure 4.9: (a) Plot of all detected droplets in dataset 1. Each droplet is represented by a dot colored by droplet diameter. (b) Enlarged view of region marked in (a). Droplets with diameters smaller than 195 nm or larger than 232 nm have been excluded to increase contrast. (c) Plot of all droplets with a diameter larger than 234 nm (not colored by size).




(a) (b)

Figure 4.10: (a) Plot of all detected droplets in dataset 2. Each droplet is represented by a dot colored by droplet diameter. (b) Enlarged view of region marked in (a). Droplets with diameters smaller than 157 nm or larger than 223 nm have been excluded to increase contrast.

The diameter maps shown in figures 4.9(a) and 4.10(a) show that the size of droplets is mostly independent of location on the sample. Some interesting phenomena are however seen. Droplets around areas with no nanowire growth tend to be slightly larger than droplets further away from such areas, as shown in figures 4.9(b) and 4.10(b). For the first dataset, it is also observed that many of the largest detected blobs lie along stitching lines in the image, as seen in figure 4.9(c). As explained in section 4.2.1, the size of these blobs are due to the fact that droplets from adjacent images have been misaligned during stitching.

### 4.2.4 Displacements from lattice

#### Dataset 1

The magnitude and angle of displacement from the lattice point of each nanowire is plotted in figure 4.11. From the displacement magnitude map it seems that the nanowire positions are well matched by the lattice in two bands to the left and right of the middle. In the middle of the image, the nanowires seems to generally be further from the lattice, and towards the left and right edge, the mismatch is even larger. The displacement angle map shows that the angle of displacement varies across the image, and nanowires in a certain region seem to be displaced at a similar angle from the lattice. Additionally, both plots show that the displace-

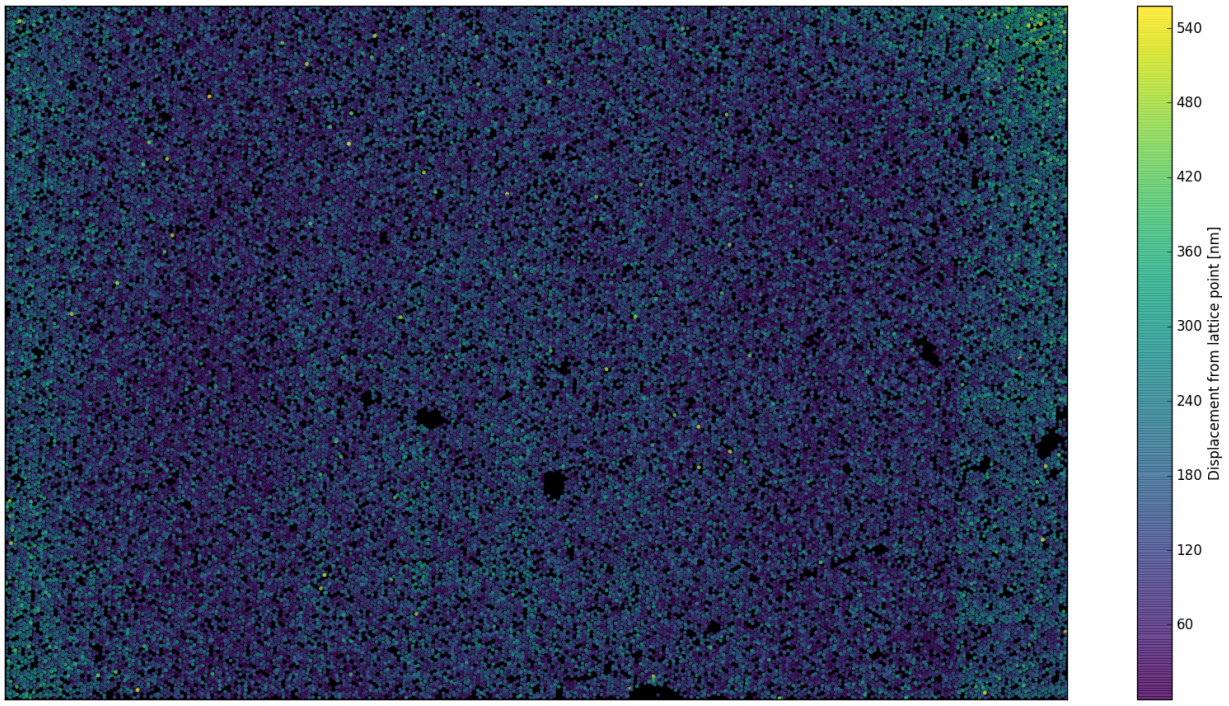
ment seems to change abruptly across the stitching boundaries of the original images, visible as a grid like pattern. This is most clearly visible in the displacement angle map.

Looking at the plots in figure 4.12, the displacement density decreases gradually from the center. The region with highest density seems to be slightly down and to the left of the center.

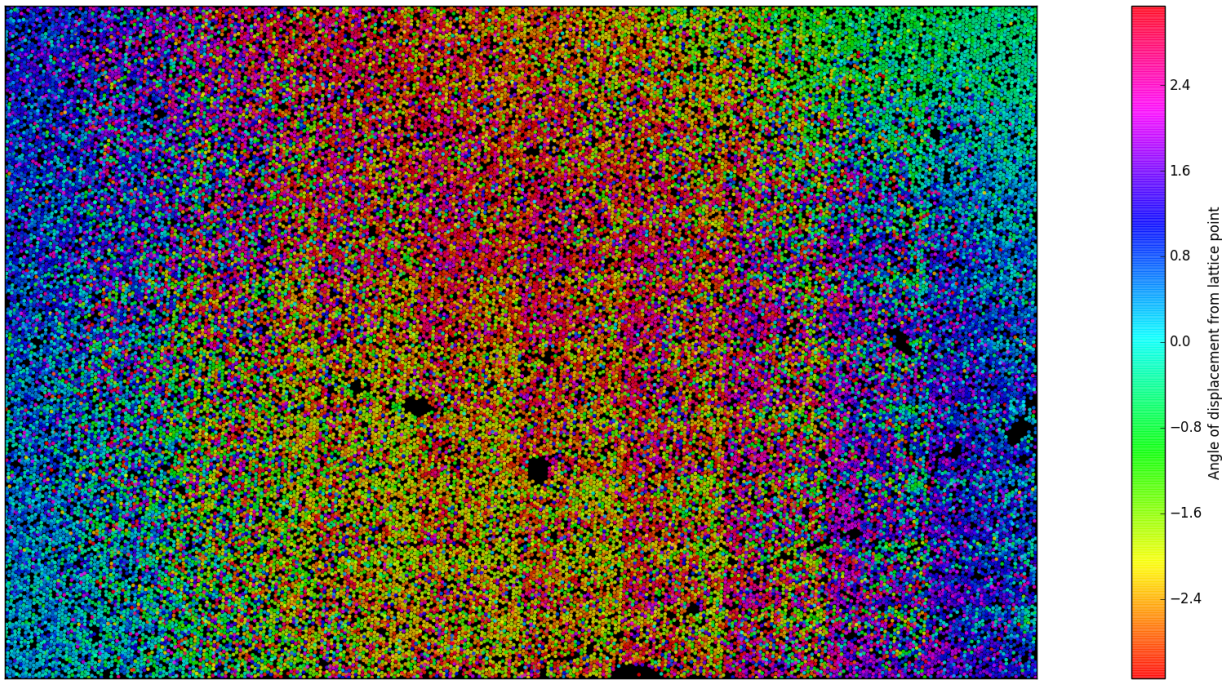
**Dataset 2**

The displacement magnitude and displacement angle maps for the second dataset, shown in figure 4.13 show that the nanowire displacement does not really vary across the sample. The magnitude of displacement (figure 4.13(a)) is consistently low, with a few outliers distributed across the whole imaged region. The displacement angle (figure 4.13(b)) seems to vary randomly, with no large regions of similarly displaced nanowires.

The plots in figure 4.14 show a displacement distribution distinctly different from that of the first dataset. Most of the nanowires are displaced less than 125 nm from their lattice point. These nanowires are displaced at all angles equally. Almost all of the rest of the nanowires are displaced between 125 nm and 232 nm from their lattice point. The angular distribution of these nanowires features a trigonal symmetry, where the wires tend to be displaced towards one of three angles 120° apart. A closer look at the images reveals that almost all of these nanowires have either other nanowires or 2D crystals growing in the same hole. This is shown in figure 4.15. The remainder of the nanowires are outliers, with a displacement larger than 232 nm.

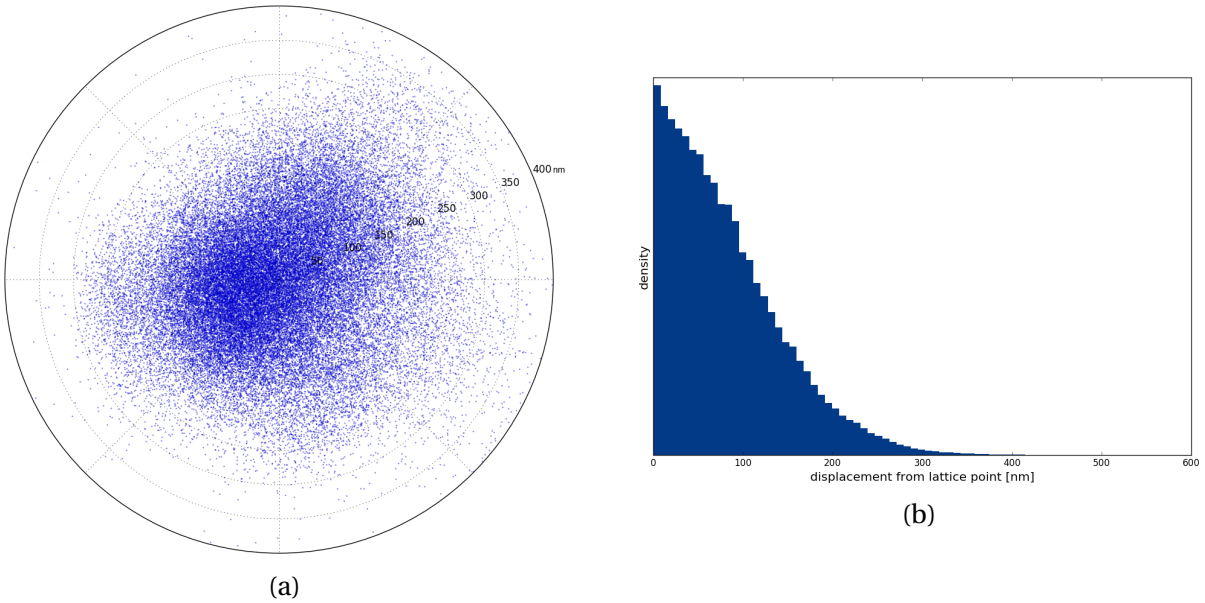



(a)



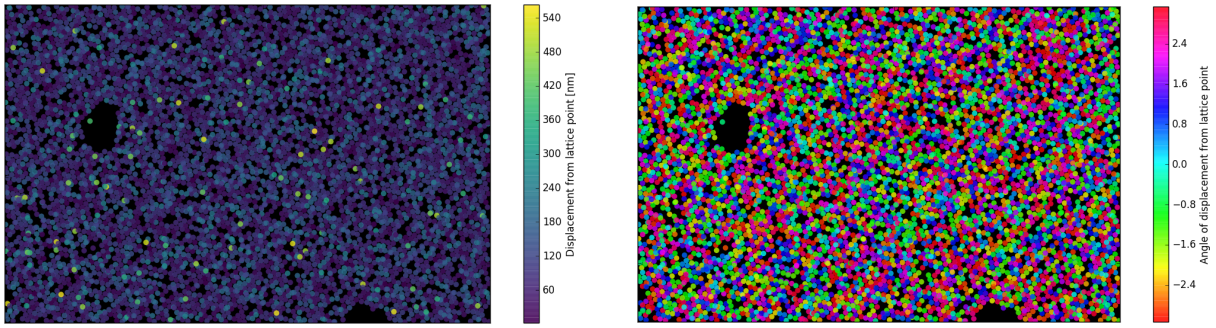

(b)

Figure 4.11: Plots showing all detected droplets in dataset 1 colored by the magnitudes (a) and angles (b) of displacements from their lattice points. Angles given in radians.



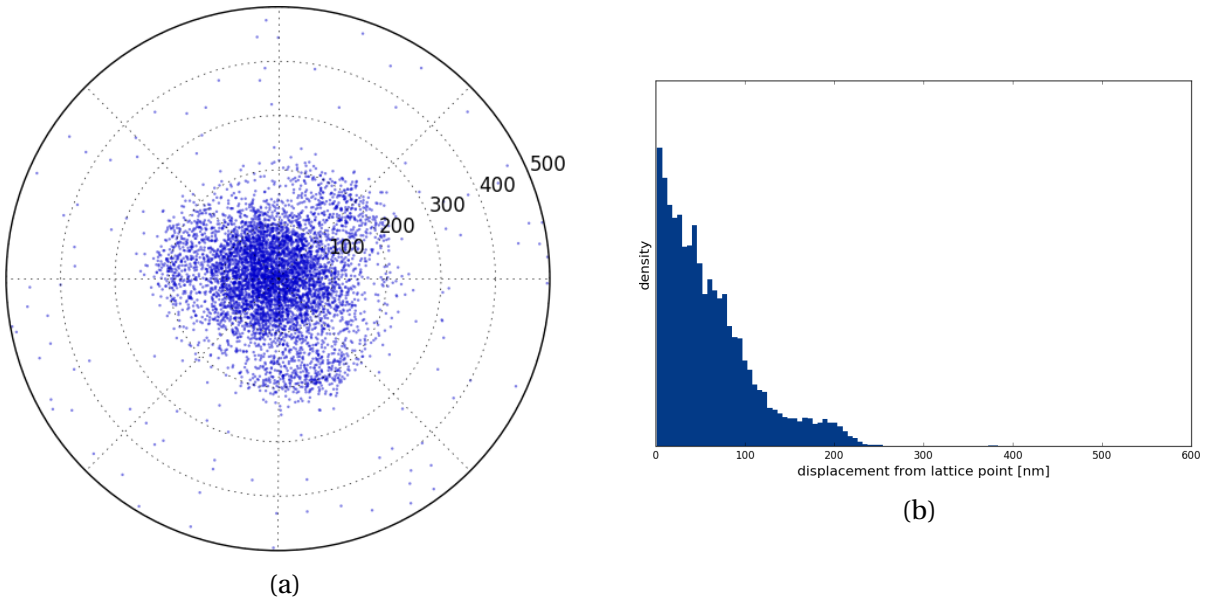

(a) (b)

Figure 4.12: (a) Scatter plot showing the displacement from lattice for all detected droplets in dataset 1. Distances in nm. (b) Histogram showing the radial density of the plot in (a).




(a) (b)

Figure 4.13: Plots showing all detected droplets in dataset 2 colored by the magnitudes (a) and angles (b) of displacements from their lattice points. Angles in radians.




(a) (b)

Figure 4.14: (a) Scatter plot showing the displacement from lattice for all detected droplets in dataset 1. Distances in nm. (b) Histogram showing the radial density of the plot in (a).

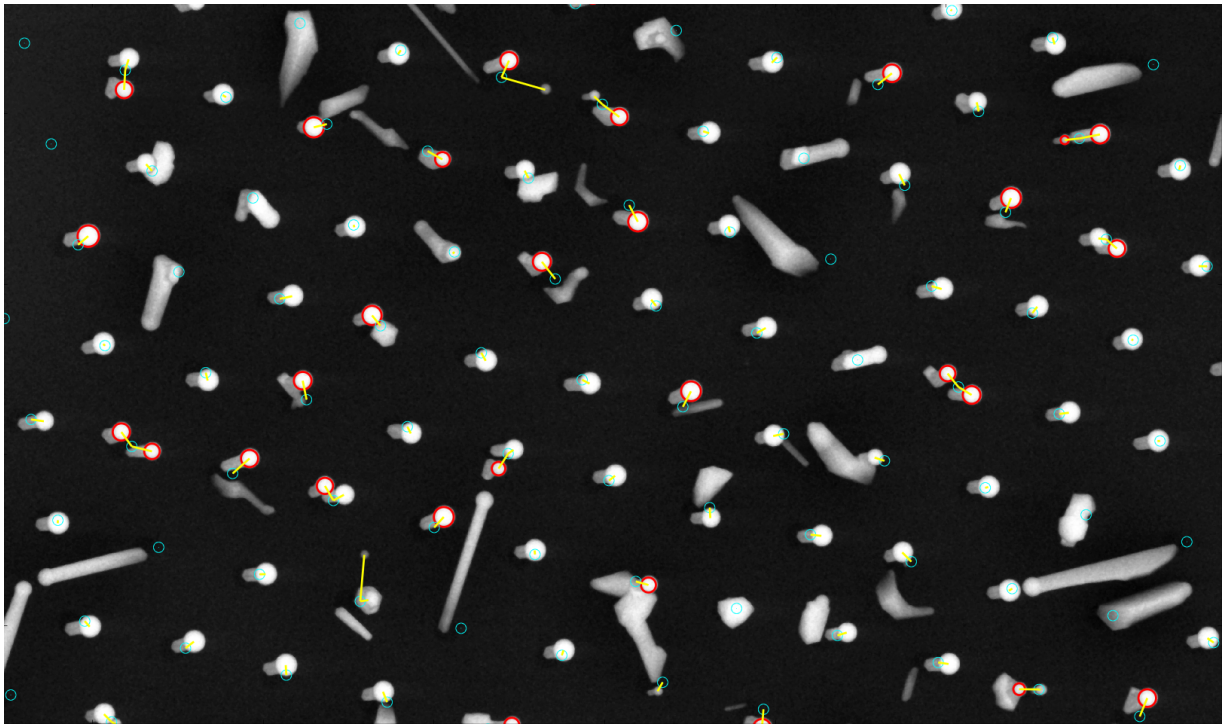


Figure 4.15: A section of the second dataset, showing blobs with a displacement distance from their lattice point of between 125 nm and 232 nm circled in red.

## 4.3 Random growth area

34 107 droplets were detected. The location and diameter of each wire was obtained. The density map in figure 4.17(a) shows a low density of droplets inside the patterned squares, a higher density in the growth areas around the squares. The density is about the same in the whole growth area, except for along the edges, where the density is higher. The size map in figure 4.17(b) shows that the droplets are smaller along the edges where the density is higher. The droplets are also smaller within the patterned squares, and the droplets found outside the growth area are smaller still. The size histogram in figure 4.16 has a large peak at ~200 nm, and a smaller one at ~100 nm.

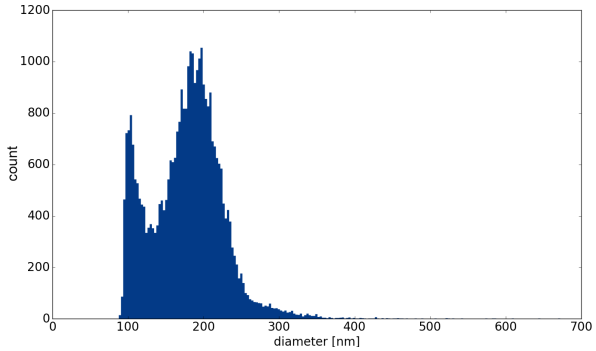
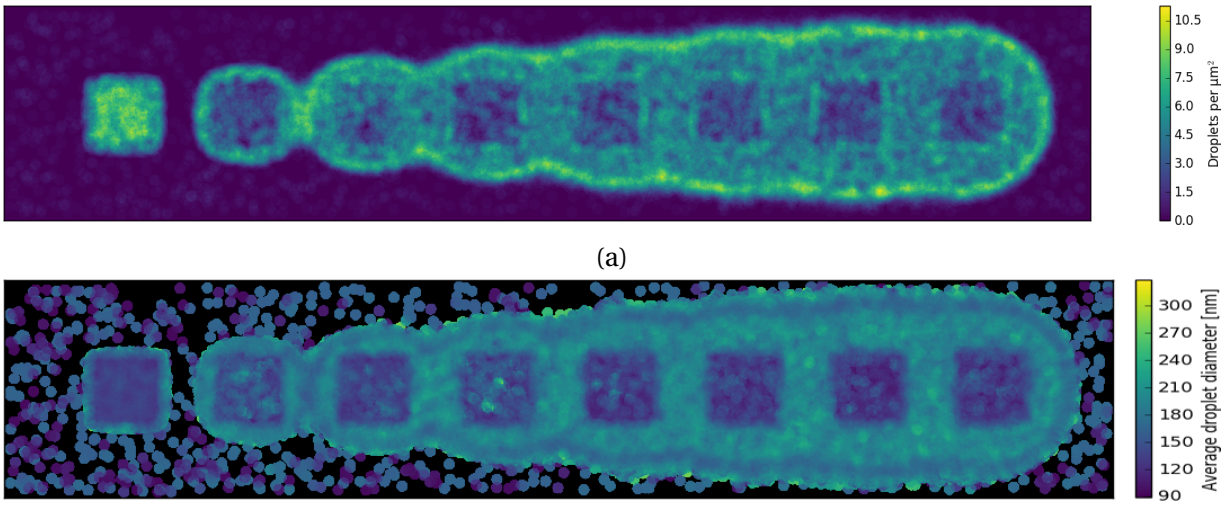



Figure 4.16: Histogram showing the diameter distribution of the detected nanowire droplets in the random growth dataset




(a)

(b)

Figure 4.17: Maps showing how the droplet density (a) and diameter (b) vary across the random growth sample.

# Chapter 5

# Discussion

## 5.1 Developed routines

### 5.1.1 Preprocessing

Median filtering was shown to be a good technique for removing noise from images to prepare them for subsequent feature detection. As the nanowire droplets are circular, their shape is not significantly distorted by median filtering, and as the droplet diameters were much larger than the kernel sizes used for median filtering, the median filtering would not reduce their size significantly. Median filtering also preserved sharp feature edges well, except for in cases with large amounts of noise, where edges were somewhat blurred. When nanowires were standing very closely together, such as in the random growth sample, the kernel size used for median filtering had to be limited to prevent the nanowire droplets from melding together.

Morphological reconstruction by erosion worked nicely to fill in the dip in intensity in the middle of nanowire catalyst droplets when this was desired for detection. With large amounts of non-vertical nanowires or 2D-crystals present, one must be careful with the usage of this

algorithm, and make sure it does not cause problems with detection by filling in areas between these features.

### 5.1.2 Detection

The segmentation based detection method used in this work is able to detect nanowire droplets in a manner sufficient to obtain several kinds of useful data. Given circular droplets, and good contrast between the droplets and other features in the image, excellent data can be obtained. If there is insufficient contrast between the droplets and other adjacent features, detection might be problematic. Contrast within the droplet will often still allow for detection, but the detected area will in that case be smaller than the actual size of the droplet, thus any size measurements will be too low.

Under some imaging conditions, the nanowire catalyst droplets will appear bright at the edges, and gradually darkening towards the center. In this case, detection can be performed in two different ways. Firstly, the dark centers of the droplets can be detected. This method is robust when it comes to features surrounding the droplet, but depending on the contrast and the resolution of the images, the detected size could end up being smaller than in reality. Given high resolution images with low noise, most droplets can however be detected with a highly accurate radius in this way.

The other way to detect such droplets is to fill in the centers using erosion based reconstruction, and detect the resulting bright droplets. If the droplets are clearly brighter than their surroundings, this will give an accurate detected size. If however there are other bright features adjacent to the droplet, these might either be detected along with the droplet, resulting in a much too large detected size, or prevent the detection of the droplet in question.

### 5.1.3 Optimizing lattice

The lattice used to approximate the ideal positions of the nanowires was defined by using an initial guess lattice, generated from a combination of knowledge about the sample, user input on images, and detected data. The lattice parameters were then optimized using a minimization algorithm to adjust the parameters until a minimum total square distance between droplets and their neighboring lattice point was obtained, thus achieving the lattice that best fit the detected droplets.

The process of assigning blobs to their nearest lattice point was found to be too time consuming to be done for each iteration of the optimization. Thus blobs were not reassigned during a round of optimization, so the lattice points used for distance calculation were the lattice points nearest each given blob at the start of optimization. This means that for the optimization result to be accurate, the initial guess must be accurate enough that each blob has the correct nearest neighbor lattice point.

For the smaller arrays, this was not a problem, as the user could input the corners, and things would be pretty good. With the large tiled images, this was more difficult. As the initial guess was based on one part of the image, any slight error in lattice distance would translate to a large error on the other side of the image. This would mean that droplets would not be assigned to the correct lattice point before optimization, and the optimization would fail to produce a good fit.

To mitigate this, a method was developed where the lattice first is optimized for a small part of the image, and the area then is stepwise expanded, running a new round of optimization each time, using the previous lattice as an initial guess. This ensures that the lattice is never too far off from the droplets in the region for which it is optimized, and finally fits the entire image.

Another concern was the fact that calculating the sum of squared distances to the nearest lattice point for all blobs becomes very time consuming when the number of blobs becomes large (up to 50 000 for the largest dataset). Since this is performed many times for every

optimization round, lattice optimization becomes unusably slow when there are too many blobs. To solve this, a smaller selection of blobs was used. It was found that using a random selection of the blobs in the region where optimization occurred was sufficient to obtain a good lattice fit, as long as the number of blobs was not too small. To ensure good accuracy of the final result, a larger selection of blobs was used for the final round of optimization.

## 5.2 FIB arrays

Patterning a matrix of arrays in the fashion done in this experiment, where the matrix represents a parameter space, with arrays patterned using all possible combinations of the selected values of the two parameters studied, allows for an in-depth study of how these parameters affect nanowire growth independently, and how they interact. Using automated computer vision techniques enables an efficient analysis of the arrays, and gives objective, reproducible and well documented results. The results of computer analysis can be presented in ways that make it easy to spot trends and see connections.

### 5.2.1 Yields

Nanowire yield is one of the most important figures when evaluating nanowire growth procedures. With nanowires identified algorithmically, the nanowire count in an area is easily obtained, much more efficiently than when counting by hand. Simply using the nanowire count to obtain a yield number is however not a good indication of growth quality. An array where half of the growth sites contained two nanowires each, while the other half contained no nanowires, would have the same nanowire count as a similar array of perfect single nanowires. To obtain yield numbers that accurately represent the state of the sample, detected nanowires must be assigned to their respective growth sites, and the percentage yield of growth sites containing the desired number of nanowires must be calculated.

Analyzing the patterned arrays in the aforementioned manner yielded the results shown in figure 4.2. It is clear that different regimes in the parameter space favors different numbers of nanowires per hole. The lower end of the parameter space favors single nanowires. As patterning diameter and fluence are increased, a regime is encountered where the growth of 2D-crystals is favored over nanowire growth, leading to a high percentage of holes containing no nanowires. Further increasing patterning diameter and fluence will lead to a high yield of double nanowires, followed by an increasing number of nanowires per hole as the parameters increase further. This is to be expected, as milling larger holes allows room for more catalyst droplets to form inside each hole, without merging with other droplets inside the same hole, thus allowing for the growth of multiple nanowires in close proximity.

### 5.2.2 Droplet size

Droplet size measurements show a clear difference in median droplet size between the arrays in the lower end of the parameter space, and the rest of the arrays. Comparing the median diameter plot in figure 4.3(a) with the plot of single nanowire yield in figure 4.2(b), it is clear that the arrays with a high yield of single nanowires also are the ones with higher median droplet diameter. Looking at the droplet size distributions for each array, we see that for these arrays, most droplets fall within a narrow peak of a diameter between about 220 nm and 250 nm, as exemplified in figure 4.3(b). The other arrays have broader droplet diameter distributions, as exemplified in figure 4.3(b), with diameters reaching from below 100 nm to above 200 nm.

This indicates that single nanowires undisturbed by other nanowires or 2D-crystals in their neighborhood develop large catalyst droplets, due to their plentiful access to Ga. Their droplet size is only limited by the maximum contact angle between the droplet and nanowire, and most droplets come close to this limit, leading to a narrow size distribution.

As patterning parameter change to accommodate 2D-crystals and multiple nanowires per hole, Ga is sparse, as it is used to grow many structures in a small space. Thus the catalyst

droplets are not provided with enough Ga to reach their maximum size, and they become smaller. The size distribution widens as droplets are no longer restricted by a hard limit, but rather limited by local conditions which might vary from wire to wire.

The histograms plotted in figure 5.1(a) confirm that the larger droplets are found on single nanowires, as the size distribution for single nanowire droplets show the aforementioned narrow peak at high diameters, whereas this peak is absent for the droplet size distributions for clustered nanowires. Interestingly, this plot also shows that droplet diameter does not seem to further decrease for holes with 3 or more nanowires. This might be due to the droplet size being just as affected by 2D-growth, which is not represented in the plot.

### 5.2.3 Displacements from lattice

The analysis of displacements form perfect lattice is something that can only be done using computerized analysis to gain accurate knowledge about the location of the lattice point, and the nanowire's displacement from it. This novel technique makes it possible to objectively quantify the regularity of positioning of the nanowires, which allows for optimization of the growth process to achieve maximum regularity. This is desirable as many of the applications for nanowire require a regular pattern, and a well defined pitch is often necessary for the nanowire array to obtain the desired properties.

From the displacement scatter plots shown in figure 4.5, it is clear that the displacement increases with increasing fluence and patterning diameter. Increasing these parameters leads to larger holes in the oxide film, and thus a larger area from which the nanowires can grow, which explains their larger displacement. For most arrays the scatter plots clearly show that there are very few nanowires with close to zero displacements. This shows that nanowires tend to grow along edges of holes. As a catalyst droplet along the hole edge can contact both the hole bottom and the hole wall, it will have a greater contact area with the substrate, and thus have lower energy than if it were situated elsewhere. Thus the droplets tend to end up along the edges of holes, and initiate nanowire growth there.

Given that the displacements correspond to the size of the holes, it is also apparent that the hole diameter is not simply a function of patterning diameter. While the intention is to vary hole depth by varying fluence, and to vary hole diameter by varying patterning diameter, we see that the hole diameter is also affected by varying the fluence. For low patterning diameter, this effect is not very pronounced, but for the higher patterning diameters, it is highly visible. For a patterning diameter of 80 nm, increasing the fluence from 0.06 nC/μm$^2$ to 0.53 nC/μm$^2$ increases the median displacement from around 100 nm to around 200 nm. This can be explained by the Gaussian profile of the ion beam. A patterning diameter of 80 nm means that most of the beam is within this area, but the tails of the Gaussian beam reach beyond this area. When increasing fluence, the amount of ions across the whole distribution increases, and a larger area gains a sufficient ion dose to reach the Si after the subsequent etching.

Some of the displacement distributions have a tail going down and to the right. This tail is likely due to the fact that the blanking of the ion beam is not instantaneous, and ions continue to be emitted as the beam moves to pattern the next hole, causing an unintended groove in the substrate along the path of he ion beam. This groove can be seen on some of the SEM images. The scatter plots show that nanowires occasionally nucleate in these grooves, especially in the arrays milled with higher fluence. This is to be expected, as the higher fluence is attained by milling the pattern many times over, something which would deepen the groove, allowing it to reach down to the Si. The presence of these grooves is unfortunate, as it reduces the accuracy of the nanowire positioning. This problem can be mitigated by using an instrument with faster beam blanking, or using alternative patterning techniques.

### 5.2.4 Optimal process parameters

The data gathered from the computer analysis of the FIB patterned arrays enables an evaluation of the nanowire growth conditions under varying patterning parameters. For most nanowire applications the desired outcome is an array with a high yield of highly uniform single nanowires. The plot shown in figure 4.2(b) makes it easily apparent what combinations of patterning diameter and fluence is best for achieving single nanowires. Two of the

arrays have a particularly high yield: array 6, with a yield of 84.1 %, and array 17, with a yield of 83.0 %. To select the best parameter combination, one could look at the other data available. Array 17 has a slightly lower median displacement magnitude than array 6 (75 nm vs 82 nm), but the median displacement of array 6 is heightened by the tail coming from insufficient beam blanking, and would be lower if this was mitigated. Looking at the droplet size histograms, array 17 seems to have a larger portion of the detected droplets fall within the high diameter peak, indicating a higher amount of well formed nanowires. The arrays with a high yield of single nanowires are on the edge of the observed parameter space. Thus it could also be useful to perform a similar study with arrays patterned with even lower diameter and fluence, to see if this would further increase single nanowire yield.

## 5.3 Large NIL array

The ability to detect and analyze nanowire growth across large areas, such as those present in the tiled NIL datasets, presents an opportunity to acquire highly accurate insight into nanowire growth from a large sample size. Being able to count and measure thousands of nanowires (more than 50 000 for the largest dataset) enables the acquisition of accurate and detailed data on nanowire yield, droplet size and nanowire displacement from the intended ideal lattice.

### 5.3.1 Yields

The calculated yield numbers were slightly different for the two NIL datasets (see table 4.1). Dataset 2 had slightly less single nanowires than dataset 1, and slightly more both empty lattice points and double nanowires. As both datasets are taken from the same sample, no difference in yields is expected. The observed difference might be the result of differences in nanowire detection between the two datasets. However, if nanowires in dataset 2 were more easily detected, one would not expect to see an increase in empty lattice points. Conversely,

if they were harder to detect, one would not see an increase of double nanowires. Thus, the effects at play must be more complex.

Excluding the anomalous nanowire-free regions when calculating yield did not have a substantial influence. The calculated yield for the datasets increased by only 0.4 % and 1.1% respectively, when using subregions of the image without empty areas. This indicates that the yield is very homogeneous across the sample, and that the nanowire free spots do not have a substantial influence on the overall yield.

### 5.3.2 Droplet size and displacements from lattice

The droplet size distribution of the two datasets seems to be similar in shape, but shifted towards somewhat lower sizes for dataset 2. This is due to the fact that only the inner part of the droplets were detected on the second dataset, leading to a constant underestimation of droplet size. In reality, the two datasets likely have very similar size distributions, as they were acquired from the same sample.

Both datasets display a bimodal droplet size distribution. Looking at the histogram in figure 5.1(a), there is one lower diameter peak which is smaller, and more spread out (peak i), and another peak at a higher diameter which is taller and narrower. Looking at the images, and marking droplets with diameters falling within either peak reveals that peak ii represents well formed single nanowires with no 2D-crystals or other nanowires within its domain, whereas peak i represents nanowires sharing their hole with either other nanowires or 2D-crystals. This is consistent with what was found for the matrix of arrays, where single nanowires were shown to have larger droplets with a narrower size distribution.

Looking at the displacement distributions, the one of the first dataset (figure 4.12(a)) was fairly featureless, whereas the one of the second dataset (figure 4.14(a)) was more interesting. Most nanowires were located within 125 nm of the ideal lattice position, with increasing density closer to the center. However, a significant portion of the nanowires were displaced between 125 nm and 250 nm from their ideal lattice position, and these nanowires displayed

a threefold symmetry in their displacement, tending towards displacement towards one of three angles.

To investigate this further, scatter plots were made of the displacements of only the nanowires within certain droplet diameter ranges, more specifically, nanowires whose droplet diameters fall within the two aforementioned peaks. The displacements of nanowires from peak i (figure 5.1(c)) are generally within 125 nm of the ideal lattice point, wit a few outliers. The nanowires with lower diameters however, are found to be the ones with larger displacements,

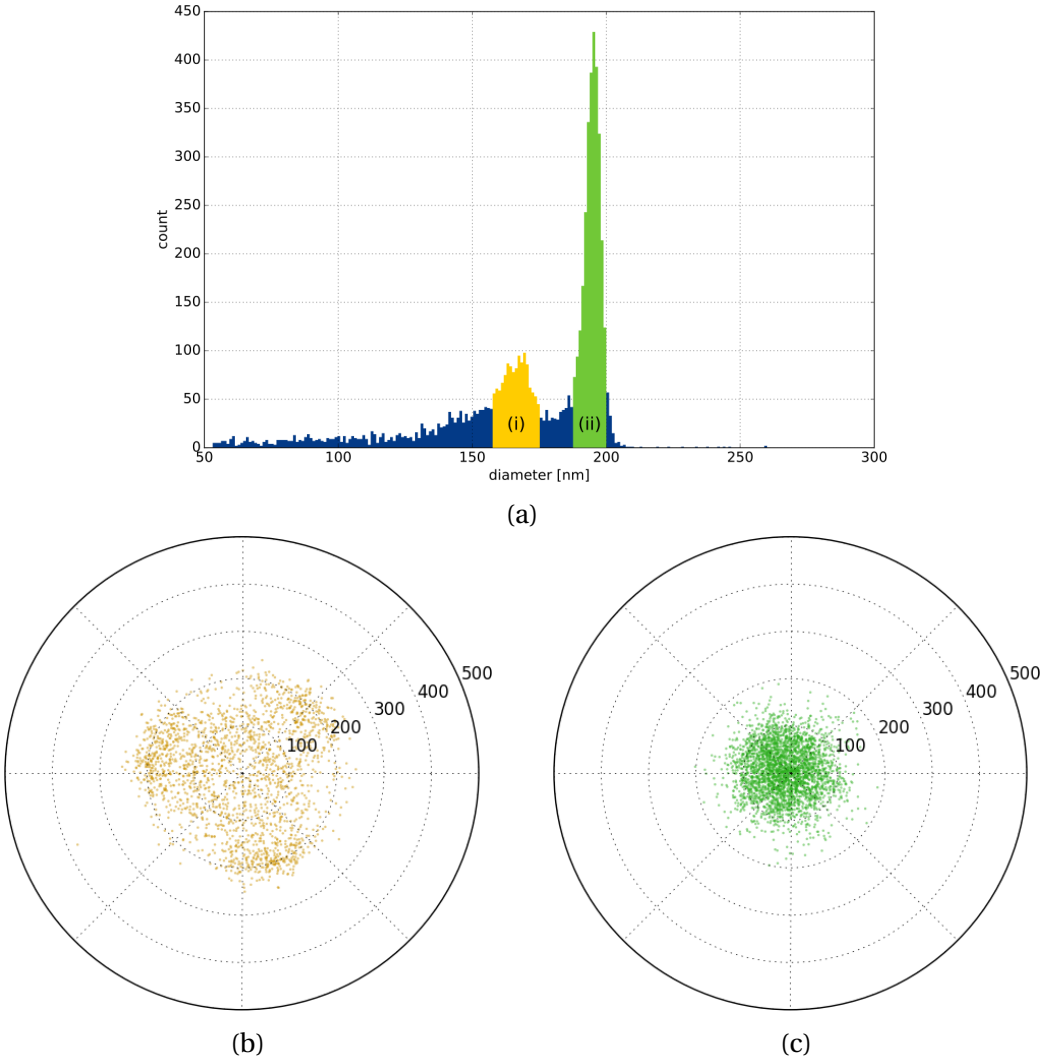



Figure 5.1: (a) Histogram of detected droplet diameters for NIL dataset 2. Two peaks are marked. (b) and (c) Scatter plots of the displacements from perfect lattice for the droplets found in peak (i) and (ii) in (a) respectively. Distances given in nm.

and displaying the threefold symmetry, as is shown in figure 5.1(b). This coincides with what is seen in figure 5.2, where there seems to be to types of nanowires: ones with larger droplets and smaller displacement, and ones with smaller droplets and larger displacements.

This shows that the presence of multiple nanowires or 2D-crystals tends to not only reduce droplet diameter, but also displace nanowires, preferentially towards one of three angles, with threefold symmetry. Both the Si substrate and the GaAsSb nanowires have crystal structures with threefold symmetry, and the Si substrate is cut along its (111) plane, which is the plane along which threefold symmetry is found. This is believed to be the cause of the threefold symmetry in the displacement.

Another interesting observation is that for both datasets the displacement distributions have a higher density for lower displacements, with many nanowires placed very close to ideal lattice position. This contrasts the nanowires in the FIB matrix, where few nanowires were located near the center of the holes, and nucleation was preferred along hole edges. The

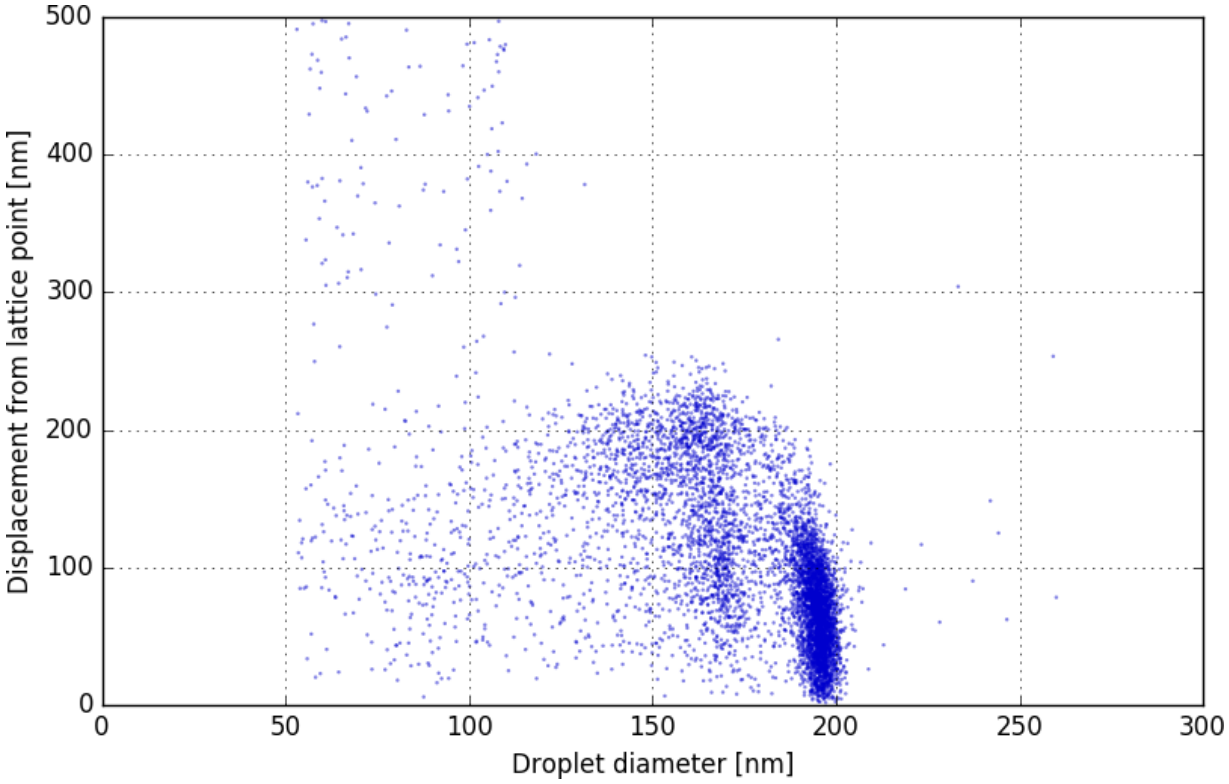



Figure 5.2: Scatter plot of diameter and displacement for all droplets on NIL dataset 2.

nanowires growing in the NIL array do not seem to preferentially nucleate along edges. This could be an effect of inaccuracies in the lattice definition blurring the displacement distribution enough to hide the dip towards the center, or it could be a physical effect.

Lastly, it is apparent through figures 4.9(b) and 4.10(b) that droplets on nanowires near empty areas tend to be slightly larger than normal. Since nothing in the empty areas consumes the Ga that hits the surface, it will end up at nanowires near these areas. Since they are supplied with Ga from a larger area, they have a larger supply, which allows their catalyst droplets to grow larger. This shows that most nanowires in the array have not reached the upper limit of catalyst droplet size, as the droplet can be further enlarged by additional supply of Ga.

**Image stitching**

Both NIL datasets were acquired from the same sample. Still, the data pertaining to nanowire displacement from the ideal lattice was substantially different between the two datasets. For the first dataset (see figure 4.11), the displacement magnitude was generally larger towards the left and right edges of the image, and along the vertical middle part of the image, with smaller displacements for nanowires located between these areas. Different regions of the image seemed to have certain preferred displacement angles, with a majority of nanowires in these regions displaced towards the preferred angle. This preferred angle varied between regions of the image. The displacement data for the second dataset (figure 4.13) showed none of these properties. Displacement magnitude was consistently low for most nanowires, and displacement angle seemed to vary randomly from wire to wire, with no correlation to that of nearby wires.

These observations indicate that the nanowire lattice displayed in dataset 1 was not a linear lattice, but rather a distorted, non-linear lattice. As the sample was patterned with a regular lattice through NIL, this is likely an effect of acquisition and processing, not an actual property of the sample. The fact that dataset 2 displayed none of the same properties, while the two datasets were taken from the same sample, further supports the notion that the lattice nonlinearity of dataset 1 was due to errors in acquisition.

The cause of the effects discussed above is believed to be errors in the stitching of the images making up dataset 1. The complete dataset was put together by stitching $10 \times 10$ smaller images into one large image covering the entire imaged region. This process was carried out automatically by software on the SEM used for acquisition. Several observations support the notion that erroneous stitching is the cause of the lattice distortions.

Observation of the images shows features like the ones shown in figure 5.3. This kind of artifact appears when attempting to stitch two images without proper alignment. One image transitions into the other, and the features do not overlap. These features were sometimes picked up by the detection algorithm, and are shown in figure 4.9(c). The figure shows that these features appear in lines, which correspond to the borders between the stitched images.

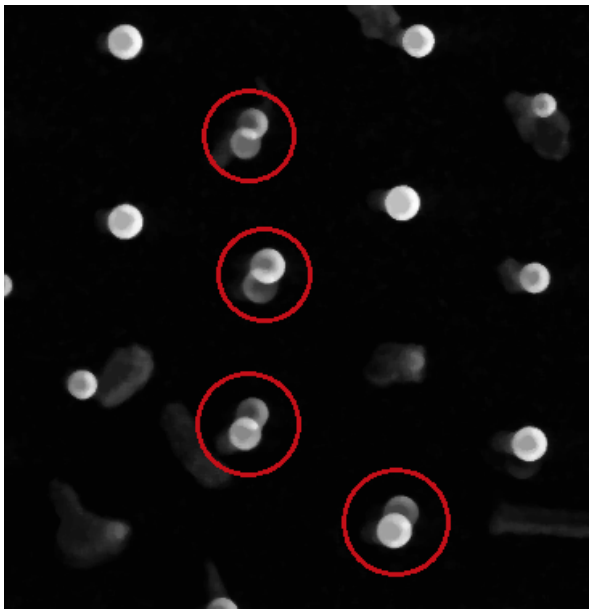


Figure 5.3: Section of dataset 1 showing stitching error artifacts (circled in red)

The displacement magnitude and angle maps for dataset 1 also shows abrupt changes across lines in the image forming a grid-like pattern corresponding with the stitching boundaries. This indicates that stitching errors causes the lattice of the individual stitched images to be improperly aligned. When the ideal lattice fitted to the detected droplets is a continuous lattice, whereas when each stitched image is displaced with respect to the others, the lattice of each sub-image will not match the overall lattice. This is why we get the variations in displacement angle and magnitude present in dataset 1.

As discussed earlier, the displacement scatter plot for dataset 2, shown in figure 4.14(a), shows interesting features such as a high density center, and a threefold symmetry of likely displacement angles. This is not visible in the corresponding plot for dataset 1, figure 4.12(a).

The displacements for dataset 1 are dominated by the effects of lattice mismatch due to stitching errors, and interesting features are thus obscured by noise.

The fact that stitching errors were only observed for the first dataset demonstrates that such errors can be avoided using the correct imaging conditions. The imaging conditions for dataset 1 makes stitching more difficult. This could be mitigated by choosing suitable imaging conditions, by correcting for distortions in the image digitally before stitching, or by utilizing more advanced stitching methods able to detect and correct nonlinear distortions between images.

## 5.4 Random growth area

The random growth dataset shows large amounts of nanowire growth in a region reaching beyond the squares patterned with the FIB. While some nanowires grow within the patterned squares, the squares are mostly dominated by the growth of 2D crystals. Most nanowires grow along the edges of the squares and in the region surrounding the squares. This can be explained by the fact that the ion beam has a Gaussian profile, causing areas surrounding the patterned squares, where the core of the ion beam is directed, to still be exposed to ions, albeit in a smaller dose. The area with nanowire growth extends further from the squares as the patterning fluence increases. This is to be expected, as an increasing overall fluence also will increase the number of ions in the tails of the Gaussian beam. This leads to an expansion of the area where the received ion dose is large enough to enable nanowire growth.

The density plot shown in figure 4.17(a) shows that the nanowire density is low within the patterned squares. An exception is the first square, which has a high density of detected droplets. Further inspection of the images reveals that these are droplets on the surface without nanowires. The area surrounding the squares has a higher density of nanowires. This indicates that the ion dose received within the squares is higher than the ideal conditions for random nanowire growth, whereas the dose in the surrounding area is more suitable.

The areas with the highest nanowire density lie along the edges of the nanowire growth area. This could be due to the edges having greater access to Ga moving along the surface. However, the area between squares 2 and 3 also shows a high density of nanowires, even though it is not near an edge. This indicates that the high nanowire density is not due to edge effects. Rather, it seems to indicate that the nanowire density is dependent on the ion dose received, and that the lower ion dose experienced further from the patterned squares is ideal for maximizing random nanowire growth.

Comparing the droplet diameter plot in figure 4.17(b) to the density plot in figure 4.17(a) allows us to draw several conclusions about droplet size:

- Droplet size inside the first square, and outside the nanowire growth area, is very small: Droplets on the substrate surface are much smaller than droplets on nanowires.
- Droplet size inside the other squares is also small: The high amount of 2D-crystals growing within the squares consumes Ga used for droplets, causing the droplets in the area to be smaller. This is consistent with what has been found for the other samples in this study.
- In the areas with high nanowire density, the droplet size is lower. This includes the area between square 2 and 3: A higher density of nanowires also causes less Ga to be available for the droplets on each wire, leading to smaller droplets.
- At the very edge of the nanowire growth area, the droplets are larger: Droplets along the edge have access to Ga from the empty area, as these areas contain no growing features consuming Ga.

The random growth dataset shows that a low fluence ion beam replicating the dose received along the edges of the area where nanowire growth was observed is ideal for inducing high density unpatterned nanowire growth on a $Si/SiO_x$ substrate. It also shows that the ion beam affects the properties of the sample with respect to nanowire growth in areas far from the designated patterned area. Thus when using FIB to pattern samples for nanowire growth, the

properties of a feature can not be assumed to be independent from the patterning performed in its vicinity.

# Chapter 6

# Conclusions

## Developed routines

In this work, computational routines have been developed to analyze SEM images of nanowires, and gather growth related data such as nanowire count, yield, catalyst droplet size, and positional deviation from the desired pattern.

Using computer vision techniques, nanowire catalyst droplets can be detected in top-down SEM images, yielding the position and size of each droplet. Imaging conditions influence the detection accuracy, but images can be optimized for detection by preprocessing using median filtering and dilation based reconstruction. A lattice can be numerically fitted to the detected nanowire positions, and used to group detected wires for yield calculations, or analyze positional uniformity, and find patterns in where nanowires grow in relation to the patterned holes.

The developed routines are able to analyze and compare nanowire arrays patterned with different process parameters, analyze large datasets containing more than 50 000 nanowires, and characterize random growth. Large amounts of data can be obtained in an efficient, objective, and reproducible manner. The obtained data can be presented in ways that are easy to interpret, and enables the identification of growth related trends and effects of process

parameters, some of which could not be deduced manually. This facilitates further optimization and upscaling of nanowire growth processes.

## FIB arrays

Using the developed routines, a FIB milled matrix of nanowire growth arrays patterned with varying hole patterning diameter and ion fluence has been analyzed to gain insight into the effect these parameters have on nanowire growth. It has been found that by varying the aforementioned parameters, one can obtain either a high yield of single nanowires, 2D-crystals with low amounts of vertical nanowire growth, or growth of two or more nanowires per milled hole. The highest yield of single nanowires were achieved using a patterning diameter of 10 nm and a fluence of 2 500 ions/nm$^2$ (84.1 % yield), or a patterning diameter of 30 nm and a fluence of 400 ions/nm$^2$ (83.0 % yield).

Analysis of the droplet diameters showed that single nanowires without nearby 2D-crystals had a low variation in droplet diameter, and a mean droplet diameter of around 240 nm, while nanowires growing alongside other nanowires or 2D-crystals had smaller droplets with a wider range of sizes, from around 100 nm to around 230 nm. Analysis of the positional deviation from the ideal lattice showed that nanowires tend to nucleate along the edges of holes, and that hole sizes increase with increasing fluence and patterning diameter. This analysis also showed that the usage of FIB milling might create grooves as the beam moves from patterning one hole to another, especially pronounced when patterning with high fluence. This is likely due to slow beam blanking Nanowires will nucleate in these grooves, leading to less positional accuracy.

## Large NIL array

Two large datasets taken from the same NIL patterned nanowire growth sample have been analyzed using the developed routines. The datasets differ in size and SEM imaging condi-

tions. They consist of several SEM images stitched together, showing in total > 50 000 and > 6 000 nanowires respectively. For both datasets, the detected droplets had a diameter distribution with two peaks. The first dataset had a sharp peak centered around 230 nm, and a wider but smaller peak at 196 nm. From the second dataset a similar diameter distribution was found, but shifted slightly towards lower diameters. This was due to the lower image quality causing the detection routine to consistently report lower droplet diameters. Inspection of the images with overlaid detection data revealed that the high diameter peak represented single nanowires without nearby 2D-crystals, whereas the low diameter peak represented nanowires with other growth at the same lattice point.

Despite the images having good contrast, displacement analysis of the first dataset failed to show any interesting features. This was to errors in the stitching of the separate images making up the dataset. The misaligned image stitching lead to an inconsistent lattice in the combined image, and mismatches with the ideal fitted lattice dominated the displacement data. The utilized stitching routine or the data acquisition need to be refined to avoid this issue.

Displacement analysis of the second dataset found that most nanowires were growing within 125 nm of ideal lattice positions, while the nanowires displaced further displayed a threefold symmetry in their displacement. The threefold symmetry in displacement was only exhibited by wires with lower diameter droplets, i.e. the wires where either other nanowires, or 2D-crystals, are present at the same lattice site. This symmetry may be related to the threefold symmetry of the nanowires or growth substrate.

## Random growth area

Using the developed routines to analyze a dataset showing unpatterned nanowire growth in and surrounding FIB patterned squares revealed the relationships between ion exposure of the substrate, nanowire growth density, and nanowire droplet size. Areas not directly patterned by the FIB, but only exposed to stray ions, i.e. areas exposed to a low effective fluence,

were shown to provide good conditions for high density non-position controlled nanowire growth. Nanowire density was shown to be highest along the edges of the area exposed to stray radiation. Catalyst droplets are found to be smaller in areas with high nanowire density, or high amounts of 2D-crystals, and larger near the edge of the nanowire growth area.

# Chapter 7

# Recommendations for Further Work

While the work done in this thesis has come a long way in creating objective and efficient methods for analysis of nanowire growth, much can still be gained from further development of these kinds of techniques. While the nanowire detection methods used in this study were able to gather useful data in several ways, there is still room for improvement. By either fine tuning the parameters of the current method, or implementing other methods of detection, the accuracy could be increased to further avoid false positives and negatives, and to more consistently report an accurate size for the detected droplets. The accuracy of detection could be quantified using either fabricated or manually measured datasets, where position and size of all nanowire droplets is already known, and comparing with the detection results. The detection routine could also be further developed to accurately detect features such as nanowires without catalyst droplets, or 2D-crystals.

The application of computer vision techniques to characterize and analyze nanowire growth is not limited to top-down images. If more advanced computer vision techniques were employed, side-view images could be analyzed to measure nanowire length and thickness, and the contact angle between the nanowire and catalyst droplet. These are important parameters for the properties of nanowires, which can not be measured from top-down images. Data gathered from the analysis of side-view images, along with other data such as photoluminescence characterization data and electrical measurements, should be combined with the data

obtained from top-down images, and further processed to produce useful visualizations and insights.

The routines developed in this thesis are currently accessible only through Python scripts calling the functions defined within the code, and providing the required parameters. Although the code is well documented to aid unfamiliar users, and examples of scripts performing useful analysis are included, the usage could be more intuitive. To encourage widespread usage of the developed routines by nanowire growers throughout the scientific community, an intuitive user interface could be developed. Researchers unfamiliar with the code should be able to utilize the developed routines simply by following provided instructions and providing the requested input. To more easily explore obtained data, an interactive visualization interface could be implemented, allowing the user to dynamically limit the set of nanowires for which to visualize a certain property by selecting a region or range of values from plots of other properties. This would ease the discovery of otherwise hard to spot connections or trends.

To avoid issues with stitching errors, as in this study with NIL dataset 1, a better stitching procedure could be developed. The displacement data provided by the already developed routines could be used to evaluate the quality of stitching, and gather data on misplacements of sub-images, which could be fed back to the stitching routine to repeat stitching with the added corrections. This could be repeated until displacement measures showed a good fit.

Analysis of the FIB milled arrays showed that the optimal conditions for patterned single nanowire growth were along the edges of the chosen parameter space. To further optimize milling parameters for FIB patterning of nanowire growth arrays, a new study could be conducted, using the same techniques on an array matrix using different values of fluence and diameter. The values should be chosen so as to explore parts of the parameter space surrounding the arrays shown to have the best conditions for single nanowire growth, but reaching beyond the already studied parameter space, or with smaller differences between each array, so as to explore the parameter space with higher resolution.

The study of non position controlled nanowire growth demonstrated that exposing a $SiO/SiO_x$ substrate to a low fluence ion beam generates good conditions for high density random nanowire growth. To explore this further, a sample patterned with a range of low ion fluences could be made, and analyzed with the developed routines. The generated density plots would clearly show what level of ion fluence generates the best conditions for high density nanowire growth.

# Bibliography


[1] Peidong Yang, Ruoxue Yan, and Melissa Fardy. Semiconductor nanowire: What's next? *Nano Letters*, 10(5):1529–1536, may 2010.

[2] Gaute Otnes and Magnus T. Borgström. Towards high efficiency nanowire solar cells. *Nano Today*, nov 2016.

[3] Jelena Vukajlovic-Plestina, Wonjong Kim, Vladimir G. Dubrovski, Gözde Tütüncüoğlu, Maxime Lagier, Heidi Potts, Martin Friedl, and Anna Fontcuberta i Morral. Engineering the size distributions of ordered GaAs nanowires on silicon. *Nano Letters*, jun 2017.

[4] R.F. Egerton. *Physical Principles of Electron Microscopy*. Springer Nature, 2016.

[5] Lucille A. Giannuzzi and Fred A. Stevie, editors. *Introduction to Focused Ion Beams*. Springer-Verlag GmbH, 2004.

[6] R. S. Wagner and W. C. Ellis. Vapor-liquid-solid mechanism of single crystal growth. *Applied Physics Letters*, 4(5):89–90, mar 1964.

[7] Yewu Wang, Volker Schmidt, Stephan Senz, and Ulrich Gösele. Epitaxial growth of silicon nanowires using an aluminium catalyst. *Nature Nanotechnology*, 1(3):186–189, nov 2006.

[8] R. L. Barns and W. C. Ellis. Whisker crystals of gallium arsenide and gallium phosphide grown by the vapor—liquid—solid mechanism. *Journal of Applied Physics*, 36(7):2296–2301, jul 1965.

[9] Michael Quirk and Julian Serda. *Semiconductor Manufacturing Technology*. Prentice Hall, 2000.

[10] T. Sato, K. Hiruma, M. Shirai, K. Tominaga, K. Haraguchi, T. Katsuyama, and T. Shimada. Site-controlled growth of nanowhiskers. *Applied Physics Letters*, 66(2):159–161, jan 1995.

[11] Richard Szeliski. *Computer Vision*. Springer-Verlag GmbH, 2010.

[12] Richard E. Woods Rafael C. Gonzalez. *Digital Image Processing*. Prentice Hall, 2007.

[13] RM Hodgson, DG Bailey, MJ Naylor, ALM Ng, and SJ McNeill. Properties, implementations and applications of rank filters. *Image and Vision Computing*, 3(1):3–14, feb 1985.

[14] A C Bovik, T S Huang, and D C Munson. The effect of median filtering on edge estimation and detection. *IEEE transactions on pattern analysis and machine intelligence*, 9:181–194, February 1987.

[15] E.R Davies. Edge location shifts produced by median filters: theoretical bounds and experimental results. *Signal Processing*, 16(2):83–96, feb 1989.

[16] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[17] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[18] G. Bradski. The opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, jan 1965.

[20] Dingding Ren, Dasa L. Dheeraj, Chengjun Jin, Julie S. Nilsen, Junghwan Huh, Johannes F. Reinertsen, A. Mazid Munshi, Anders Gustafsson, Antonius T. J. van Helvoort, Helge Weman, and Bjørn-Ove Fimland. New insights into the origins of sb-induced effects on self-catalyzed GaAsSb nanowire arrays. *Nano Letters*, 16(2):1201–1209, feb 2016.

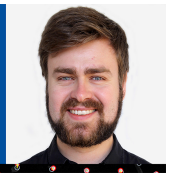
# Appendix A

# Poster, Nanowire Week

The following poster, presenting part of the work done for this Master's thesis, was presented at the conferences Nanowire Week 29 May-2 June, Lund, Sweden and EMAG, 3-6 July, Manchester, UK.

# Evaluating FIB patterning for nanowire growth

Aleksander B. Mosberg[1], Steinar Myklebost[1], Dingding Ren[2], Helge Weman[2], Bjørn-Ove Fimland[2], Antonius T. J. van Helvoort[1]
[1]Department of Physics & [2]Department of Electronic Systems,
Norwegian University of Science & Technology (NTNU), NO-7491, Trondheim, Norway



## Introduction

**To realize large arrays of uniform semiconductor nanowires for devices[1], growth optimization work must study objectively a significant number of wires.**

**By studying FIB and NIL patterning after nanowire growth, this work demonstrates automated nanowire detection and objective, statistical analysis for further growth optimization.**

## Samples and detection

To explore the ion fluence - hole diameter parameter space of focused ion beam (FIB) milling for nanowire growth, hole arrays with varying parameters and random growth areas were milled (Fig. 1(a)). The resulting self-catalyzed GaAsSb[2] wire arrays (Fig. 1(b)) were imaged in SEM and feature detection (Fig. 1(c)) used on Ga droplets to find and characterize all wires.

To demonstrate scalability, 0.27 x 0.18 mm with over 50 000 nanowires was imaged and characterized from a nanoimprint lithograph (NIL) patterned sample[3].

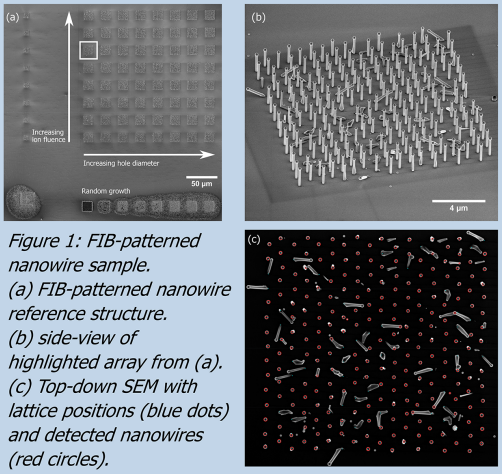



*Figure 1: FIB-patterned nanowire sample. (a) FIB-patterned nanowire reference structure. (b) side-view of highlighted array from (a). (c) Top-down SEM with lattice positions (blue dots) and detected nanowires (red circles).*

## Characterizing growth

By summing up detected Ga droplets per FIB-milled array the total yield can be calculated on an array-to-array basis (Fig. 2(a)), from which the optimal milling parameters can be deduced. As some holes have more than one nanowire, the yield of wires per hole must be specified. By assigning each wire to the nearest hole, the yield of single wires per hole is plotted in Fig. 2(b).

For randomly grown nanowires, density of wires per area can be shown to increase at edges (Fig. 2(c)), showing the effect of Ga surface diffusion on yield.

The distribution of droplet size can be analysed, shown in Fig. 2(d) for a large area (NIL). Deviations from averaged lattice position (of a fitted lattice) can also be quantified and visualized (poster background).

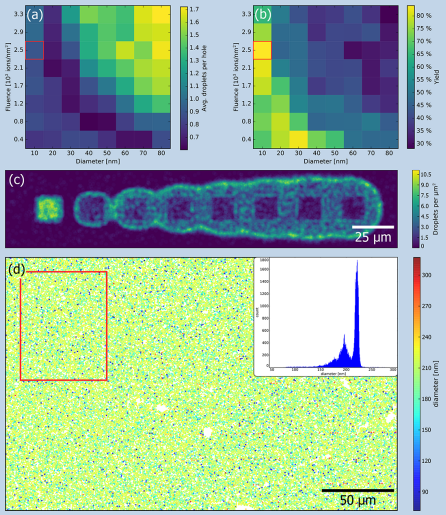



*Figure 2: (a) Nanowire yield per array of FIB-milled sample, normalized over number of milled holes. (Fig. 1(b) in red). (b) Yield of single nanowires per fitted lattice point. (c) Nanowire density map of random growth area. (d) Diameter map & histogram of NIL sample (poster background in red).*

*Background: Nanowires (red) from NIL-patterned sample with displacement (yellow) from fitted lattice (blue).*

**References**
[1] Y. Zhang *et al.*, J. Phys. D: Appl. Phys. **48**, 463001 (2015).
[2] D. Ren *et al.*, Nano Lett. **16**, 1201–1209 (2016).
[3] A. M. Munshi *et al.*, Nano Lett. **14,** 960–966 (2014).

*Source code and demonstration: bit.ly/nwstats*

**Acknowledgements**
This work is supported by the Research Council of Norway through funding for the NorFab (197411) facility and the FRINATEK (214235) program.

## Conclusions

- **Novel FIB patterned nanowire arrays**
- **Objective nanowire growth evaluation**
- **Representative growth statistics**

NTNU NanoLab

Contact: aleksander.b.mosberg@ntnu.no

NTNU

# Appendix B

# Conference paper, EMAG

The following paper, presenting part of the work done for this Master's thesis, has been submitted and accepted for the 2017 EMAG conference, and will be published in the *Journal of Physics: conference series*.

# Evaluating focused ion beam patterning for position-controlled nanowire growth using computer vision

**A B Mosberg[1], S Myklebost[1], D Ren[2], H Weman[2], B O Fimland[2] and A T J van Helvoort[1]**

[1] Department of Physics & [2] Department of Electronic Systems, Norwegian University of Science and Technology (NTNU), 7491 Trondheim, Norway

E-mail: aleksander.b.mosberg@ntnu.no

**Abstract.** To efficiently evaluate the novel approach of focused ion beam (FIB) direct patterning of substrates for nanowire growth, a reference matrix of hole arrays has been used to study the effect of ion fluence and hole diameter on nanowire growth. Self-catalyzed GaAsSb nanowires were grown using molecular beam epitaxy and studied by scanning electron microscopy (SEM). To ensure an objective analysis, SEM images were analyzed with computer vision to automatically identify nanowires and characterize each array. It is shown that FIB milling parameters can be used to control the nanowire growth. Lower ion fluence and smaller diameter holes result in a higher yield (up to 83 %) of single vertical nanowires, while higher fluence and hole diameter exhibit a regime of multiple nanowires. The catalyst size distribution and placement uniformity of vertical nanowires is best for low-value parameter combinations, indicating how to improve the FIB parameters for positioned-controlled nanowire growth.

## 1. Introduction

III-V semiconductor nanowires are a promising material system for the creation of future optoelectronic devices [1]. Using lithography-based patterning of an oxide mask, well-defined nucleation sites are placed at will, often in homogeneous patterns to ensure identical growth conditions. This approach has been successful in achieving large arrays of similar nanowires, but wire-to-wire variations still need to be evaluated [2]. In order to improve nucleation and further reduce variation, direct oxide patterning is expected to be advantageous, allowing for more flexible hole geometry to optimize patterning and nanowire nucleation conditions.

In this work, focused ion beam (FIB) is used to pattern growth substrates for self-catalyzed GaAsSb nanowires grown using molecular beam epitaxy (MBE) [3]. The FIB patterning enables direct patterning of the oxide mask and is more flexible than the lithography based techniques conventionally used. The parameter space to optimize nanowire growth is efficiently explored on a single growth sample. To evaluate the effect of milling conditions on nanowire growth, a sufficient quantity of nanowires need to be characterized in an objective and efficient way. This is especially important for evaluating this rather novel approach to patterning for nanowire growth. By utilizing feature detection techniques from the field of computer vision [4] to automatically detect nanowires from top-down scanning electron microscope (SEM) images, a detailed and objective characterization of the parameter space is achieved.

## 2. Methods and materials

Self-catalyzed GaAsSb nanowires were grown in a Varian GEN II Modular MBE system [5]. To pattern the substrate for position-controlled nanowire growth, a FEI Helios NanoLab 600 DualBeam FIB was used at 30 kV to mill a growth matrix into a Si(111) wafer with a 40 nm thick $SiO_x$ film (Fig. 1(a)). The growth matrix consists of $8 \times 8$ hole arrays with linearly increasing combinations of ion fluence ($0.418 - 3.329\ 10^{17}$ ions/cm$^2$) and hole diameter (10 – 80 nm). Each hole array contains 270 holes in a hexagonal pattern with 1 µm pitch (Fig. 1(b)). The sample was cleaned using 1 % HF for 150 s before insertion in the MBE system.

Each array was imaged with 5 kV SEM in the DualBeam (Fig. 1(c)). Top-down images were then used as input for feature detection, implemented in open source Python libraries. By optimizing the SEM contrast for computer vision, Ga catalyst droplets were identified and used to count and characterize nanowires (Fig. 2(a)). A lattice based on the FIB-milled pattern, fitted to the detected nanowires, assigns each nanowire to a lattice point corresponding to a FIB-milled hole. The Python code for detection and analysis shown has been made freely available [6].

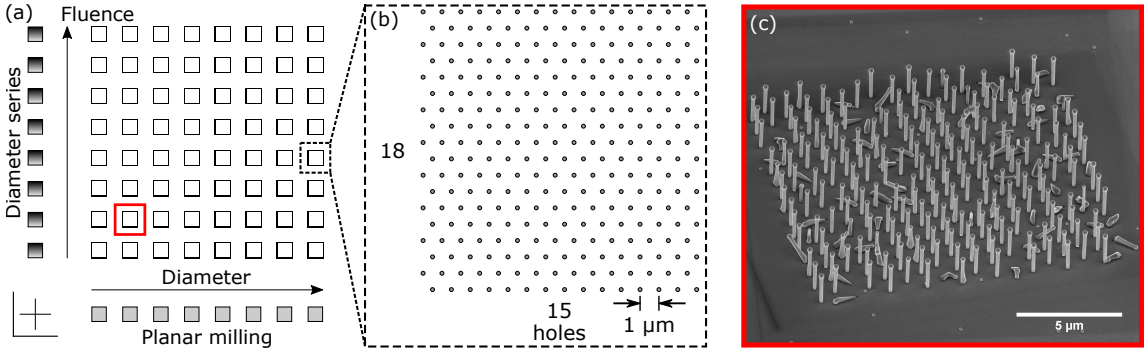



**Figure 1.** (a) Reference design created in FIB: ion fluence - diameter matrix with 64 arrays and supplemental reference fields. (b) Each array consists of $15 \times 18$ holes. (c) Tilted SEM of an array after growth, with a high yield of single vertical nanowires.

## 3. Results and discussion

Vertical nanowire growth is observed in all milled arrays, demonstrating the viability of FIB as an alternative patterning technique for nanowire growth substrates. There is a general trend that smaller, shallower (i.e., lower fluence) holes give a high yield of single vertical nanowires per hole (Fig. 2(a,b)). For larger, deeper holes, multiple vertical nanowires are observed per hole (Fig. 2(d,e)). Between the single and multiple wire regimes, total nanowire yield is lower, dominated by more parasitic growth. This variety demonstrates the necessity of FIB patterning optimization to obtain full growth control among a broad range of possible structures.

The feature detection is consistently able to detect Ga droplets and distinguish them from other features in the SEM image, such as stray Ga droplets on the sample surface (Fig. 2(a,b)). This allows for automatic detection of nanowires giving a quantitative and objective analysis of how the FIB milling parameter space affects nanowire growth. The maximal yield of single nanowires, 83 %, was observed for two arrays at lower parameter combinations (Fig. 2(c)). At higher fluence and diameter, array have more variance with several holes containing one or multiple nanowires. The computer vision-based approach ensures correct characterization and classification of growth regimes. For example, a maximal yield of 35 % for two nanowires per hole is found (Fig. 2(d-f)) in the same array where one of the lowest yields of single nanowires (30 %) is observed. For larger arrays of nanowires, the convenience and reliability of automated over manual characterization becomes more important and ensures reproducibility.

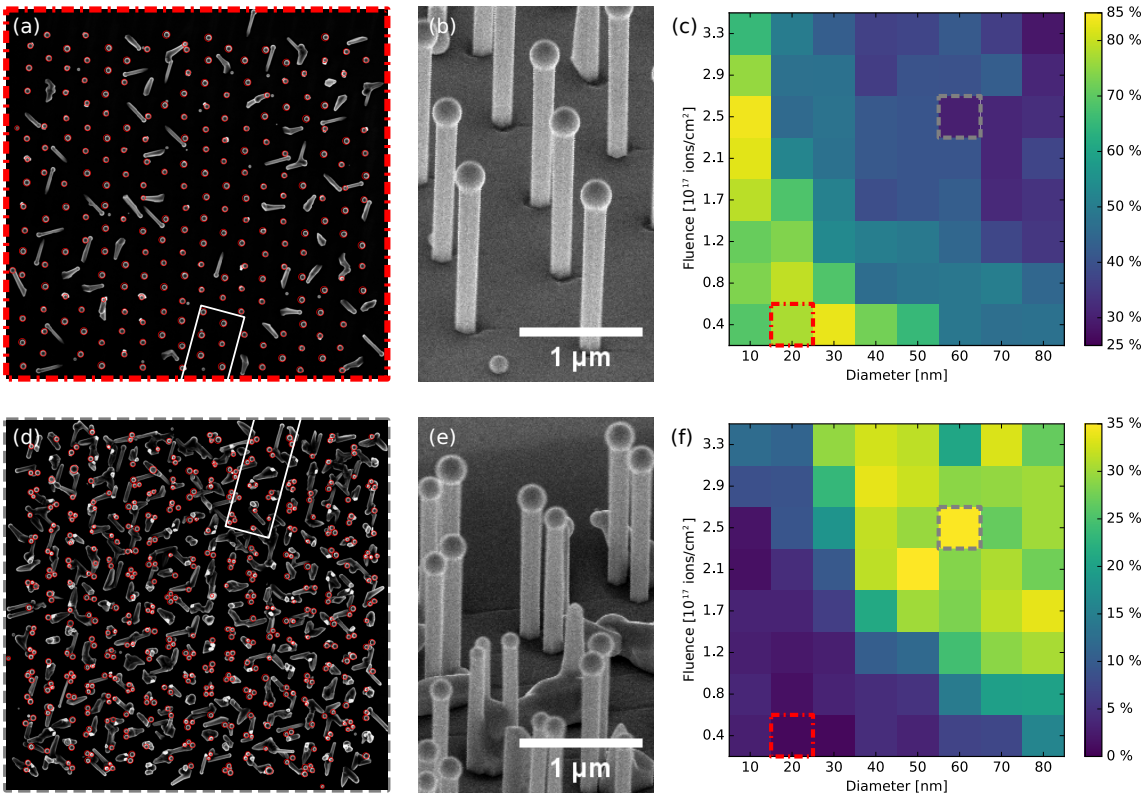



**Figure 2.** (a) Detected droplets overlaid on SEM image for low parameter combination array. (b) Tilted-view SEM image of single nanowire regime, from frame in (a). (c) Single nanowire yield plot across the growth matrix. (d) Detected droplets on SEM image for higher parameter combination. (e) Tilted-view SEM image of multiple-nanowire regime, from frame in (d). (f) Yield plot for two nanowires per hole across the growth matrix.

Taking advantage of the breadth of information provided by computer vision on SEM images, mean droplet diameter and displacement from the fitted lattice positions can be evaluated (Fig. 3(a,b)). The catalyst diameter decreases with increasing ion fluence and hole diameter. This trend can be explained by the additional parasitic growth and multiple nanowires per hole for higher fluence-diameter combinations (Fig. 2(d)). With constant Ga flux across the sample during MBE growth, the Ga supply per wire decreases with increasing number of droplets.

At the same time, the mean deviation from fitted lattice positions increases with both fluence and diameter (Fig. 3(b)). Plotting the deviations in scatter plots (Fig. 3(c), shown for the growth matrix extremes), two distinct effects are identified: First, higher diameter holes consistently have nanowires nucleating further out from lattice centers, indicating that nanowires seem to preferentially nucleate along the hole side walls rather than in the hole center. For larger holes, the increased circumference allows for multiple nucleation sites. Second, deeper (i.e., higher fluence) holes lead to more off-center nucleating nanowires (Fig. 3(c(i))) and less radially symmetric displacement. The off-center cluster of nanowire nucleation is believed to be linked to slow ion beam blanking, resulting in an asymmetric hole edge (visible in Fig. 2(b), blanking lines visible in Fig. 2(e)) leading to an uneven distribution of nanowire nucleation sites within a single hole. This can be remedied by the use of a faster beam blanker or alternative scan strategies when deeper holes are desired.

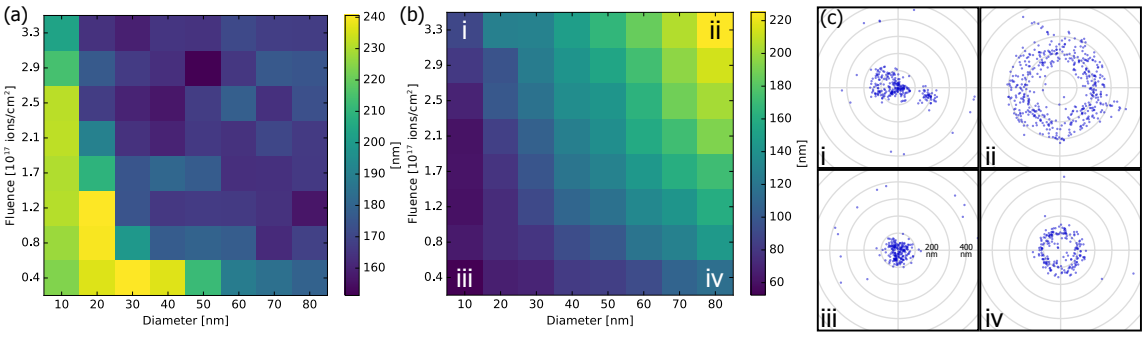



**Figure 3.** (a) Median droplet diameter per array. (b) Median droplet distance from fitted lattice position per array. (c) Scatter plots of droplet displacements from fitted lattice, for the four arrays indicated in (b).

As droplet size and contact angle has been shown to influence both crystallinity and composition [7], the variation in yield, droplet size, and effective Ga/As ratio across the growth matrix is expected to influence the GaAsSb nanowire composition and optoelectronic properties [5]. To further investigate this, micro-photoluminescence spectroscopy, electrical probing of single nanowires, and transmission electron microscopy of the nanowire-substrate interface should be performed for the different arrays and correlated to the results from computer vision-based studies. In this way the trends in growth results can be linked to FIB patterning parameters to systematically study and achieve the optimal properties for nanowire-based devices.

## 4. Conclusion

FIB milling has been systematically studied as a promising and flexible direct patterning method for self-catalyzed nanowire growth substrates. Computer vision was successfully applied to detect Ga droplets on top of vertical nanowires for a substantial number of holes (17 280 holes across 64 different arrays) on a single sample and thereby shed light on how the FIB milling parameters affect nanowire growth. The ion fluence and hole diameter were found to affect vertical nanowire yield, number of nanowires per hole, droplet size distribution, and nanowire displacement from patterned lattice position. These nanowire characteristics can thus be correlated and optimized in future growth trials.

## Acknowledgments

The authors acknowledge NTNU and the Research Council of Norway for financial support via the initiative Enabling technologies: Nanotechnology, NORFAB (grant 197411), NORTEM (grant 197405), and the FRINATEK-program (grant 214235).

## References

[1] Joyce H J *et al.* 2011 *Progress in Quantum Electronics* **35** 23–75
[2] Nilsen J S, Reinertsen J F, Mosberg A, Fauske V T, Munshi A M, Dheeraj D L, Fimland B O, Weman H and van Helvoort A T J 2015 *Journal of Physics: Conference Series* **644** 012007
[3] Detz H, Kriz M, Lancaster S, MacFarland D, Schinnerl M, Zederbauer T, Andrews A M, Schrenk W and Strasser G 2017 *Journal of Vacuum Science & Technology* B **35** 011803
[4] Lindeberg T 1998 *International Journal of Computer Vision* **30** 79–116
[5] Ren D *et al.* 2016 *Nano Letters* **16** 1201–9
[6] NWstats GitHub repository URL `https://www.github.com/NWstats/NWstats`
[7] Munshi A M, Dheeraj D L, Todorovic J, van Helvoort A T J, Weman H and Fimland B O 2013 *Journal of Crystal Growth* **372** 163–9

# Appendix C

# Source code

This appendix presents the source code written in this project. After the summer of 2017, an updated version of the code can be found at https://github.com/nwstats/nwstats

The code is colored according to the following legend.

**Magenta** Keywords

**Blue** Identifiers

**Purple** Strings

**Green** Comments

The following files are included:

**p. 104 - tileset.py**

Defines the `Tileset` class, containing all the code used to deal with the datasets consisting of multiple tiles. In this project, the two NIL patterned datasets, and the random growth area.

**p. 133 - lattice.py**

Defines the `Lattice` class, used for dealing with lattices of a limited size. Used by the `NewField` class.

**p. 134 - arbitraryLattice.py**

Defines the `ArbitraryLattice` class, used for dealing with lattices of arbitrary size. Used by the `Tileset` class.

**p. 137 - detect.py**

Defines the functions used for feature detection.

**p. 140 - functions.py**

Defines a variety of helper functions used elsewhere in the code.

Note: What is referred to as a "matrix of arrays" in the rest of the thesis, is referred to as an "array of fields" in the code.

# newField.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import cv2
4  from scipy import misc
5  import os
6  import pickle
7
8  import functions as f
9  from lattice import Lattice
10 import detect
11
```

```
12 class Field:
13
14     def __init__(self, Na, Nb, path, name, scale, ext='.tif'):
15         self.Na = Na
16         self.Nb = Nb
17         self.number_of_points = Na * Nb
18         self.path = path
19         self.name = name
20         self.scale = scale
21         self.ext = ext
22         self.image_path = path + '/' + name + ext
23         self.blobs_path = path + '/data/' + name + '_blobs.p'
24         self.lattice_path = path + '/data/' + name + '_lattice.p'
25         self.blobs_by_point_path = path + '/data/' + name + '_blobs_by_point.p'
26         self.figure_path = path + '/figures/'
27         self.lattice = None
28         self.blobs = np.array([])
29         self.blobs_by_point = []
30
31         data_dir = path + '/data'
32         if not os.path.exists(data_dir):
33             os.makedirs(data_dir)
34
35     def prepImage(self):
36         """Preprocess the image of the field by applying median filtering"""
37         from scipy.signal import medfilt2d
38         image = misc.imread(self.image_path, flatten=True)
39
40         image = medfilt2d(image, 3)
41
42         path = self.path + '/prep_2/' + self.name + '.png'
43         print(path)
44
45         misc.imsave(path, image)
46         print('Saved image ' + self.name)
47
48     def detectBlobs(self, methods=(detect.droplets,)):
49         """Detect blobs using up to several methods, and store them in self.blobs
50
51             Keyword arguments:
52             methods -- a tuple of methods to use for detecting blobs
53         """
54
55         image = cv2.imread(self.image_path)
56         blobs_a = []
```

```
57          for method in methods:
58              blobs_detected = method(image)
59              blobs_a.append(blobs_detected)
60
61          blobs = np.concatenate(blobs_a)
62
63          self.clearBlobsByPoint()
64
65          self.blobs = blobs
66          pickle.dump(blobs, open(self.blobs_path, 'wb'))
67          print('Blobs detected for field ', self.name, ': ', blobs.shape[0], ' blobs', sep
    ='')
68
69      def getBlobs(self, methods=(detect.droplets,)):
70          """Return all detected blobs for field. Load if possible, detect if necessary."""
71          if self.blobs.shape[0] > 0:
72              return self.blobs
73          else:
74              try:
75                  self.blobs = pickle.load(open(self.blobs_path, 'rb'))
76                  if self.blobs.shape[0] < 1:
77                      print('Loaded blobs, but array was empty. Detecting blobs.')
78                      self.detectBlobs(methods)
79              except FileNotFoundError:
80                  print('Blobs file not found! Detecting blobs.')
81                  self.detectBlobs(methods)
82
83          if self.blobs.shape[0] > 0:
84              return self.blobs
85          else:
86              raise RuntimeError('Not able to obtain blobs!')
87
88      def clearBlobs(self):
89          """Delete all stored and loaded information about blobs for this tile"""
90          self.blobs = np.array([])
91
92          try:
93              os.remove(self.blobs_path)
94          except FileNotFoundError:
95              pass
96
97          self.clearBlobsByPoint()
98
99      def makeLattice(self):
```

```
100         """Generate and save a lattice for the field by user input and lattice
    optimization"""
101         from math import floor
102
103         image = cv2.imread(self.image_path)
104
105         good_guess = False
106         while not good_guess:
107             fig, ax = plt.subplots(figsize=(24, 12))
108             ax.imshow(image, cmap='gray')
109             plt.get_current_fig_manager().window.showMaximized()
110
111             print('Please input points to define an initial guess for lattice.')
112             points = plt.ginput(3)
113             plt.close()
114
115             adjust = floor((self.Nb - 1) / 2)
116
117             offset = np.array(points[0])
118             vec_a = (np.array(points[1]) - offset) / (self.Na - 1)
119             vec_b = (np.array(points[2]) - offset + vec_a * (adjust - self.Na + 1)) / (
    self.Nb - 1)
120
121             self.lattice = Lattice(self.Na, self.Nb, vec_a, vec_b, offset)
122
123             self.plotLattice()
124
125             answer = input('Does the lattice look decent? (Y/N) ')
126             if answer == 'y' or answer == 'Y':
127                 good_guess = True
128             else:
129                 print('Try again.')
130
131         print('Optimizing lattice')
132         self.lattice = self.optimizeLattice(self.lattice)
133         print('Lattice optimized')
134         self.plotLattice()
135         print('Saving new lattice')
136         self.clearBlobsByPoint()
137         pickle.dump(self.lattice, open(self.lattice_path, 'wb'))
138
139     def optimizeLattice(self, lattice):
140         """Optimze the lattice to fit with the detected blobs"""
141
142         def getRSS(params, Na, Nb, blobs_by_point):
```

```
143             """Return the sum of squared distances between all blobs and their lattice
    point"""
144             vax, vay, vbx, vby, ox, oy = params
145             lattice = Lattice(Na, Nb, [vax, vay], [vbx, vby], [ox, oy])
146 
147             lattice_points = lattice.getLatticePoints()
148             sum = 0
149 
150             for i, point in enumerate(blobs_by_point):
151                 point_x, point_y = lattice_points[i]
152                 for blob in point:
153                     blob_y, blob_x, r = blob
154                     square_dist = (point_x - blob_x) ** 2 + (point_y - blob_y) ** 2
155 
156                     sum += square_dist
157 
158             return sum
159 
160         def fixParams(params):
161             """Help function for optimizeLattice
162 
163                 Format the parameters given by lattice.getParams to be used by scipy.
    optimize.minimize
164             """
165             vax = params[2][0]
166             vay = params[2][1]
167             vbx = params[3][0]
168             vby = params[3][1]
169             ox = params[4][0]
170             oy = params[4][1]
171 
172             return vax, vay, vbx, vby, ox, oy
173 
174         from scipy.optimize import minimize
175         params = np.array(fixParams( lattice.getParams() ))
176         res = minimize(getRSS, params, args=(self.Na, self.Nb, self.getBlobsByPoint()),
    method='Nelder-Mead')
177 
178         vax, vay, vbx, vby, ox, oy = res['x']
179         lattice = Lattice(self.Na, self.Nb, [vax, vay], [vbx, vby], [ox, oy])
180 
181         return lattice
182 
183     def readjustLattice(self):
```

```
184         """If field already has lattice defined, readjuts lattice to fit best with
    current detected blobs"""
185         found = True
186         if self.lattice == None:
187             try:
188                 self.lattice = pickle.load(open(self.lattice_path, 'rb'))
189                 if self.lattice == None:
190                     found = False
191             except FileNotFoundError:
192                 found = False
193 
194         if found:
195             self.lattice = self.optimizeLattice(self.lattice)
196             pickle.dump(self.lattice, open(self.lattice_path, 'wb'))
197             print('Lattice', self.name, 'readjusted')
198 
199             return 1
200 
201         else:
202             print('No lattice to adjust for field', self.name)
203 
204             return 0
205 
206     def getLattice(self):
207         """Return lattice object for field. Load if possible, make if necessary."""
208         if self.lattice != None:
209             return self.lattice
210         else:
211             try:
212                 self.lattice = pickle.load(open(self.lattice_path, 'rb'))
213                 if self.lattice == None:
214                     print('Loaded lattice, but object was empty.')
215                     self.makeLattice()
216             except FileNotFoundError:
217                 print('Lattice file not found!')
218                 self.makeLattice()
219 
220         return self.lattice
221 
222     def clearLattice(self):
223         """Delete all stored and loaded information about the lattice for this tile"""
224         self.lattice = None
225 
226         try:
227             os.remove(self.lattice_path)
```

```
228         except FileNotFoundError:
229             pass
230 
231         self.clearBlobsByPoint()
232 
233     def generateBlobsByPoint(self):
234         """Assigns all blobs to their nearest lattice point, and stores the list of blobs
     by point"""
235         blobs = self.getBlobs()
236         lattice = self.getLattice()
237 
238         lattice_points = lattice.getLatticePoints()
239         lattice_distance = lattice.getMinLatticeDist()
240         self.blobs_by_point = []
241 
242         for point in lattice_points:
243             blobs_for_this_point = []
244 
245             for blob in blobs:
246                 y, x, r = blob
247 
248                 if f.isInCircle(x, y, point[0], point[1], lattice_distance / 2):
249                     blobs_for_this_point.append(blob)
250 
251             self.blobs_by_point.append(blobs_for_this_point)
252 
253         pickle.dump(self.blobs_by_point, open(self.blobs_by_point_path, 'wb'))
254         print('Blobs assigned for field', self.name)
255 
256     def getBlobsByPoint(self):
257         """Return all detected blobs for field. Load if possible, detect if necessary."""
258         if len(self.blobs_by_point) > 0:
259             return self.blobs_by_point
260         else:
261             try:
262                 self.blobs_by_point = pickle.load(open(self.blobs_by_point_path, 'rb'))
263                 if len(self.blobs_by_point) < 1:
264                     print('Loaded blobs by point, but array was empty. Assigning blobs.')
265                     self.generateBlobsByPoint()
266             except FileNotFoundError:
267                 print('Blobs per point file not found! Assigning blobs.')
268                 self.generateBlobsByPoint()
269 
270         if len(self.blobs_by_point) > 0:
271             return self.blobs_by_point
```

```
272         else:
273             raise RuntimeError('Not able to obtain blobs by point!')
274 
275     def clearBlobsByPoint(self):
276         """Delete all stored and loaded information about blobs by point for this tile"""
277         self.blobs_by_point = []
278 
279         try:
280             os.remove(self.blobs_by_point_path)
281         except FileNotFoundError:
282             pass
283 
284     def getBlobCount(self):
285         """Return the number of detected blobs"""
286         blobs = self.getBlobs()
287         return blobs.shape[0]
288 
289     def getDiameters(self):
290         """Return list of diameters of detected blobs in nm"""
291         blobs = self.getBlobs()
292         diameters = blobs[:, 2] * 2 * self.scale
293 
294         return diameters
295 
296     def getMeanDiameter(self):
297         """Return mean diameter of detected blobs in nm"""
298         return np.mean(self.getDiameters())
299 
300     def getMedianDiameter(self):
301         """Return median diameter of detected blobs in nm"""
302         return np.median(self.getDiameters())
303 
304     def getBlobCountByPoint(self):
305         """Return list of count of assigned blobs for each lattice point"""
306         blobs_by_point = self.getBlobsByPoint()
307         return [len(point) for point in blobs_by_point]
308 
309     def getDisplacements(self):
310         """Returns an array of x and y displacement of blobs from their lattice point"""
311         lattice_points = self.getLattice().getLatticePoints()
312         blobs_by_point = self.getBlobsByPoint()
313         displacements = []
314 
315         for i, point in enumerate(blobs_by_point):
316             point_x, point_y = lattice_points[i]
```

```
317             for blob in point:
318                 blob_y, blob_x, r = blob
319 
320                 displacements.append([ blob_x - point_x, blob_y - point_y ])
321 
322         return displacements
323 
324     def getDisplacementMagnitudes(self):
325         """Returns an array of displacement magnitudes of blobs from their lattice point
    """
326         displacements = self.getDisplacements()
327         return [np.linalg.norm(displacement) * self.scale for displacement in
    displacements]
328 
329     def getDisplacementAngles(self):
330         """Returns an array of displacement angles of blobs from their lattice point"""
331         displacements = self.getDisplacements()
332         angles = [np.angle( d[0] - 1j*d[1] ) for d in displacements]
333 
334         return angles
335 
336     def getYields(self):
337         """Returns an array of the yield numbers for n blobs per point for all applicable
     values of n"""
338         blob_count_by_point = self.getBlobCountByPoint()
339         yields = []
340 
341         for n in range(0, max(blob_count_by_point) + 1):
342             yields.append( blob_count_by_point.count(n) )
343 
344         return yields
345 
346     def getYield(self, n, percentage=False):
347         """Returns the number or yield percentage for n blobs per point for a given value
     of n"""
348         yields = self.getYields()
349 
350         try:
351             number = yields[n]
352         except IndexError:
353             number = 0
354 
355         if percentage:
356             return number * 100 / self.number_of_points
357         else:
```

```
358             return number
359
360     def plotBlobs(self, show_image=True, save=False, prefix='', postfix=''):
361         """Plot detected blobs
362
363             Keyword arguments:
364             path -- if set, the plot is saved to the path, otherwise the plot is
    displayed
365             show_image -- if false, blobs are plotted without the background image
366         """
367
368         blobs = self.getBlobs()
369
370         fig, ax = plt.subplots(figsize=(24, 24))
371         ax.set_aspect('equal', adjustable='box-forced')
372         plt.axis((0, 1024, 883, 0))
373         if show_image:
374             image = cv2.imread(self.image_path)
375             plt.imshow(image, cmap='gray', interpolation='nearest')
376             plt.axis((0, image.shape[1], image.shape[0], 0))
377         f.plotCircles(ax, blobs, fig, dict(color='red', linewidth=2, fill=False))
378         ax.set_yticklabels([])
379         ax.set_xticklabels([])
380         plt.tight_layout()
381         fig.subplots_adjust(0, 0, 1, 1)
382
383         fig_name = 'blobs'
384         if save:
385             full_path = self.figure_path + fig_name + '_' + prefix + self.name + postfix
    + '.png'
386             plt.savefig(full_path)
387             print('Saved', fig_name, 'plot for field', self.name)
388         else:
389             plt.show()
390         plt.close()
391
392     def plotLattice(self, lattice_color='red', figsize=(10, 10), save=False, prefix='',
    postfix=''):
393         """Plot lattice points"""
394         image = cv2.imread(self.image_path)
395         lattice_points = self.getLattice().getLatticePoints()
396
397         plt.figure(figsize=figsize)
398         plt.imshow(image, cmap='gray')
399         ax = plt.gca()
```

```
400         ax.set_axis_off()
401
402         x = [x for [x, y] in lattice_points]
403         y = [y for [x, y] in lattice_points]
404
405         plt.scatter(x, y, marker='.', color=lattice_color, zorder=10)
406
407         plt.tight_layout()
408
409         fig_name = 'lattice'
410         if save:
411             full_path = self.figure_path + fig_name + '_' + prefix + self.name + postfix
    + '.png'
412             plt.savefig(full_path)
413             print('Saved', fig_name, 'plot for field', self.name)
414         else:
415             plt.show()
416         plt.close()
417
418     def plotLatticeAndBlobs(self, blob_color='', lattice_color='cyan', figsize=(10, 10),
    save=False, prefix='', postfix=''):
419         """Plot lattice points, and detected blobs colored by lattice point """
420         from matplotlib.collections import PatchCollection
421
422         lattice_points = self.getLattice().getLatticePoints()
423         blobs_by_point = self.getBlobsByPoint()
424         image = cv2.imread(self.image_path)
425
426         plt.figure(figsize=figsize)
427         plt.imshow(image, cmap='gray')
428         ax = plt.gca()
429         ax.set_axis_off()
430
431         if blob_color == '':
432             colors = f.randomColors(len(lattice_points))
433         else:
434             colors = [blob_color] * len(lattice_points)
435
436         patches = []
437
438         for i, point in enumerate(blobs_by_point):
439             color = colors[i]
440
441             for blob in point: # Plot blobs
442                 y, x, r = blob
```

```
443                 c = plt.Circle((x, y), r, color=color, linewidth=2, fill=False)
444                 patches.append(c)
445 
446         x = [x for [x, y] in lattice_points]
447         y = [y for [x, y] in lattice_points]
448 
449         plt.scatter(x, y, marker='.', color=lattice_color, zorder=10)
450 
451         p = PatchCollection(patches, match_original=True)
452         ax.add_collection(p)
453 
454         plt.tight_layout()
455 
456         fig_name = 'blobs+lattice'
457         if save:
458             full_path = self.figure_path + fig_name + '_' + prefix + self.name + postfix
    + '.png'
459             plt.savefig(full_path)
460             print('Saved', fig_name, 'plot for field', self.name)
461         else:
462             plt.show()
463         plt.close()
464 
465     def plotHistogram(self, property, bins=40, fontsize=20, save=False, prefix='',
    postfix=''):
466         """Plot a histogram of a given property of the detected blobs
467 
468         :param property: the property to be plotted. Can be either 'diameter', 'distance'
     or 'angle'
469         :param bins: the number of bins used for the histogram
470         :param fontsize: size of the font used in the plot
471         """
472         if property == 'diameter':
473             label = 'diameter [nm]'
474             data = self.getDiameters()
475         elif property == 'distance':
476             label = 'displacement from lattice point [nm]'
477             data = self.getDisplacementMagnitudes()
478         elif property == 'angle':
479             label = 'angle'
480             data = self.getDisplacementAngles()
481         else:
482             raise ValueError("'" + property + "' is not a valid property")
483 
```

```
484         fig, ax = plt.subplots(1, 1, figsize=(6, 3), subplot_kw={'adjustable': 'box-
    forced'})
485 
486         ax.set_ylim((0, 70))
487         ax.hist(data, bins=bins, range = [0, 300], edgecolor='none', color='#033A87')
488         plt.xlabel(label, fontsize=fontsize)
489         plt.ylabel('count', fontsize=fontsize)
490 
491         for tick in ax.xaxis.get_major_ticks():
492             tick.label.set_fontsize(fontsize)
493 
494         for tick in ax.yaxis.get_major_ticks():
495             tick.label.set_fontsize(fontsize)
496 
497         plt.tight_layout()
498 
499         fig_name = property + ' histogram'
500         if save:
501             full_path = self.figure_path + fig_name + '_' + prefix + self.name + postfix
    + '.png'
502             plt.savefig(full_path)
503             print('Saved', fig_name, 'plot for field', self.name)
504         else:
505             plt.show()
506         plt.close()
```

## newFieldArray.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3 
4  from math import ceil, log10
5 
6  import functions as f
7  import detect
8  from newField import Field
9 
10 class FieldArray:
11     field_ext = '.fld'
12 
13     def __init__(self, nfa, nfb, Na, Nb, path, scale, ext='.tif'):
14         self.nfa = nfa
15         self.nfb = nfb
16         self.Na = Na
```

```
17          self.Nb = Nb
18          self.path = path
19          self.scale = scale
20
21          self.num_fields = nfa * nfb
22          self.fields = [Field(Na, Nb, path, str(num+1).zfill(3), scale, ext) for num in
    range(0, self.num_fields)]
23
24      def detectBlobs(self, methods=(detect.droplets,)):
25          for field in self.fields:
26              field.detectBlobs(methods)
27
28      def ensureBlobs(self, methods=(detect.droplets,)):
29          for field in self.fields:
30              field.getBlobs(methods)
31
32      def clearBlobs(self):
33          for field in self.fields:
34              field.clearBlobs()
35
36      def makeLattices(self):
37          for field in self.fields:
38              field.makeLattice()
39
40      def ensureLattices(self):
41          for field in self.fields:
42              field.getLattice()
43
44      def readjustLattices(self):
45          for field in self.fields:
46              success = field.readjustLattice()
47              if not success:
48                  raise RuntimeError('No lattice to adjust for field ' + field.name)
49
50      def clearLattices(self):
51          for field in self.fields:
52              field.clearLattice()
53
54      def generateBlobsByPoint(self):
55          for field in self.fields:
56              field.generateBlobsByPoint()
57
58      def ensureBlobsByPoint(self):
59          for field in self.fields:
60              field.getBlobsByPoint()
```

```
61
62      def clearBlobsByPoint(self):
63          for field in self.fields:
64              field.clearBlobsByPoint()
65
66      def getMeanDiameters(self):
67          return [field.getMeanDiameter() for field in self.fields]
68
69      def getMedianDiameters(self):
70          return [field.getMedianDiameter() for field in self.fields]
71
72      def getYields(self, n):
73          return [field.getYield(n, percentage=True) for field in self.fields]
74
75      def listFieldsByYield(self, n):
76          yields = self.getYields(n)
77          field_list = list(enumerate(yields, start=1))
78          field_list = sorted(field_list, key=lambda x: (x[1]), reverse=True)
79          for field in field_list:
80              print(str(field[0]).rjust(ceil(log10(len(field_list)))), ': ', round(field
    [1], 1), sep='')
81
82      def plotBlobs(self):
83          for field in self.fields:
84              field.plotBlobs(save=True)
85
86      def plotLattices(self, path):
87          for field in self.fields:
88              field.plotLattice(save=True)
89
90      def plotLatticesWithBlobs(self, path):
91          for field in self.fields:
92              field.plotLatticeAndBlobs(save=True)
93
94      def plotAvgBlobs(self, kwargs):
95          average_blobs = [field.getBlobCount() / field.number_of_points for field in self.
    fields]
96          plt = f.surfacePlot(average_blobs, **kwargs)
97          plt.show()
98
99      def plotMeanDiameters(self):
100         mean_diameters = self.getMeanDiameters()
101         plt = f.surfacePlot(mean_diameters)
102         plt.show()
103
```

```
104     def plotMedianDiameters(self, kwargs):
105         median_diameters = self.getMedianDiameters()
106         plt = f.surfacePlot(median_diameters, **kwargs)
107         plt.show()
108 
109     def plotMeanDisplacements(self, kwargs):
110         mean_displacements = [np.mean(field.getDisplacementMagnitudes()) for field in
    self.fields]
111         for i, d in enumerate(mean_displacements):
112             print(i+1, d)
113         plt = f.surfacePlot(mean_displacements, **kwargs)
114         plt.show()
115 
116     def plotMedianDisplacements(self, kwargs):
117         median_displacements = [np.median(field.getDisplacementMagnitudes()) for field in
     self.fields]
118         plt = f.surfacePlot(median_displacements, **kwargs)
119         plt.show()
120 
121     def plotDisplacementStd(self):
122         std_displacements = [np.std(field.getDisplacementMagnitudes()) for field in self.
    fields]
123         plt = f.surfacePlot(std_displacements)
124         plt.show()
125 
126     def plotDisplacementMdev(self):
127         mdevs = []
128         for field in self.fields:
129             data = field.getDisplacementMagnitudes()
130             d = np.abs(data - np.median(data))
131             mdev = np.median(d)
132             mdevs.append(mdev)
133 
134         plt = f.surfacePlot(mdevs)
135         plt.show()
136 
137     def plotYield(self, n):
138         yields = self.getYields(n)
139         title = str(n) + ' blobs'
140         plt = f.surfacePlot(yields, title=title, percentages=True)
141         plt.show()
142 
143     def plotAllYields(self):
144         pass  # TODO: Make the function
145 
```

```
146     def plotDiameterHistograms(self):
147         from math import ceil
148         diametersPerField = [field.getDiameters() for field in self.fields]
149         fig, axes = plt.subplots(8, 8, figsize=(21.5, 10), subplot_kw={'adjustable': 'box
    -forced'})
150         for n, diameters in enumerate(diametersPerField):
151             a = 215
152             row = 7 - (n) % 8
153             col = ceil((n + 1) / 8) - 1
154 
155             x_max = 300
156             ax = axes[row, col]
157             ax.set_title(n + 1)
158             ax.set_xlim((0, x_max))
159             ax.set_ylim((0, 70))
160             ax.get_xaxis().set_ticks([])
161             ax.get_yaxis().set_ticks([])
162             x = np.arange(len(diameters))
163             ax.hist(diameters, bins=40, histtype='stepfilled', color='#033A87', edgecolor
    ='none', range=[0, x_max])
164             line_kwargs = {'linewidth': 2, 'color': 'red'}
165             ax.plot([a, a], [0, 100], **line_kwargs)
166 
167         plt.tight_layout()
168         plt.show()
169 
170     def plotAmountOverLimit(self, limit):
171         values = []
172         for field in self.fields:
173             diameters = field.getDiameters()
174             amount_over_limit = len([d for d in diameters if d > limit])
175             ratio = amount_over_limit / len(diameters) * 100
176             values.append(ratio)
177 
178         f.surfacePlot(values, percentages=True)
179         plt.show()
180 
181     @staticmethod
182     def reject_outliers(data, m):
183         d = np.abs(data - np.median(data))
184         mdev = np.median(d)
185         s = d / mdev if mdev else 0.
186         return data[s < m]
187 
188     def plotDisplacementHistograms(self):
```

```
189         from math import ceil
190         displacements_per_field = [field.getDisplacementMagnitudes() for field in self.
    fields]
191 
192         fig, axes = plt.subplots(8, 8, figsize=(21.5, 10), subplot_kw={'adjustable': 'box
    -forced'})
193         for n, displacements in enumerate(displacements_per_field):
194 
195             displacements_per_field[n] = displacements
196 
197             row = 7 - (n) % 8
198             col = ceil((n + 1) / 8) - 1
199 
200             ax = axes[row, col]
201             ax.set_title(n + 1)
202             ax.set_xlim((0, 30))
203             ax.set_ylim((0, 70))
204             ax.get_xaxis().set_ticks([])
205             ax.get_yaxis().set_ticks([])
206             ax.hist(displacements, bins=50, histtype='stepfilled', edgecolor='none',
    range=(0, 30))
207 
208         plt.tight_layout()
209         plt.show()
210         plt.close()
211 
212     def plotDisplacementAngleHistograms(self):
213         from math import ceil
214         from math import pi
215         displacement_angles_per_field = [field.getDisplacementAngles() for field in self.
    fields]
216         fig, axes = plt.subplots(8, 8, figsize=(21.5, 10), subplot_kw={'adjustable': 'box
    -forced'})
217         for n, angles in enumerate(displacement_angles_per_field):
218 
219             row = 7 - (n) % 8
220             col = ceil((n + 1) / 8) - 1
221 
222             ax = axes[row, col]
223             ax.set_title(n + 1)
224             ax.set_xlim((-pi, pi))
225             ax.set_ylim((0, 25))
226             ax.get_xaxis().set_ticks([])
227             ax.get_yaxis().set_ticks([])
228             ax.hist(angles, bins=50, histtype='stepfilled', edgecolor='none')
```

```
229
230         plt.tight_layout()
231         plt.show()
232
233     def plotDisplacementScatterPlots(self):
234         from math import ceil, pi
235         fig, axes = plt.subplots(8, 8, figsize=(12, 12), subplot_kw=dict(projection='
    polar'))
236         for n, field in enumerate(self.fields):
237
238             row = 7 - (n) % 8
239             col = ceil((n + 1) / 8) - 1
240             ax = axes[row, col]
241
242             r = field.getDisplacementMagnitudes()
243             angles = field.getDisplacementAngles()
244
245             ax.scatter(angles, r, color='mediumblue', s=3, alpha=0.5, edgecolor='none')
246             ax.grid(color='#EEEEEE', linestyle='-', linewidth=1)
247             ax.set_axisbelow(True)
248
249             ax.set_ylim((0, 500))
250             ax.get_xaxis().set_ticklabels([])
251             ax.get_xaxis().set_ticks([0, pi/2, pi, -pi/2])
252             ax.get_yaxis().set_ticklabels([])
253
254         plt.tight_layout()
255         plt.show()
256
257     def plotSingleScatterPlot(self, n):
258         from math import pi
259
260         ax1 = plt.subplot(121, projection='polar')
261
262         field = self.fields[n-1]
263
264         r = field.getDisplacementMagnitudes()
265         angles = field.getDisplacementAngles()
266
267         ax1.scatter(angles, r, color='mediumblue', s=10, alpha=0.5, edgecolor='none')
268         ax1.grid(color='#EEEEEE', linestyle='-', linewidth=1)
269         ax1.set_axisbelow(True)
270
271         ax1.set_ylim((0, 500))
272         ax1.get_xaxis().set_ticklabels([])
```

```
273         ax1.get_xaxis().set_ticks([0, pi / 2, pi, -pi / 2])
274         ax1.text(2.5, 34*self.scale, str(n).rjust(2))
275 
276         ax2 = plt.subplot(122)
277         ax2.hist(r, bins=70, range=[0, 500], histtype='stepfilled', color='limegreen',
    edgecolor='none')
278 
279         plt.show()
280 
281     def plotSingleDisplacementHistogram(self, n):
282         displacements = self.fields[n-1].getDisplacementMagnitudes()
283         plt.hist(displacements, bins=26, range=(0, 26), histtype='stepfilled', edgecolor=
    'none', color='#033A87')
284         plt.show()
285 
286     def plotOverallDiameterHistogram(self):
287         diameters = [field.getDiameters() for field in self.fields]
288         data = [item for sublist in diameters for item in sublist]
289         plt.hist(data, bins=150, histtype='stepfilled', edgecolor='none', color='#033A87'
    )
290         plt.show()
291 
292     def plotFancyDiameterHistogram(self):
293         bbp = [field.getBlobsByPoint() for field in self.fields]
294         bbp = [item for sublist in bbp for item in sublist]  # Flatten list
295 
296         points_single = [point for point in bbp if len(point) == 1]
297         diameters_single = [blob[2] * 2 * self.scale for point in points_single for blob
    in point]
298 
299         max_diameter = max(diameters_single)
300         min_diameter = min(diameters_single)
301         bins = 50
302 
303         max_num = 3
304         diameters_sorted = []
305         for n in range(1, max_num+1):
306             points = [point for point in bbp if len(point) == n]
307             diameters = [blob[2] * 2 * self.scale for point in points for blob in point]
308 
309             diameters_sorted.append(diameters)
310             max_diameter = max(max(diameters), max_diameter)
311             min_diameter = min(min(diameters), min_diameter)
312 
313         points = [point for point in bbp if len(point) > max_num]
```

```
314         diameters = [blob[2] * 2 * self.scale for point in points for blob in point]
315 
316         diameters_sorted.append(diameters)
317 
318         colors2 = ['dodgerblue', 'red', 'orange', 'limegreen', 'purple', 'magenta']
319         for n, data in enumerate(diameters_sorted):
320             if n == 0:
321                 label = '1 nanowire'
322             else:
323                 label = str(n+1) + ' nanowires'
324             plt.hist(data, bins=bins, zorder=0.5+n, label=label, histtype='step',
    edgecolor=colors2[n], alpha=1, range=[min_diameter, max_diameter], linewidth=3)
325 
326         plt.xlabel('diameter [nm]')
327         plt.ylabel('count')
328 
329         plt.legend()
330         plt.show()
331 
332     def plotOverallDisplacementHistogram(self):
333         displacements_per_field = [field.getDisplacementMagnitudes() for field in self.
    fields]
334         data = [item for sublist in displacements_per_field for item in sublist]
335         plt.hist(data, bins=100, histtype='stepfilled', edgecolor='none', color='#033A87'
    )
336         plt.show()
337 
338     def plotOverallDisplacementAngleHistogram(self):
339         displacements_per_field = [field.getDisplacementAngles() for field in self.fields
    ]
340         data = [item for sublist in displacements_per_field for item in sublist]
341         plt.hist(data, bins=20, histtype='stepfilled', edgecolor='none', color='#033A87')
342         plt.show()
```

# tileset.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy import misc
4 import random
5 import pickle
6 import os
7 
8 from math import pi
```

```
9
10 import functions as f
11 from arbitraryLattice import Lattice, makeLatticeByAngles, loadLattice
12
13 from timeCheckpoint import timeCheckpoint
14 from time import clock
15
16 class Tileset:
17     default_padding = 100
18
19     def __init__(self, path, cols, rows, tilew, tileh, scale, detection_method, ext='.tif
       '):
20         self.path = path
21         self.ext = ext
22         self.rows = rows
23         self.cols = cols
24         self.tileh = tileh
25         self.tilew = tilew
26         self.scale = scale
27         self.detection_method = detection_method
28         self.blobs = np.array([])
29         self.assigned_blobs = []
30         self.lattice = None
31
32
33     def getTile(self, col, row):
34         """ Load from file and return a tile specified by row and column.
35         If there is no tile at the specified position, returns an array of zeroes with
       the same size as a tile.
36
37         :param col: the column of the tile to be returned
38         :param row: the row of the tile to be returned
39         :return: numpy array of the tile image
40         """
41         file_path = self.path + '/c_' + str(col) + '/tile_' + str(row) + self.ext
42         try:
43             tile = misc.imread(file_path)
44         except FileNotFoundError:
45             tile = np.zeros((self.tileh, self.tilew), dtype=np.uint8)
46
47         return tile
48
49     def getTileRegion(self, col_min, col_max, row_min, row_max):
50         """Return a region of the image by concatenating a set of tiles
51
```

```
52          :param col_min:
53          :param col_max:
54          :param row_min:
55          :param row_max:
56          :return: numpy array of an image spanning the specified region
57          """
58          r_width = self.tilew * (col_max - col_min + 1)
59          r_height = self.tileh * (row_max - row_min + 1)
60          region = np.zeros((r_height, r_width), dtype=np.uint8)
61
62          for col in range(col_min, col_max+1):
63              for row in range(row_min, row_max+1):
64
65                  tile = self.getTile(col, row)
66
67                  h_min = self.tileh * (row - row_min)
68                  h_max = self.tileh * (row - row_min + 1)
69                  w_min = self.tilew * (col - col_min)
70                  w_max = self.tilew * (col - col_min + 1)
71
72                  region[h_min:h_max, w_min:w_max] = tile
73
74          return region
75
76      def getPaddedTile(self, col, row, padding=default_padding):
77          """Return a tile with padding from adjacent tiles
78
79          :param col: the column of the tile to be returned
80          :param row: the row of the tile to be returned
81          :param padding: size of the padding in pixels
82          :return: numpy array of the padded tile
83          """
84          if padding > min(self.tilew, self.tileh):
85              raise RuntimeError('Padding of ' + str(padding) + ' is too large!')
86
87          region = self.getTileRegion(col-1, col+1, row-1, row+1)
88
89          h_crop = self.tileh - padding
90          v_crop = self.tilew - padding
91
92          padded = region[h_crop:-h_crop, v_crop:-v_crop]
93
94          return padded
95
96      def prepTiles(self, output_path, kernel_size, fill=True):
```

```
97          """Do preprocessing on all tiles in tileset, and save the preprocessed tiles to
    output_path
98          Preprocessing consists of filling in using reconstruction, and median filtering
99
100         :param output_path: the path where the processed tiles will be saved
101         :param kernel_size: width of the kernel used for median filtering
102         :return: Tileset object containing the preprocessed tiles
103         """
104         from scipy.signal import medfilt2d
105
106         for col in range(0, self.cols):
107             col_path = output_path + '/c_' + str(col)
108             if not os.path.exists(col_path):
109                 os.makedirs(col_path)
110
111             for row in range(0, self.rows):
112                 tile = self.getPaddedTile(col, row)
113
114                 if fill:
115                     tile = f.fillWires(tile)
116
117                 tile = medfilt2d(tile, kernel_size)
118
119                 if fill:
120                     tile = f.fillWires(tile)
121
122                 p = self.default_padding
123                 cropped_tile = tile[p:-p, p:-p]
124
125                 filename = col_path + '/tile_' + str(row) + self.ext
126                 misc.imsave(filename, cropped_tile)
127                 print('Saved tile ' + str(col) + ', ' + str(row))
128
129         return Tileset(output_path, self.cols, self.rows, self.tilew, self.tileh, self.
    scale, self.detection_method,
130                        self.ext)
131
132     def detectBlobs(self, col, row, globalize=False):
133         """Run detection and return array of detected blobs for the specified tile
134
135         :param col: column number of the tile on which to perform detection
136         :param row: row number of the tile on which to perform detection
137         :param globalize: if true, blobs will be returned with global coordinates
138         :return: numpy array of the detected blobs
139         """
```

```
140         padded_tile = self.getPaddedTile(col, row)
141 
142         blobs = self.detection_method(padded_tile)  # detect blobs
143 
144         padding = self.default_padding
145         outside = []
146         for i, blob in enumerate(blobs):  # figure out which blobs lie outside the non-
    padded tile
147             if min(blob[0:2]) < padding or blob[0] >= (self.tileh+padding) or blob[1] >=
    (self.tilew+padding):
148                 outside.append(i)
149 
150         blobs = np.delete(blobs, outside, 0)  # delete blobs that lie outside the non-
    padded tile, to avoid duplicates
151 
152         blobs[:, 0:2] -= padding  # readjust the coordinates of the blobs to be relative
    to the non-padded tile
153         if globalize:  # convert the coordinates of the blob from coords within the tile
    to coords for the whole tileset
154             blobs[:, 1] += col * self.tilew
155             blobs[:, 0] += row * self.tileh
156 
157         print('Blobs found:', blobs.shape[0])
158 
159         return blobs
160 
161     def detectAllBlobs(self):
162         """Run detection on all tiles, and save the result."""
163         blobs = False
164         for col in range(0, self.cols):
165             for row in range(0, self.rows):
166                 found = self.detectBlobs(col, row, globalize=True)
167                 print('Detected blobs for tile ' + str(col) + ', ' + str(row))
168                 if blobs is False:
169                     blobs = found
170                 else:
171                     blobs = np.append(blobs, found, axis=0)
172 
173         self.blobs = blobs
174         self.assigned_blobs = []
175 
176         self.saveBlobs()
177 
178     def saveBlobs(self):
179         """Store all currently detected blobs to a file located at self.path"""
```

```
180         full_path = self.path + '/blobs.p'
181         pickle.dump(self.blobs, open(full_path, 'wb'))
182 
183     def deleteBlobs(self):
184         """Delete the file and clear the variable containing detected blobs."""
185         full_path = self.path + '/blobs.p'
186         try:
187             os.remove(full_path)
188         except FileNotFoundError:
189             print('No file to delete')
190         self.blobs = np.array([])
191 
192     def getBlobs(self):
193         """Return all detected blobs for tileset. Load if possible, detect if necessary.
    """
194         if self.blobs.shape[0] > 0:
195             return self.blobs
196         else:
197             try:
198                 full_path = self.path + '/blobs.p'
199                 self.blobs = pickle.load(open(full_path, 'rb'))
200                 if self.blobs.shape[0] < 1:
201                     print('Loaded blobs, but array was empty. Detecting blobs.')
202                     self.detectAllBlobs()
203             except FileNotFoundError:
204                 print('Blobs file not found. Detecting blobs.')
205                 self.detectAllBlobs()
206 
207         if self.blobs.shape[0] > 0:
208             return self.blobs
209         else:
210             raise Exception('Not able to obtain blobs!')
211 
212     def getSubsetOfBlobs(self, x_min, x_max, y_min, y_max):
213         """Return all detected blobs for specified coordinate region. Load if possible,
    detect if necessary."""
214         # Get all blobs
215         blobs = self.getBlobs()
216 
217         # Remove the ones outside the specified area
218         outside = []
219         for i, blob in enumerate(blobs):
220             if blob[0] < y_min or blob[0] > y_max or blob[1] < x_min or blob[1] > x_max:
221                 outside.append(i)
222 
```

```
223         blobs = np.delete(blobs, outside, 0)
224 
225         return blobs
226 
227     @staticmethod
228     def findFirstBlob(blobs):
229         """Helper function for makeLattice: Return the most top left blob in the given
    set of blobs."""
230         vals = []
231         for blob in blobs:
232             vals.append(blob[0] + blob[1])  # x + y coordinate of blob
233 
234         i = np.argmin(vals)  # minimum x + y is top left
235 
236         return blobs[i]
237 
238     @staticmethod
239     def getAnglesFromInput(tile, blobs, offset):
240         """Helper function for makeLattice:
241         Display an image with detected blobs plotted, and get user input to define angles
     for lattice vectors.
242 
243         :param tile: the tile to be displayed as background image
244         :param blobs: the detected blobs for the tile
245         :param offset: the point representing the origin of the lattice, angles are
    defined relative to this point
246         :return: the two angles defined by the user input, in radians
247         """
248         fig, ax = plt.subplots(figsize=(24, 12))
249         ax.set_aspect('equal', adjustable='box-forced')
250         plt.axis((0, tile.shape[1], tile.shape[0], 0))
251         plt.title("Please click two points")
252         plt.tight_layout()
253 
254         plt.imshow(tile, cmap='gray', interpolation='nearest')
255         f.plotCircles(ax, blobs, fig, dict(color='#114400', linewidth=4, fill=False))
256         plt.plot(offset[0], offset[1], '.', color='red', markersize=10)
257 
258         # Get input
259         points = plt.ginput(2)
260         plt.close()
261 
262         # Calculate angles from input
263         displacements = [np.array(point) - offset for point in points]
264         angles = [np.angle(dis[0] + 1j*dis[1]) for dis in displacements]
```

```
265
266         return angles
267
268     @staticmethod
269     def getTypicalDistance(blobs):
270         """Helper function for makeLattice: Get an initial guess for the magnitude of
    lattice vectors by finding the
271         typical distance between blobs.
272
273         :param blobs: the blobs between which to find the typical distance
274         :return: float representing the typical distance between blobs in pixels
275         """
276
277         def reject_outliers(data, m):
278             """Removes outliers from a dataset using deviations from the median instead
    of the mean, since this is
279             more robust, and less affected by outliers
280
281             :param data: the data to be filtered, must be a numpy array of numbers
282             :param m: the amount of median deviations from the median beyond which to
    discard data
283             :return: all the data points laying within m median deviations from the
    median
284             """
285             d = np.abs(data - np.median(data))  # array of each number's deviation from
    the median value
286             mdev = np.median(d)  # the median deviation from the median value
287             s = d / mdev if mdev else 0.  # array of each number's deviation from the
    median value, given in
288                                           # multiples of the median deviation from the
    median value
289             return data[s < m]
290
291         from scipy.spatial import KDTree
292         points = blobs[:, 0:2]  # get just x and y coordinates of blobs, not the radii
293
294         tree = KDTree(points)  # put the points into a data structure allowing for quick
    neighbor distance lookup
295         results = [tree.query(points, 7)]  # return the six nearest points to each point
296
297         distances = [result[1:7] for result in results[0][0]]  # get the actual distances
298         distances = np.array(distances).flatten()  # make a flattened array of all the
    inter-point distances
299         distances = reject_outliers(distances, 5)  # remove outliers beyond 5 median
    deviations from the median
```

```
300
301         return np.mean(distances)  # return the median value of the filtered distances
302
303     @staticmethod
304     def optimizeLattice(lattice, assigned_blobs, debug=False):
305         """Use given lattice as an initial guess, and numerically optimize lattice to
    minimize
306         the sum of square distances between each blob and it's nearest lattice point.
307
308         :param lattice: the initial guess for a lattice
309         :param assigned_blobs: the blobs to which to fit the lattice, allready assigned
    to their nearest lattice point
310         :param debug: if True, debug info will be printed
311         :return: the optimized lattice
312         """
313         def getRSS(params, assigned_blobs):
314             """Return the sum of the squared distance between each of the given blobs and
     it's nearest lattice point."""
315             mag_a, ang_a, mag_b, ang_b, ox, oy = params
316             lattice = makeLatticeByAngles(mag_a, ang_a, mag_b, ang_b, [ox, oy])
317
318             sum = 0
319
320             for blob_p in assigned_blobs:
321                 if blob_p['point'] != []:
322                     blob_y, blob_x, r = blob_p['blob']
323                     [point_x, point_y] = lattice.getCoordinates(*blob_p['point'])
324                     square_dist = (point_x - blob_x) ** 2 + (point_y - blob_y) ** 2
325
326                     sum += square_dist
327
328             return sum
329
330         from scipy.optimize import minimize
331         params = np.array([lattice.len_a, lattice.ang_a, lattice.len_b, lattice.ang_b,
332                            lattice.offset[0], lattice.offset[1]])
333
334         print('Blobs: ' + str(len(assigned_blobs)))
335
336         # Minimize the sum of square distances between blobs and their nearest lattice
    point, by adjusting the given
337         # parameters.
338         res = minimize(getRSS, params, args=(assigned_blobs), method='Nelder-Mead')
339
```

```
340         mag_a, ang_a, mag_b, ang_b, ox, oy = res['x']  # the parameters found to give the best lattice fit
341         lattice = makeLatticeByAngles(mag_a, ang_a, mag_b, ang_b, [ox, oy])
342 
343         if debug:
344             print(mag_a, ang_a, mag_b, ang_b, ox, oy)
345 
346         return lattice
347 
348     def makeLattice(self, max_blobs=500, final_blobs=4000, step=3, debug=False):
349         """Run the whole process necessary to get a lattice defined for the tileset, and save it to file.
350 
351         :param max_blobs: the maximum number of blobs to use for each round of optimization
352         :param final_blobs: the maximum number of blobs to use for the final round of optimization
353         :param step: how many new rows/columns to add for each new round of optimization
354         :param debug: if True, some debug info will be printed, and extra steps will be shown
355         """
356         # Setup
357         tw = self.tilew
358         th = self.tileh
359         bounds = (0, tw, 0, th)
360 
361         # The process starts with an initial guess based on the top left tile.
362         tile = self.getTile(0, 0)
363         blobs = self.getSubsetOfBlobs(*bounds)  # get the blobs for the top left tile
364         # The top left blob is used as the offset for the initial lattice guess.
365         first = self.findFirstBlob(blobs)
366         offset = [first[1], first[0]]
367 
368         # Angles of the lattice vectors for the initial lattice guess are given by manual input.
369         angles = self.getAnglesFromInput(tile, blobs, offset)
370         if len(angles) < 2:
371             raise RuntimeError("Insufficient input received.")
372         # The magnitude of the lattice vectors for the initial lattice guess is given by the typical neighbor distance.
373         magnitude = self.getTypicalDistance(self.getSubsetOfBlobs(0, 4*tw, 0, 4*th))
374 
375         lattice = makeLatticeByAngles(magnitude, angles[0], magnitude, angles[1], offset)
376         assigned_blobs = self.assignBlobs(blobs, lattice)
377 
```

```
378         # Show the initial guess lattice to the user, to ensure input was not completely
    wrong
379         self.lattice = lattice  # needs to be set for displayTileRegion
380         self.displayTileRegion(0, 0, 0, 0, blob_color='green', lattice_color='red')
381 
382         lattice = self.optimizeLattice(lattice, assigned_blobs)
383         print('Lattice optimized for first tile.')
384 
385         if debug:
386             self.lattice = lattice  # needs to be set for displayTileRegion
387             self.displayTileRegion(0, 0, 0, 0, blob_color='green', lattice_color='red')
388 
389         def optimizeWithBounds(self, lattice, bounds, max_blobs):
390             """Optimize the given lattice to fit best with blobs selected from a region
    of the tileset
391 
392             :param self: the tileset object
393             :param lattice:  the lattice to optimize
394             :param bounds: bounds of the region from which to select blobs
395             :param max_blobs: the max number of blobs to optimize against. If the total
    number of blobs in the region
396                               specified by bounds is larger than max_blobs, a random
    selection of mox_blobs blobs from
397                               the region is used
398             :return: optimized lattice
399             """
400             blobs = self.getSubsetOfBlobs(*bounds)
401             # If there are more than max_blobs blobs within bounds, get a random
    selection of max_blobs blobs
402             if blobs.shape[0] > max_blobs:
403                 blobs_list = list(blobs)
404                 blobs_list = [blobs_list[i] for i in random.sample(range(len(blobs_list))
    , max_blobs)]
405                 blobs = np.array(blobs_list)
406 
407             assigned_blobs = self.assignBlobs(blobs, lattice)
408             optimized_lattice = self.optimizeLattice(lattice, assigned_blobs)
409 
410             return optimized_lattice
411 
412         # Gradually expand the area for which the lattice is being optimized column by
    column
413         for n in range(1, self.cols, step):
414             bounds = (0, (n+1)*tw, 0, th)
415             lattice = optimizeWithBounds(self, lattice, bounds, max_blobs)
```

```
416             print('Lattice optimized for', n+1, 'of', self.cols, 'columns.')
417
418         # Gradually expand the area for which the lattice is being optimized row by row
419         for n in range(1, self.rows, step):
420             bounds = (0, self.cols*tw, 0, (n+1)*th)
421             lattice = optimizeWithBounds(self, lattice, bounds, max_blobs)
422             print('Lattice optimized for', n+1, 'of', self.rows, 'rows.')
423
424         # Run one last optimization, using a larger selection of blobs taken from the
    entire tileset
425         # Optimization is never done for all blobs, as this would take a very long time,
    and a random selection is
426         # sufficient if the selection is large enough.
427         bounds = (0, self.cols * tw, 0, self.rows * th)
428         lattice = optimizeWithBounds(self, lattice, bounds, final_blobs)
429         print('Final optimization finished.')
430
431         if debug:
432             self.lattice = lattice  # needs to be set for displayTileRegion
433             self.assignBlobs()
434             self.displayTileRegion(0, 0, 0, 0, blob_color='green', lattice_color='red')
435
436         self.lattice = lattice
437         self.saveLattice()
438         self.deleteAssignedBlobs()
439
440     def saveLattice(self):
441         """Save the lattice stored in self.lattice to a file located at self.path"""
442         full_path = self.path + '/lattice.p'
443         pickle.dump(self.lattice, open(full_path, 'wb'))
444
445     def deleteLattice(self):
446         """Delete the file and clear the variable containing the lattice."""
447         full_path = self.path + '/lattice.p'
448         try:
449             os.remove(full_path)
450         except FileNotFoundError:
451             print('No file to delete')
452         self.lattice = None
453
454     def getLattice(self):
455         """Obtain a lattice by whatever means necessary. Try the following order:
456         1: return self.lattice
457         2: load lattice from file
458         3: generate new lattice
```

```
459         """
460         if self.lattice != None:
461             return self.lattice
462         else:
463             try:
464                 full_path = self.path + '/lattice.p'
465                 self.lattice = pickle.load(open(full_path, 'rb'))
466                 if self.lattice == None:
467                     print('Loaded lattice, but array was empty.')
468                     self.makeLattice()
469             except FileNotFoundError:
470                 print('Lattice file not found!')
471                 self.makeLattice()
472 
473         return self.lattice
474 
475     def assignBlobs(self, blobs=None, lattice=None, save=True):
476         """Assign a set of blobs to a lattice. Each blob is assigned to it's nearest
    lattice point.
477         Return an array of dictionaries, each dictionary representing a blob, and
    containing the following:
478         ['blob']: y, x, and r of the blob
479         ['point']: lattice indices of the nearest lattice point
480         ['distance']: absolute distance to the nearest lattice point
481         ['angle']: angle of the displacement vector from blob to point
482 
483         :param blobs: the blobs to be assigned to lattice points, if none is given, self.
    getBlobs() is used
484         :param lattice: the lattice to which to assign the bobs, if none is given, self.
    getLattice() is used
485         :param save: if True, self.blobs will be set to the result, and assigned blobs
    will be saved to file
486                      if False, the result will be returned, but not saved
487         :return: described above
488         """
489         from scipy.spatial import KDTree
490         checkpoint = clock()
491 
492         if blobs == None:
493             blobs = self.getBlobs()
494         if lattice == None:
495             lattice = self.getLattice()
496 
497         assigned_blobs = [{'blob': blob} for blob in blobs]
498         radius = lattice.getMinLatticeDist()/2
```

```
499
500         x_min = min(blobs[:, 1]) - radius
501         x_max = max(blobs[:, 1]) + radius
502         y_min = min(blobs[:, 0]) - radius
503         y_max = max(blobs[:, 0]) + radius
504
505         points = lattice.getLatticePoints(x_min, x_max, y_min, y_max)
506         tree = KDTree(points)
507
508         for a_blob in assigned_blobs:
509             y, x, r = a_blob['blob']
510             distance, index = tree.query([x, y])
511             point = tree.data[index]
512             a_blob['point'] = lattice.getIndices(point[0], point[1])
513             a_blob['distance'] = distance
514             dis = np.array(point) - np.array([x, y])
515             a_blob['angle'] = np.angle(dis[0] + 1j * dis[1])
516
517         timeCheckpoint(checkpoint, 'assigning blobs')
518
519         if save:
520             self.assigned_blobs = assigned_blobs
521             self.saveAssignedBlobs()
522
523         return assigned_blobs
524
525     def saveAssignedBlobs(self):
526         """Store all currently assigned blobs to a file located at self.path."""
527         full_path = self.path + '/assigned_blobs.p'
528         pickle.dump(self.assigned_blobs, open(full_path, 'wb'))
529
530     def deleteAssignedBlobs(self):
531         """Delete the file and clear the variable containing assigned blobs."""
532         full_path = self.path + '/assigned_blobs.p'
533         try:
534             os.remove(full_path)
535         except FileNotFoundError:
536             print('No file to delete')
537         self.assigned_blobs = []
538
539     def getAssignedBlobs(self):
540         """Return assigned blobs for tileset. Load if possible, detect and assign if
    necessary."""
541         if len(self.assigned_blobs) > 0:
542             return self.assigned_blobs
```

```
543         else:
544             try:
545                 full_path = self.path + '/assigned_blobs.p'
546                 self.assigned_blobs = pickle.load(open(full_path, 'rb'))
547                 if len(self.assigned_blobs) < 1:
548                     print('Loaded assigned blobs, but array was empty. Assigning blobs.')
549                     blobs = self.getBlobs()
550                     lattice = self.getLattice()
551                     self.assignBlobs(blobs, lattice)
552             except FileNotFoundError:
553                 print('Assigned blobs file not found. Assigning.')
554                 blobs = self.getBlobs()
555                 lattice = self.getLattice()
556                 self.assignBlobs(blobs, lattice)
557 
558         if len(self.assigned_blobs) > 0:
559             return self.assigned_blobs
560         else:
561             raise Exception('Not able to obtain assigned blobs!')
562 
563     def getBlobCountPerPoint(self, region=None):
564         """
565         Get a list of all lattice points containing blobs, and the number of blobs they
    contain.
566 
567         :param region: list of 4 ints
568             if given, this denotes the limits of the subregion from which to get the list
     of points
569             if nothing is given, the whole tilset is used
570         :return: list of dicts
571             a list of dictionaries containing indices of each lattice point containing
    blobs, and the number of blobs
572             in it's neighborhood
573             the dictionary has the following elements:
574             'point': a list of 2 ints, representing the indices of the lattice point
575             'count': the number of blobs that have this lattice point as their nearest
    lattice point
576         """
577         checkpoint = clock()
578 
579         if region == None:
580             assignedBlobs = self.getAssignedBlobs()
581         elif len(region) != 4:
582             raise RuntimeError("'region' must have exactly 4 elements (x_min, x_max,
    y_min, y_max)")
```

```
583         else:
584             blobs = self.getSubsetOfBlobs(*region)
585             assignedBlobs = self.assignBlobs(blobs, self.getLattice())
586 
587         checkpoint = timeCheckpoint(checkpoint, 'get blobs')
588 
589         # Sort the blobs by lattice point
590         sortedBlobs = sorted(assignedBlobs, key = lambda x: (x['point'][0], x['point'
    ][1]))
591 
592         points = [{'indices': sortedBlobs[0]['point'], 'count': 1}]  # Initialize the
    dict
593 
594         for i, blob in enumerate(sortedBlobs):  # Go through the sorted blobs
595             if i == 0:
596                 continue  # Skip the first blob, as it is allready counted
597             if blob['point'] == sortedBlobs[i-1]['point']:  # If this blob belongs to the
     same point as the last blob
598                 points[-1]['count'] += 1  # Increment the count of the last listed point
    by 1
599             else:
600                 points.append({'indices': blob['point'], 'count': 1})  # Append a new
    point to the list
601 
602         timeCheckpoint(checkpoint, 'count points')
603 
604         return points
605 
606     def getYield(self, count=1):
607         """
608         Get the yield of n nanowires per point, default = 1 nanowire
609 
610         :param count:
611         :return:
612         """
613         lattice_points = self.getLattice().getLatticePoints(*self.getExtremes())
614         counts = self.getBlobCountPerPoint()
615 
616         total = len(lattice_points)
617         good = sum(1 for point in counts if point['count'] == count)
618 
619         return good/total
620 
621     def getBlobsOfCount(self, count):
622         """
```

```
623         Get all blobs located near lattice points with a given number of blobs in their
    neighborhood
624 
625         :param count: int > 1
626             the number of blobs that have to be in a lattice point's neighborhood for
    those blobs to be in the returned list
627         :return: a list on assigned blobs format containing blobs meeting the criterion
    described above
628         """
629         if count < 1:
630             raise RuntimeError('count must be 1 or larger')
631 
632         points = self.getBlobCountPerPoint()  # get a list of blob counts for each
    lattice point
633         # filter the list to only contain points with the desired blob count
634         points_with_count = [point['indices'] for point in points if point['count'] ==
    count]
635 
636         assigned_blobs = self.getAssignedBlobs()
637         # filter the list of blobs to only contain blobs belonging to points listed in
    the filtered point list
638         assigned_blobs_of_count = [a_blob for a_blob in assigned_blobs if a_blob['point']
     in points_with_count]
639 
640         return assigned_blobs_of_count
641 
642     def getExtremes(self, plus_radius=False, flip=False, region=None):
643         """
644         Get the coordinates limiting a set of blobs
645 
646         :param plus_radius:
647         :param flip:
648         :param region:
649         :return:
650         """
651         if region == None:
652             blobs = self.getBlobs()
653         else:
654             blobs = self.getSubsetOfBlobs(*region)
655 
656         if plus_radius:
657             r_max = blobs[:, 2].max()  # Largest radius
658         else:
659             r_max = 0
660 
```

```
661         x_min = blobs[:, 1].min() - r_max
662         x_max = blobs[:, 1].max() + r_max
663         y_min = blobs[:, 0].min() - r_max
664         y_max = blobs[:, 0].max() + r_max
665 
666         if flip:
667             return (x_min, x_max, y_max, y_min)
668         else:
669             return (x_min, x_max, y_min, y_max)
670 
671     def displayTileRegion(self, col_min, col_max, row_min, row_max, blob_color='red',
    lattice_color='cyan',
672                           connector_color='yellow', figsize=(24, 12), path='', hide_axes=
    False, feature_scale=1):
673         """Display a figure showing a region of the image, with blobs, lattice points and
     displacement vectors marked
674 
675         :param col_min:
676         :param col_max:
677         :param row_min:
678         :param row_max:
679         :param blob_color:
680         :param lattice_color:
681         :param connector_color:
682         :return:
683         """
684         checkpoint = clock()
685         total_checkpoint = clock()
686         tiles = self.getTileRegion(col_min, col_max, row_min, row_max)
687 
688         x_min = self.tilew * col_min
689         x_max = self.tilew * (col_max + 1) - 1
690         x_len = x_max - x_min
691         y_min = self.tileh * row_min
692         y_max = self.tileh * (row_max + 1) - 1
693         y_len = y_max - y_min
694         checkpoint = timeCheckpoint(checkpoint, 'initialize')
695 
696         blobs = self.getSubsetOfBlobs(x_min, x_max, y_min, y_max)
697 
698         checkpoint = timeCheckpoint(checkpoint, 'filter blobs')
699 
700         fig, ax = plt.subplots(figsize=figsize)
701         ax.set_aspect('equal', adjustable='box-forced')
702         plt.axis((x_min, x_max, y_max, y_min))
```

```
703         checkpoint = timeCheckpoint(checkpoint, 'setup plot')
704 
705         plt.imshow(tiles, extent=[x_min, x_max, y_max, y_min], cmap='gray', interpolation
    ='nearest')
706         checkpoint = timeCheckpoint(checkpoint, 'plot tiles')
707         f.plotCircles(ax, blobs, fig, dict(color=blob_color, linewidth=1*feature_scale,
    fill=False))
708         checkpoint = timeCheckpoint(checkpoint, 'plot blobs')
709 
710         if self.lattice:
711             lattice = self.getLattice()
712             points = self.lattice.getLatticePoints(x_min, x_max, y_min, y_max)
713             flip_points = np.fliplr(points)
714             f.plotCircles(ax, flip_points, fig, dict(color=lattice_color, linewidth=5*
    feature_scale, fill=True))
715             checkpoint = timeCheckpoint(checkpoint, 'plot lattice')
716 
717             assigned_blobs = self.getAssignedBlobs()
718             checkpoint = timeCheckpoint(checkpoint, 'get assigned blobs')
719 
720             from matplotlib.collections import LineCollection
721             from matplotlib.colors import colorConverter
722 
723             connectors = np.zeros((len(assigned_blobs), 2, 2), float)
724             for i, a_blob in enumerate(assigned_blobs):
725                 if len(a_blob['point']) > 0:
726                     bx = a_blob['blob'][1]
727                     by = a_blob['blob'][0]
728                     [px, py] = lattice.getCoordinates(*a_blob['point'])
729                     connectors[i, :, :] = [[bx, by], [px, py]]
730 
731             colors = colorConverter.to_rgba(connector_color)
732             line_segments = LineCollection(connectors, colors=colors, linewidths=1*
    feature_scale)
733             ax.add_collection(line_segments)
734 
735         timeCheckpoint(total_checkpoint, 'total time')
736 
737         if hide_axes:
738             ax.set_yticklabels([])
739             ax.set_xticklabels([])
740             plt.tight_layout()
741 
742         if path == '':
743             plt.show()
```

```
744             plt.close()
745         else:
746             plt.savefig(path)
747             print('Saved figure to', path)
748 
749     def plotBlobs(self, col, row):
750         """Display a figure showing a given tile, and blobs detected for that tile
751 
752         :param col: column number of the tile to be displayed
753         :param row: row number of the tile to be displayed
754         """
755         blobs = self.detectBlobs(col, row)
756         tile = self.getTile(col, row)
757 
758         fig, ax = plt.subplots(figsize=(24, 12))
759         ax.set_aspect('equal', adjustable='box-forced')
760         plt.axis((0, 1023, 1023, 0))
761 
762         plt.imshow(tile, cmap='gray', interpolation='nearest')
763         f.plotCircles(ax, blobs, fig, dict(color='red', linewidth=1, fill=False))
764         ax.set_yticklabels([])
765         ax.set_xticklabels([])
766         plt.tight_layout()
767 
768         plt.show()
769         plt.close()
770 
771     def plotBlobRegion(self, col_min=0, col_max=None, row_min=0, row_max=None, property='
    radius', hide_axes=False, colormap='',
772                        bg_color='', auto_limits=False):
773         """Show a figure plotting all detected blobs from the specified tile region
    without any background image
774 
775         :param col_min:
776         :param col_max:
777         :param row_min:
778         :param row_max:
779         :param property:
780         :return:
781         """
782         if col_max == None:
783             col_max = self.cols - 1
784         if row_max == None:
785             row_max = self.rows - 1
786 
```

```
787         checkpoint = clock()
788         x_min = self.tilew * col_min
789         x_max = self.tilew * (col_max + 1) - 1
790         y_min = self.tileh * row_min
791         y_max = self.tileh * (row_max + 1) - 1
792 
793         label = ''
794         if property == 'radius' or property == 'diameter':
795             property = 'radius'
796             label = 'diameter [nm]'
797             if colormap == '': colormap = 'jet'
798         elif property == 'displacement' or property == 'distance':
799             property = 'distance'
800             label = 'Displacement from lattice point [nm]'
801             if colormap == '': colormap = 'viridis'
802         elif property == 'angle':
803             label = 'Angle of displacement from lattice point'
804             if colormap == '': colormap = 'hsv'
805         else:
806             raise RuntimeError("Invalid property '" + str(property) + "'")
807 
808         checkpoint = timeCheckpoint(checkpoint, 'setup')
809 
810         def isInside(a_blob, x_min, x_max, y_min, y_max):
811             inside = False
812             blob_x = a_blob['blob'][1]
813             blob_y = a_blob['blob'][0]
814             if x_min <= blob_x <= x_max and y_min <= blob_y <= y_max:
815                 inside = True
816 
817             return inside
818 
819         assigned_blobs = self.getAssignedBlobs()
820         assigned_blobs = [a_blob for a_blob in assigned_blobs if isInside(a_blob, x_min,
    x_max, y_min, y_max)]
821 
822         blobs = np.zeros((len(assigned_blobs), 4))
823         for i, a_blob in enumerate(assigned_blobs):
824             blobs[i, 0] = a_blob['blob'][0]
825             blobs[i, 1] = a_blob['blob'][1]
826             blobs[i, 2] = self.getLattice().getMinLatticeDist() * 0.5
827             if property == 'radius':
828                 blobs[i, 3] = a_blob['blob'][2] * 2 * self.scale
829             elif property == 'distance':
830                 blobs[i, 3] = a_blob['distance'] * self.scale
```

```
831             elif property == 'angle':
832                 blobs[i, 3] = a_blob['angle']
833 
834         checkpoint = timeCheckpoint(checkpoint, 'getting stuff')
835 
836         fig, ax = plt.subplots(figsize=(12, 6))
837         ax.set_aspect('equal', adjustable='box-forced')
838 
839         if auto_limits:
840             plt.axis(self.getExtremes(plus_radius=True, flip=True))
841         else:
842             plt.axis((x_min, x_max, y_max, y_min))
843 
844         from matplotlib.collections import PatchCollection
845 
846         patches = []
847         colors = []
848 
849         checkpoint = timeCheckpoint(checkpoint, 'figure setup')
850 
851         for circle in blobs:
852             y, x, r, c = circle
853             colors.append(c)
854             patch = plt.Circle((x, y), r, linewidth=0, fill=True)
855             patches.append(patch)
856 
857         checkpoint = timeCheckpoint(checkpoint, 'figure loop')
858 
859         p = PatchCollection(patches, match_original=True, cmap=colormap)
860         p.set_array(np.array(colors))
861         fig.colorbar(p, ax=ax, label=label)
862         ax.add_collection(p)
863 
864         checkpoint = timeCheckpoint(checkpoint, 'figure end')
865 
866         plt.tight_layout()
867         if hide_axes:
868             ax.set_yticklabels([])
869             ax.set_xticklabels([])
870         if bg_color != '':
871             ax.set_axis_bgcolor(bg_color)
872 
873         plt.show()
874         plt.close()
875 
```

```
876     def plotBlobCountPerPoint(self, region=None, only_ones=False):
877         counts = self.getBlobCountPerPoint(region)
878         if only_ones:
879             counts = [point for point in counts if point['count']==1]
880         lattice = self.getLattice()
881 
882         blobs = np.zeros((len(counts), 4))
883         for i, point in enumerate(counts):
884             coordinates = lattice.getCoordinates(point['indices'][0], point['indices'
    ][1])
885             blobs[i, 0] = coordinates[1]
886             blobs[i, 1] = coordinates[0]
887             blobs[i, 2] = self.getLattice().getMinLatticeDist() * 0.55
888             blobs[i, 3] = point['count']
889 
890         fig, ax = plt.subplots(figsize=(11, 6))
891         ax.set_aspect('equal', adjustable='box-forced')
892         plt.axis(self.getExtremes(plus_radius=True, flip=True, region=region))
893 
894         from matplotlib.collections import PatchCollection
895 
896         patches = []
897         colors = []
898 
899         for circle in blobs:
900             y, x, r, c = circle
901             colors.append(c)
902             patch = plt.Circle((x, y), r, linewidth=0, fill=True)
903             patches.append(patch)
904 
905         p = PatchCollection(patches, match_original=True, cmap='jet')
906         p.set_array(np.array(colors))
907         fig.colorbar(p, ax=ax, ticks=range(0, 10), label='Blobs per lattice point')
908         ax.add_collection(p)
909         ax.set_yticklabels([])
910         ax.set_xticklabels([])
911 
912         plt.tight_layout()
913         plt.show()
914         plt.close()
915 
916     def printYields(self, region=None):
917         extremes = self.getExtremes(region=region)
918         lattice_points = self.getLattice().getLatticePoints(*extremes)
919         counts = self.getBlobCountPerPoint(region)
```

```
920
921         total = len(lattice_points)
922         print(total)
923         empty = total
924
925         for n in range(1, 10):
926             good = sum(1 for point in counts if point['count'] == n)
927             empty -= good
928             ratio = good / total * 100
929
930             print('Yield ', n, ': ', ratio, sep='')
931
932         ratio = empty / total * 100
933         print('Yield 0:', ratio)
934
935     def plotHistogram(self, property, bins=100, fontsize=16, normalized=True):
936         """Plot a histogram of a given property of the detected blobs
937
938         :param property: the property to be plotted. Can be either 'diameter', 'distance'
        or 'angle'
939         :param bins: the number of bins used for the histogram
940         :param fontsize: size of the font used in the plot
941         :param normalized: when plotting displacement distance, determines weather to
    normalize histogram bins by the
942                            area they represent, to get a plot of radial density
943         """
944         if property == 'diameter':
945             label = 'diameter [nm]'
946             blobs = self.getBlobs()
947             data = blobs[:, 2] * self.scale * 2
948             normalized = False
949         elif property == 'distance':
950             label = 'displacement from lattice point [nm]'
951             assigned_blobs = self.getAssignedBlobs()
952             data = [a_blob['distance'] * self.scale for a_blob in assigned_blobs]
953         elif property == 'angle':
954             label = 'angle'
955             assigned_blobs = self.getAssignedBlobs()
956             data = [a_blob['angle'] for a_blob in assigned_blobs]
957             normalized = False
958         else:
959             raise ValueError("'" + property + "' is not a valid property")
960
961         fig, ax = plt.subplots(1, 1, figsize=(9, 6), subplot_kw={'adjustable': 'box-
    forced'})
```

```
962
963         plt.grid()
964         plt.xlabel(label, fontsize=fontsize)
965
966         if normalized:
967             hist, bin_edges = np.histogram(data, bins=bins)
968             adjusted_hist = np.zeros(hist.shape[0])
969
970             for i, count in enumerate(hist):
971                 r0 = bin_edges[i]  # inner radius of the region represented by the bin
972                 r1 = bin_edges[i+1]  # outer radius of the region represented by the bin
973                 # divide the counts by the area of the region
974                 adjusted_hist[i] = float(count) / ( (pi * r1**2) - (pi*r0**2) )
975
976             center = (bin_edges[:-1] + bin_edges[1:]) / 2
977             width = (bin_edges[1] - bin_edges[0])
978             ax.bar(center, adjusted_hist, align='center', width=width, edgecolor='none',
    color='#033A87')
979
980             plt.ylabel('density', fontsize=fontsize)
981             ax.get_yaxis().set_ticks([])
982         else:
983             ax.hist(data, bins=bins, histtype='stepfilled', edgecolor='none', color='#033
    A87')
984             plt.ylabel('count', fontsize=fontsize)
985
986         for tick in ax.xaxis.get_major_ticks():
987             tick.label.set_fontsize(fontsize)
988
989         for tick in ax.yaxis.get_major_ticks():
990             tick.label.set_fontsize(fontsize)
991
992         plt.tight_layout()
993         plt.show()
994
995     def plotRadialHistogram(self, bins=90, fontsize=20):
996         """Plot a radial histogram of the displacement angles of the detected blobs
997
998         :param bins: the number of bins used for the histogram
999         :param fontsize: size of the font used in the plot
1000        """
1001        assigned_blobs = self.getAssignedBlobs()
1002        data = [a_blob['angle'] for a_blob in assigned_blobs]
1003
1004        plt.figure(figsize=[8, 8])
```

```
1005            ax = plt.subplot(111, projection='polar')
1006
1007            ax.hist(data, bins=bins, histtype='stepfilled', edgecolor='none', color='#033A87'
        )
1008
1009            for tick in ax.xaxis.get_major_ticks():
1010                tick.label.set_fontsize(fontsize)
1011
1012            for tick in ax.yaxis.get_major_ticks():
1013                tick.label.set_fontsize(fontsize)
1014
1015            plt.show()
1016
1017        def scatterPlotDisplacements(self):
1018            assigned_blobs = self.getAssignedBlobs()
1019            angles = [blob['angle'] for blob in assigned_blobs]
1020            displacements = [blob['distance'] * self.scale for blob in assigned_blobs]
1021
1022            fig, ax = plt.subplots(figsize=(6, 6), subplot_kw={'projection': 'polar'})
1023
1024            ax.scatter(angles, displacements, color='mediumblue', alpha=1, s=10, edgecolor='
        none')
1025            ax.set_ylim([0, 500])
1026            plt.show()
1027
1028        def scatterPlotDisplacementsFiltered(self):
1029            assigned_blobs = self.getAssignedBlobs()
1030            assigned_blobs = [a_blob for a_blob in assigned_blobs if 185 < a_blob['blob'
        ][2]*2*self.scale < 202]
1031            angles = [blob['angle'] for blob in assigned_blobs]
1032            displacements = [blob['distance'] * self.scale for blob in assigned_blobs]
1033
1034            fig, ax = plt.subplots(figsize=(6, 6), subplot_kw={'projection': 'polar'})
1035
1036            ax.scatter(angles, displacements, color='mediumblue', alpha=0.5, s=3, edgecolor='
        none')
1037            ax.set_ylim([0, 500])
1038            plt.show()
1039
1040        def scatterSizeVsDisplacement(self):
1041            assigned_blobs = self.getAssignedBlobs()
1042            sizes = [blob['blob'][2] * 2 * self.scale for blob in assigned_blobs]
1043            displacements = [blob['distance'] * self.scale for blob in assigned_blobs]
1044
1045            fig, ax = plt.subplots(figsize=(12, 8))
```

```
1046            plt.xlabel('Droplet diameter [nm]')
1047            plt.ylabel('Displacement from lattice point [nm]')
1048            plt.grid()
1049
1050            ax.scatter(sizes, displacements, color='mediumblue', alpha=0.5, s=3, edgecolor='
        none')
1051            ax.set_ylim([0, 500])
1052            plt.show()
1053
1054        def scatterPlotDisplacementsByCount(self, count):
1055            assigned_blobs = self.getBlobsOfCount(count)
1056            angles = [blob['angle'] for blob in assigned_blobs]
1057            displacements = [blob['distance'] * self.scale for blob in assigned_blobs]
1058
1059            fig, ax = plt.subplots(figsize=(6, 6), subplot_kw={'projection': 'polar'})
1060
1061            ax.scatter(angles, displacements, color='mediumblue', alpha=0.5, s=3, edgecolor='
        none')
1062            ax.set_ylim([0, 500])
1063            plt.show()
1064
1065        @staticmethod
1066        def showMap(map):
1067            plt.imshow(map, cmap='viridis')
1068            plt.colorbar(label='Average droplet diameter [nm]')
1069            plt.gca().get_xaxis().set_ticks([])
1070            plt.gca().get_yaxis().set_ticks([])
1071            plt.gca().set_axis_bgcolor('black')
1072            plt.show()
1073            plt.close()
1074
1075        def getDensityMap(self, scale_factor, radius):
1076            from math import floor, ceil, pi
1077
1078            sf = scale_factor
1079            r = ceil(radius/scale_factor)
1080            d = 2 * r  # diameter
1081
1082            x_min = self.tilew * 0
1083            x_max = self.tilew * self.cols - 1
1084            x_len = x_max - x_min
1085            y_min = self.tileh * 0
1086            y_max = self.tileh * self.rows - 1
1087            y_len = y_max - y_min
1088
```

```
1089            blobs = self.getBlobs()
1090
1091            add_array = f.getCircleOfOnes(r)
1092
1093            blob_points = [(floor(blob[0] / sf), floor(blob[1] / sf)) for blob in blobs]
1094
1095            data = np.zeros((ceil(y_len / sf) + d, ceil(x_len / sf) + d))
1096
1097            for point in blob_points:
1098                data[point[0]:point[0] + d, point[1]:point[1] + d] += add_array
1099
1100            px_area = self.scale**2 / 10**6
1101            c_area = pi * radius**2 * px_area
1102            data = data / c_area
1103
1104            return data
1105
1106        def getRadiusMap(self, scale_factor, radius):
1107            from math import floor, ceil
1108
1109            sf = scale_factor
1110            r = ceil(radius/scale_factor)
1111            d = 2 * r  # diameter
1112
1113            x_min = self.tilew * 0
1114            x_max = self.tilew * self.cols - 1
1115            x_len = x_max - x_min
1116            y_min = self.tileh * 0
1117            y_max = self.tileh * self.rows - 1
1118            y_len = y_max - y_min
1119
1120            blobs = self.getBlobs()
1121
1122            add_array = f.getCircleOfOnes(r)
1123
1124            blob_points = [(floor(blob[0] / sf), floor(blob[1] / sf), blob[2] * 2 * self.
        scale) for blob in blobs]
1125
1126            data = np.zeros((ceil(y_len / sf) + d, ceil(x_len / sf) + d))
1127
1128            for point in blob_points:
1129                data[point[0]:point[0] + d, point[1]:point[1] + d] += add_array*point[2]
1130
1131            return data
1132
```

```
1133     def plotDensity(self, scale_factor, radius):
1134         map = self.getDensityMap(scale_factor, radius)
1135 
1136         A = np.argwhere(map)
1137         (y_start, x_start), (y_stop, x_stop) = A.min(0), A.max(0) + 1
1138         map = map[y_start:y_stop, x_start:x_stop]
1139 
1140         self.showMap(map)
1141 
1142     def plotRadius(self, scale_factor, radius):
1143         r = self.getRadiusMap(scale_factor, radius)
1144         d = self.getDensityMap(scale_factor, radius)
1145         map = r / d
1146 
1147         crop_map = np.where(np.isnan(map), 0, 1)
1148         A = np.argwhere(crop_map)
1149         (y_start, x_start), (y_stop, x_stop) = A.min(0), A.max(0) + 1
1150         map = map[y_start:y_stop, x_start:x_stop]
1151 
1152         self.showMap(map)
1153 
1154 
1155 def createTilesFromImage(path, image_name, tilew=1024, tileh=1024):
1156     """Cut the given image into tiles of the specified size, and store them in specified
     path.
1157 
1158     :param path:
1159     :param image_name:
1160     :param tilew:
1161     :param tileh:
1162     :return:
1163     """
1164     from math import ceil
1165 
1166     image_path = path + '/' + image_name
1167     image = misc.imread(image_path, flatten=True)
1168 
1169     print(image.shape)
1170     im_h = image.shape[0]
1171     im_w = image.shape[1]
1172 
1173     rows = ceil(im_h / tileh)
1174     cols = ceil(im_w / tilew)
1175 
1176     padded_height = tileh * rows
```

```
    padded_width  = tilew * cols

    padding_bottom = padded_height - im_h
    padding_right  = padded_width  - im_w

    padded_image = np.pad(image, ((0, padding_bottom), (0, padding_right)), 'constant')

    for c in range(0, cols):
        col = []

        col_path = path + '/c_' + str(c)
        if not os.path.exists(col_path):
            os.makedirs(col_path)

        for r in range(0, rows):
            tile = padded_image[r*tileh:r*tileh+1024, c*tilew:c*tilew+1024]
            col.append(tile)

            filename = col_path + '/tile_' + str(r) + '.png'
            misc.imsave(filename, tile)
            print('Saved tile ', c, ', ', r, sep='')
```

# lattice.py

```
import numpy as np
from math import floor
import pickle

class Lattice:

    def __init__(self, Na, Nb, vec_a, vec_b, offset):
        self.Na = Na
        self.Nb = Nb
        self.vec_a = np.array(vec_a)
        self.vec_b = np.array(vec_b)
        self.vec_c = self.vec_a - self.vec_b
        self.offset = offset

    def getParams(self):
        """Return a tuple of all relevant parameters"""
        return self.Na, self.Nb, self.vec_a, self.vec_b, self.offset

    def getMinLatticeDist(self):
        """Return magnitude of shortest lattice vector"""
```

```
21          return min(np.linalg.norm(self.vec_a), np.linalg.norm(self.vec_b), np.linalg.norm
    (self.vec_c))
22
23      def getMaxLatticeDist(self):
24          """Return magnitude of longest lattice vector"""
25          return max(np.linalg.norm(self.vec_a), np.linalg.norm(self.vec_b), np.linalg.norm
    (self.vec_c))
26
27      def getLatticePoints(self):
28          """Return a list of coordinates of all points in the lattice"""
29          points = []
30          for i in range(0, self.Na):
31              for j in range(0, self.Nb):
32                  points.append([ (i-floor(j/2))*self.vec_a[0] + j*self.vec_b[0] + self.
    offset[0],
33                                  (i-floor(j/2))*self.vec_a[1] + j*self.vec_b[1] + self.
    offset[1] ])
34
35          return points
36
37      def save(self, filename):
38          """Save the parameters of the lattice"""
39          params = [self.Na, self.Nb, self.vec_a, self.vec_b, self.offset]
40          pickle.dump(params, open(filename, 'wb'))
41
42  def loadLattice(filename):
43      """Return a lattice made from parameters in a file"""
44      Na, Nb, vec_a, vec_b, offset = pickle.load(open( filename, 'rb'))
45      lattice = Lattice(Na, Nb, vec_a, vec_b, offset)
46
47      return lattice
```

# arbitraryLattice.py

```
1  import numpy as np
2  from math import floor, ceil
3  import pickle
4
5  class Lattice:
6
7      def __init__(self, vec_a, vec_b, offset):
8          self.vec_a = np.array(vec_a)
9          self.vec_b = np.array(vec_b)
10         self.vec_c = self.vec_a - self.vec_b
```

```
        self.len_a = np.linalg.norm(self.vec_a)
        self.len_b = np.linalg.norm(self.vec_b)
        self.len_c = np.linalg.norm(self.vec_c)
        self.ang_a = np.angle(self.vec_a[0] + 1j*self.vec_a[1])
        self.ang_b = np.angle(self.vec_b[0] + 1j*self.vec_b[1])
        self.ang_c = np.angle(self.vec_c[0] + 1j*self.vec_c[1])
        self.offset = np.array(offset)

    def getParams(self):
        """Return a tuple of all relevant parameters"""
        return self.vec_a, self.vec_b, self.offset

    def getMinLatticeDist(self):
        """Return magnitude of shortest lattice vector"""
        return min(np.linalg.norm(self.vec_a), np.linalg.norm(self.vec_b))

    def getMaxLatticeDist(self):
        """Return magnitude of longest lattice vector"""
        return max(np.linalg.norm(self.vec_a), np.linalg.norm(self.vec_b))

    def decompose(self, subject, vec_a, vec_b):
        """Decompose a given vector into a linear combination of two other given vectors
    """
        x = np.transpose(np.array([vec_a, vec_b]))
        y = np.array(subject)
        ans = np.linalg.solve(x, y)

        return ans

    def getLatticePoints(self, x_min, x_max, y_min, y_max):
        """Return an array of coordinates of all points in the lattice bounded by the
    given x and y values"""
        offset = np.array(self.offset)
        corners = [[x_min, y_min], [x_max, y_min], [x_max, y_max], [x_min, y_max]] #
    coordinates of the corners of the region
        displacements = [np.array(corner) - offset for corner in corners] # displacements
     of the region corners from the offset point

        dd = np.array([self.decompose(displacement, self.vec_a, self.vec_b) for
    displacement in displacements]) # dd = decomposed displacements

        a_min = floor(min(dd[:,0]))
        a_max = ceil(max(dd[:,0]))
        b_min = floor(min(dd[:,1]))
        b_max = ceil(max(dd[:,1]))
```

```
51
52          points = []
53          for i in range(a_min, a_max + 1):
54              for j in range(b_min, b_max + 1):
55                  point = offset + self.vec_a*i + self.vec_b*j
56                  if point[0] > x_min and point[0] < x_max and point[1] > y_min and point
    [1] < y_max:
57                      points.append(offset + self.vec_a*i + self.vec_b*j)
58
59          return np.array(points)
60
61      def getCoordinates(self, a, b):
62          """Return the coordinates of a lattice point with the given indeces"""
63          return self.offset + a*self.vec_a + b*self.vec_b
64
65      def getIndices(self, x, y, roundIndices=True):
66          """Return the indices of a lattice point with the given coordinates"""
67          displacement = np.array([x, y]) - self.offset
68          indices = self.decompose(displacement, self.vec_a, self.vec_b)
69          if roundIndices:
70              rounded = [round(index) for index in indices]
71              indices = rounded
72
73          return indices
74
75      def save(self, filename):
76          """Save the parameters of the lattice"""
77          params = [self.vec_a, self.vec_b, self.offset]
78          pickle.dump(params, open(filename, 'wb'))
79
80  def loadLattice(filename):
81      """Return a lattice made from parameters in a file"""
82      vec_a, vec_b, offset = pickle.load(open( filename, 'rb'))
83      lattice = Lattice(vec_a, vec_b, offset)
84
85      return lattice
86
87  def makeLatticeByAngles(mag_a, ang_a, mag_b, ang_b, offset):
88      """Initialize an ArbitraryLattice class by giving angles and magnitudes of the
    lattice vectors"""
89      from math import sin, cos
90
91      vec_a = mag_a * np.array([cos(ang_a), sin(ang_a)])
92      vec_b = mag_b * np.array([cos(ang_b), sin(ang_b)])
93      lattice = Lattice(vec_a, vec_b, offset)
```

```
94
95      return lattice
```

# detect.py

```
1  import numpy as np
2  import cv2
3
4  def detect(image,
5             invert,
6             minThreshold=0,
7             maxThreshold=255,
8             thresholdStep=1,
9             minDistBetweenBlobs=0,
10            filterByArea=False,
11            minArea=0,
12            maxArea=None,
13            filterByCircularity=False,
14            minCircularity=0.0,
15            maxCircularity=None,
16            filterByConvexity=False,
17            minConvexity=0.0,
18            maxConvexity=None,
19            filterByInertia=False,
20            minInertiaRatio=0.0,
21            maxInertiaRatio=None
22            ):
23      """Wrapper for the cv2.SimpleBlobDetector method"""
24
25      if invert:
26          image = 255 - image
27
28      params = cv2.SimpleBlobDetector_Params()
29      params.minThreshold = minThreshold
30      params.maxThreshold = maxThreshold
31      params.thresholdStep = thresholdStep
32      params.minDistBetweenBlobs = minDistBetweenBlobs
33      params.filterByArea = filterByArea
34      params.minArea = minArea
35      params.maxArea = maxArea
36      params.filterByCircularity = filterByCircularity
37      params.minCircularity = minCircularity
38      params.maxCircularity = maxCircularity
39      params.filterByConvexity = filterByConvexity
```

```
40      params.minConvexity = minConvexity
41      params.maxConvexity = maxConvexity
42      params.filterByInertia = filterByInertia
43      params.minInertiaRatio = minInertiaRatio
44      params.maxInertiaRatio = maxInertiaRatio
45
46      # Set up the detector with the given parameters.
47      detector = cv2.SimpleBlobDetector_create(params)
48      # Do the detection
49      keypoints = detector.detect(image)
50
51      blobs = np.zeros((len(keypoints), 3))
52
53      # This is to make the output work with other code
54      for i, keypoint in enumerate(keypoints):
55          blobs[i][0] = keypoint.pt[1]
56          blobs[i][1] = keypoint.pt[0]
57          blobs[i][2] = keypoint.size / 2
58
59      return blobs
60
61  def droplets(image):
62      """Used to detect droplets on FIB arrays"""
63
64      blobs = detect(image,
65                     invert=False,
66                     minThreshold=50,
67                     filterByArea=True,
68                     minArea=200,
69                     filterByCircularity=True,
70                     minCircularity=0.85,
71                     filterByInertia=True,
72                     minInertiaRatio=0.8
73                     )
74
75      return blobs
76
77  def wiresWithoutDroplets(image):
78      """Used to detect wires without droplets on FIB arrays"""
79
80      blobs = detect(image,
81                     invert=True,
82                     maxThreshold=130,
83                     filterByArea=True,
84                     minArea=20,
```

```
85                      filterByCircularity=True,
86                      minCircularity=0.7,
87                      filterByConvexity=True,
88                      minConvexity=0.9,
89                      )
90
91      return blobs
92
93  def tiled(image):
94      """Used to detect droplets in the first NIL dataset"""
95
96      blobs = detect(image,
97                      invert=True,
98                      maxThreshold=200,
99                      filterByArea=True,
100                     minArea=40,
101                     filterByCircularity=True,
102                     minCircularity=0.8,
103                     filterByConvexity=True,
104                     minConvexity=0.9,
105                     )
106
107     return blobs
108
109 def tiled_2(image):
110     """Used to detect droplets in the second NIL dataset"""
111
112     blobs = detect(image,
113                     invert=True,
114                     maxThreshold=200,
115                     filterByArea=True,
116                     minArea=200,
117                     filterByCircularity=True,
118                     minCircularity=0.85,
119                     filterByConvexity=True,
120                     minConvexity=0.9,
121                     )
122
123     return blobs
124
125 def random(image):
126     """Used to detect droplets in the random growth dataset"""
127
128     blobs = detect(image,
129                     invert=True,
```

```
                    maxThreshold=200,
                    filterByArea=True,
                    minArea=40,
                    maxArea=450,
                    filterByCircularity=True,
                    minCircularity=0.7
                    )

    return blobs
```

# functions.py

```
import numpy as np
import matplotlib.pyplot as plt

def isInCircle(point_x, point_y, circle_x, circle_y, radius):
    """Check if a given point is within a given circle"""
    return ( (point_x - circle_x)**2 + (point_y - circle_y)**2 ) < radius**2

def fillWires(image):
    """Performs hole filling by morphological reconstruction by erosion on the given
    image"""
    from skimage.morphology import reconstruction

    seed = np.copy(image)
    seed[1:-1, 1:-1] = image.max()
    mask = image

    reconstructed = reconstruction(seed, mask, method='erosion')

    return reconstructed

def plotCircles(axes, circles, fig, kwargs):
    """Plots a collection of circles on the given axes"""
    from matplotlib.collections import PatchCollection

    patches = []

    for circle in circles:
        if len(circle) == 3:
            y, x, r = circle
        elif len(circle) == 2:
            y, x = circle
            r = 1
```

```
32          else:
33              raise RuntimeError('Wrong number of elements to define circle: ' + str(len(
    circle)))
34          patch = plt.Circle((x, y), r, **kwargs)
35          patches.append(patch)
36
37      p = PatchCollection(patches, match_original=True)
38      axes.add_collection(p)
39
40  def randomColors(n):
41      """Returns a list of n arrays that can be used as random colors"""
42      colors = []
43      for x in range(0, n):
44          colors.append(np.random.rand(3,1))
45
46      return colors
47
48  def surfacePlot(data, title='', colorbar_label='', percentages=False, real_axes=False):
49      """Plot type used for plotting properties of all fields in an array"""
50      plt.figure(figsize=(6.5, 5))
51      ax = plt.gca()
52
53      if real_axes:
54          X = np.arange(5, 95, 10)
55          Y = np.linspace(0.416-(0.416/2),3.328+(0.416/2),9)
56      else:
57          X = np.arange(0.5, 9.5)
58          Y = np.arange(0.5, 9.5)
59
60      X, Y = np.meshgrid(X, Y)
61      Z = np.array(data).reshape((8, 8)) # Makes Z a 8x8 2d array
62      Z = np.transpose(Z) # Use this if fluence and diameter are flipped. Otherwise comment
     out.
63      plt.pcolor(X, Y, Z, cmap='viridis')
64
65      plt.title(title)
66
67      if real_axes:
68          from matplotlib.ticker import FormatStrFormatter
69
70          plt.xlabel('Diameter [nm]')
71          plt.ylabel(r'Fluence [10$^3$ ions/nm$^2$]')
72          plt.axis([5, 85, (0.416 / 2), 3.328 + (0.416 / 2)])
73          plt.gca().yaxis.set_major_formatter(FormatStrFormatter('%.1f'))
74          plt.yticks(np.arange(1, 9)*0.41610061)
```

```
75
76      else:
77          plt.xlabel('Diameter')
78          plt.ylabel('Dose')
79          plt.axis([0.5, 8.5, 0.5, 8.5])
80          ax.set_aspect('equal', adjustable='box-forced')
81
82      if percentages:
83          plt.colorbar(format='%1.0f %%', label=colorbar_label)
84      else:
85          plt.colorbar(label=colorbar_label)
86
87      return plt
88
89  def getCircleOfOnes(radius):
90      """Returns an array where all elements are 0 except for elements within a circle of
        the given radius, which are 1"""
91      diameter = 2 * radius
92      ones = np.ones((diameter, diameter))
93      y, x = np.ogrid[-radius: radius, -radius: radius]
94      mask = x ** 2 + y ** 2 <= radius ** 2
95      circle = ones * mask
96
97      return circle
```