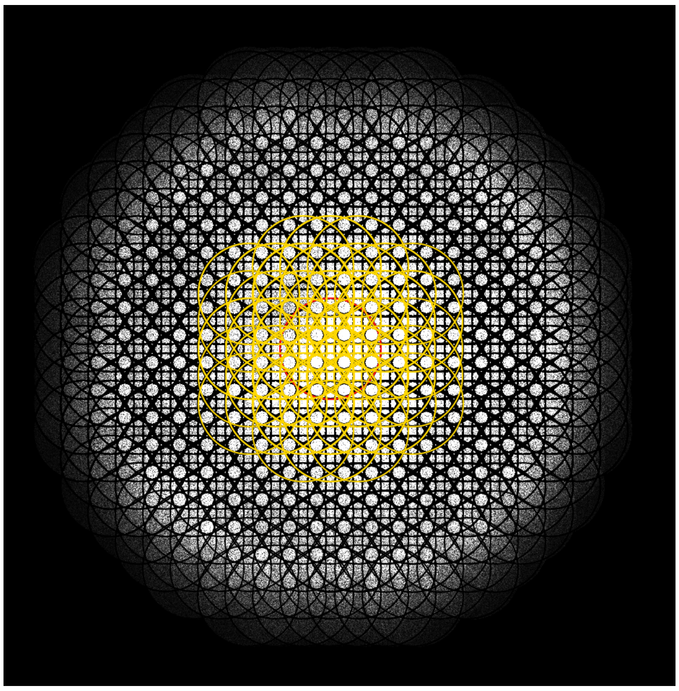
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Øystein Rognebakke Krogstie

Dynamic Task Parallelism in FPM and Related Methods

Master's thesis in MTDT
Supervisor: Anne C. Elster
September 2021



NTNU
Norwegian University of
Science and Technology

Øystein Rognebakke Krogstie

Dynamic Task Parallelism in FPM and Related Methods

Master’s thesis in MTDT
Supervisor: Anne C. Elster
September 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Dynamic Task Parallelism in FPM and Related Methods

Øystein R. Krogstie

2021-09-06

Project description

Implementations of Fourier ptychographic microscopes are slowed primarily by image acquisition time, memory movement, and image processing time. A goal in Fourier ptychographic imaging is real time operation. To achieve this, all of these aspects must be accelerated. In this thesis, both the use of reduced precision parallel algorithms for the image processing aspect, and compression and streaming techniques for the memory movement aspect are explored and implemented in an integrated Fourier microscopy system. Other related algorithms and techniques be also be considered.

Abstract

Microscopy allows us to see details too small for the naked eye, but the area of images are strongly limited by the magnification level. In fields such as medical imaging and materials science, however, both high magnification levels and larger images are desired. Increasing both resolution and image size in a conventional microscope requires either more expensive optical components, or large and complex setups. In the last decades, using computing rather than more expensive optical setups to improve imaging systems, so called "computational imaging" has become popular. A recent technique in this field is Fourier Ptychographic Microscopy (FPM). In FPM, an object is illuminated from different angles and imaged through a regular light microscope. The obtained set of images are combined into a single higher resolution result image by solving a phase recovery problem. However, this recovery problem is very computationally intensive, prohibiting real time operation.

In this thesis, we present a modified iterative solver for phase recovery in the context of FPM that permits concurrent execution of sub-iterations, detecting and exploiting task dynamic parallelism using a priority queue containing a dependency graph and a cost function. This speeds up both the recovery, and exposes more data parallelism in the FPM recovery. Our novel implementation obtains a speedup of x5 on a 16 core CPU even without exploiting any available data parallelism, and at only a small reduction in reconstruction quality compared to a single threaded baseline. As many phase recovery algorithms in both FPM and ptychography have similar structures, this method shows promise as a more broadly applicable tool. Suggestions for future work are also included.

Sammendrag

Mikroskopi lar oss se detaljer som vanligvis er for små til å se med det blotte øye, men størrelsen på området som avbildes begrenses av forstørrelsesgraden. Innenfor fagfelt som medisinsk bildeforskning og materialvitenskap ønsker man bilder som både er høyoppløste, men som også dekker en stor del av prøven. Å øke både størrelsen og oppløsningen i et konvensjonelt mikroskop krever enten dyrere optiske komponenter, eller store og kompliserte oppsett. I løpet av de siste tiårene har det å bruke datamaskiner til å forbedre bildesystemer, såkalt "computational imaging", blitt stadig mer populært. En ny teknikk innen dette feltet er fourierptychografisk mikroskopi (FPM). I FPM blir en prøve belyst med lys fra ulike vinkler og avbildet i et konvensjonelt lysmikroskop. Resultatbildene blir deretter slått sammen til ett enkelt høyoppløst bilde ved å løse et fasegjenfinningsproblem. Å løse dette fasegjenfinningsproblemet krever imidlertid mye regnekraft, noe som forhindrer FPM-mikroskopet fra å kunne operere i sanntid.

I denne avhandlingen presenterer vi en modifisert iterativ løser for fasegjenfinning som brukt i FPM som tillater samtidig utførelse av deliterasjoner. Den modifiserte løseren avdekker og utnytter oppgaveparalellitet mellom de ulike deliterasjonene ved hjelp av en prioritetskø som bruker en avhengighetsgraf og en kostnadsfunksjon. Dette reduserer kjøretiden, og eksponerer mer dataparalellitet i FPM-rekonstrussjonsprosessen. Implementasjonen vår kjører ca 5 ganger raskere på en CPU med 16 tråder enn en tilsvarende entrådet versjon av algoritmen, selv uten å utnytte dataparalellitet i problemet, og ved kun en svært liten reduksjon i rekonstruksjonskvalitet. Ettersom mange fasegjenfinningsalgoritmer i både FPM og ptychografi har lignende struktur er det gode muligheter for at denne metoden kan anvendes som et paralelliseringsverktøy på et bredere plan. Forslag til videre arbeid er også inkludert.

Acknowledgements

The work in this thesis was performed at the HPC laboratory at the Department of Computer Science (IDI) at the Norwegian University of Technology and Science (NTNU) in collaboration with the COMPMIC group at the Department of Physics (NTNU). I thus would like to thank Prof. Dag and all others associated with the RCN (NFR) FRIPRO project "4D Computed Tomography of Porous Materials" [1].

In particular, I would like to thank my supervisor Professor Anne C. Elster (HPC lab, IDI) for help with the thesis, especially for the help of focusing the work into a coherent whole. I would also like to thank Professor Dag W. Breiby (COMPMIC group, Department of Physics NTNU) for the great help he provided in understanding the physics of the problem, and the rest of the COMPMIC group for interesting discussions and insights into the world of computational microscopy. I would like to thank Professor Muhammad N. Akram (Department of Microsystems, University of South-Eastern Norway) for the commented baseline code he provided me as a starting point for the thesis work, along with advice on how to make it run.

I would like to thank NTNU and the HPC lab for giving me access to the computational infrastructure required for the thesis work.

Finally, i would like to thank my fellow students at the HPC lab, in particular Jacob O. Tørring, Ingvild B. Høgstøyl, Andreas Hammer and Ivar Sandvik for useful scientific discussions and good company in a year of pandemic and hard work. It really has made my time at the HPC lab a memorable experience, and an otherwise dark year a lot brighter.

[1]https://prosjektbanken.forskningsradet.no/project/FORISS/275182

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

Microscopes that create images that cover large sample areas with high resolution[1] are desirable in fields such as medical imaging and materials. A conventional microscope must trade field of view (FoV) for higher resolution. Increasing both the resolution and the sampling area requires either the size of the aperture be increased, the wavelength of the illumination reduced. Increasing the lens size makes the lens more difficult and expensive to manufacture, and decreasing the wavelength of the illumination can in some case damage the sample. It is possible to increase the resolution to the desired level by trading FoV for magnification and then cover the whole sample by mechanically scanning the microscope over the sample and merge the resulting images. This requires precise and expensive equipment however.

A relatively new technique called Fourier Ptychographic Microscopy (FPM)[2] provides an alternative to the scanning method. In FPM, multiple images taken with varying illumination rather than varying translation are used to reconstruct an image with higher resolution over a larger sample area than the optical system used normally would be able to achieve. FPM uses a mathematical model of the imaging system to relate the illumination angle of a given image to the information it provides about the sample. This model then allows for a computational reconstruction process that merges the info from all the images and produces a single result image that covers a large area of the sample at a high resolution. In addition, FPM reconstructs both phase and amplitude of the object, revealing details about the sample not visible in a normal intensity image.

However, FPM reconstruction can be a time consuming process, however, when using a microscope, near real-time or real-time operation is desirable. There are two main approaches for speeding it up. An inherently faster reconstruction process can be used, but the trend in the literature has been towards more advanced recovery schemes that are more computationally expensive, as can be seen in reviews like[3]. The alternative is to try and implement the chosen method in a more efficient manner, for instance by exploiting task- and data parallelism to run as many of the sub-calculations of the iterative process as possible in parallel.

Performing the iterations on a highly data parallel machine like a GPU has been shown to drastically reduce the reconstruction time[4]. However, the inherent parallelism between different iteration steps even in iterative FPM algorithms could potentially also be exploited. As parallelism at the iteration step level can be exploited not only by massively data parallel machines like GPUs, but also by multi-core CPUs, a task parallel implementation can achieve speedups also on machines without co-processors, making FPM more accessible. In addition, unlocking task parallelism between iteration steps would unlock even more data parallelism, as the identified independent tasks can be accelerated by data parallel machines like GPUs as well.

To uncover task parallelism, the structure of the recovery problem, in which an initial guess is fitted to a large collection of constraints in the form of images captured with varying illumination angle, is examined. By mapping out dependencies between the constraints, it is possible to identify constraints that can be fitted in parallel without interfering

1.1 Goals and Contributions

The high level goal of this thesis is to create a method of dynamically identifying independent iteration steps in the recovery algorithm and use it to run them in parallel on either multi-core CPUs or GPUs in a way that both reduces recovery time and keeps reconstruction quality high.

This goal can be split up into the following sub-goals:

- Evaluate to what degree performing iteration steps in parallel can speed up the computational recovery process in FPM while preserving the quality of the reconstruction.
- Implement a Fresnel propagation based FPM recovery algorithm that can run multiple iteration steps in parallel
- Evaluate the performance of this implementation on a multi-core CPU
- Evaluate the effect of relaxing the order of iteration steps and frequency of pupil recovery on both processing time and recovery quality
- Discuss the implemented method in context of related methods such as ptychographic solvers and alternating projections methods, and evaluate its broader applicability

The main contributions from this thesis are:

- A novel method to find and distribute independent iteration steps for iterative FPM schemes.
- A task-parallel software implementation of iterative FPM that integrates this method and runs the independent iteration steps in parallel on a multi-core CPU.

- A general method for identifying and exploiting task parallelism in FPM, ptychography and other related algorithms

Our method for handling and distributing the iteration steps includes a dependency graph and a priority queue structure to keep track of which iteration steps are currently being worked on, which are locked by dependent iteration steps, which are complete and which are available. In addition, an extra heuristic we've named "deviation tolerance" is built into the queue as an adjustable parameter. This parameter lets us control how out of order sub-iterations in the algorithm will be executed with respect to the single threaded baseline, an aspect that is shown to be crucial both for the quality of the reconstruction, and the speed of execution.

Our task-parallel solution then utilizes this priority queue to dynamically distribute independent work to the desired number of threads, while not significantly reducing the quality of the output compared to a single threaded implementation. The preservation of quality is achieved by adding a problem specific cost function to the priority queue, tweaking the priority queue scheduler to work better in this specific case.

We have also worked on a CUDA-based version that allows for GPU parallelisation, but due to time-constraints, this work has not been fully completed and has not been included in the tests. However, data parallel acceleration on GPU has been shown in the master thesis of Anders Treland[4] to be an efficient way to reduce the running time of FPM reconstruction, and fully combining the task parallel techniques from this thesis and data parallel acceleration looks like the natural next step. As the task parallelism and data parallelism in FPM is independent, it is possible to make use of both types of parallelism to further increase performance and utilisation.

1.2 Thesis Outline

The rest of the thesis consists of the following chapters:

- **Chapter 2: Background – Image system modeling** – Description of the image system model used, and how to use it to perform FPM recovery

- **Chapter 3: Background – Parallel Computers** – An introduction to parallel computing, including CPU threading and CUDA

- **Chapter 4: Related work** – An overview of parallelisation and optimisation efforts within Fourier ptychography and the related field of ptychography, as well as other algorithms related on a more genreal level

- **Chapter 5: Implementation** – Details and considerations around the implementation in this thesis. This chapter highlights a number of practical implementation aspects that usually are not described in the literature

- **Chapter 6: Methodology** – How the implementation is tested with multiple parameters and data set to gain insights into to reconstruction quality and speed

- **Chapter 7: Results** – Performance and reconstruction quality

- **Chapter 8: Discussion** – Evaluation of all experiments, and discussion of the broader applicability of the priority queue method in the context of other parallel phase recovery implementation.

- **Chapter 9: Conclusions and future work**
- **Appendix A: Code listings**

- **Appendix B: Extra results**

- **Appendix C: Poster**

Chapter 2

Background - Image system modeling

The correctness of Fourier ptychographic microscopy (FPM) as an imaging technique is ensured through the correct use of a mathematical model of the image system. In the development of this model, some assumptions and simplifications are made to make FPM possible. There are a number of different image system models and multiple formulations of the recovery problem, each suited to different circumstances and with different properties. The choice of model and problem formulation constrains the implementation of the algorithm, and the physical characteristics of the microscope. After choosing a model and problem formulation, it is important to not violate these constraints in the implementation. On the other hand, a chosen model also comes with its own opportunities for paralellisation and efficient implementations. A somewhat thorough walk-through of the image formation model is thus necessary to discuss the implementation of the algorithm, and will be provided in this chapter

2.1 Image System Modelling

To be able to recombine the images produced by a Fourier ptychographic microscope, the original object and the images taken must be related through a mathematical model of image formation. To create a useful and reasonably simple model, some assumptions and simplifications about the light source, optical system and the propagation of light must be made. These assumptions and simplifications gives the model properties that enables Fourier ptychography, but also impose some restrictions on where and when it is valid. The development of this model is sketched below, based on the description found in Introduction to Fourier Optics[5] and in Fourier Ptychographic Imaging[6], as well as the extensions found in the PhD thesis of Pavan Chandra Konda[1]

At its most basic, the imaging model considers the microscope to be a mathematical system that takes some input function and returns some output function. In the case of an imaging system, the input function is the object to be imaged illuminated by a plane wave and the output is the resulting image. The system is assumed to be linear, meaning that any input function can be decomposed into a sum of individual sub-functions where processing these functions individually and recombining them yields the same result as processing the original full image as a unit.

The two types of elementary functions relevant in Fourier ptychography are impulse functions and complex exponentials. When decomposing the input into impulses(through the use of delta functions and their sifting property) the imaging system can be modelled, under certain assumptions, as the following 2D convolution

$$i(x,y) = o(x,y) \otimes h(x,y) \tag{2.1}$$

Here $i(x,y)$ denotes the output image, $o(x,y)$ the input object, $h(x,y)$ the impulse response function of the system, often referred to as the point spread function(PSF), and $\otimes$ the 2D convolution operation(2.2).

$$o(x,y) \otimes h(x,y) = \iint_{-\infty}^{\infty} o(\xi,\eta) * h(x-\xi, y-\eta)\delta\xi\delta\eta \tag{2.2}$$

This model of image assumes that our light source is coherent and that the system is invariant in space and time. Also, it considers the light field to be a field of scalars rather than vectors, ignoring the polarisation of the light. Finally, it assumes that the sample is thin. For thicker, three dimensional samples, different models must be used[7]

2.1.1 Coherent illumination

The Fourier ptychographic imaging model assumes that the light illuminating the sample behaves like a coherent light source. Coherence means that all the phases of the light waves hitting the object vary in unison, and it allows us to model the imaging system as a convolution of the objects amplitude and an amplitude transfer function, as illustrated in equation (2.1)

In practice, any microscope illumination will be only partially coherent. This partial coherence can both be in time and in space. Coherence in time shows how well coherence is preserved at a single point at different times, while coherence in space measures how well two points are correlated at a given point in time.

When a partially coherent light source illuminates an object in an imaging system, it can only be treated as coherent for points that are sufficiently close at the object. The area for which a collection of points can be considered coherently illuminated relative to each other is given by the area of coherence. The width of the area of coherence at the sample for an illuminating LED can be expressed as a function of λ, w and z. Here λ is the wavelength produced by the LED, w the width of the LED and z the distance from the LED to the sample[1].

$$I_c = 1.22 * \frac{\lambda z}{w} \tag{2.3}$$

A different expression for the area of coherence is given in Schnells master thesis[8] as equation (2.4), where A_c is the area of coherence, l is the distance between the observation plane and the source, $\overline{\lambda}$ is the average wavelength radiating from the source and A_s is the area of the emitting source. It is assumed that the source and observation plane are parallel.

$$A_c \approx \frac{l^2 \overline{\lambda}^2}{A_s} \tag{2.4}$$

In their review article[3], Konda et al. provides a similar equation to (2.3), but with the scaling constant 1.22 removed. Chung[9] gives the area of coherence by formula (2.5), where a is the light source radius, λ the wavelength of the emitted light and z the distance from LED to object.

$$L = 0.61 \frac{\lambda z}{a} \tag{2.5}$$

All these formulas are based on the van Cittert-Zernike theorem, and yield similar results.

2.1.2 Space and time invariance

In an idealised optical system the PSF does not vary with space or time. In practice, while the time invariance property will hold, the space invariance will not. Luckily, the PSF will in most cases change quite slowly with space, meaning that if one looks at a limited area of the input field, the point spread function will stay mostly constant[5].

2.1.3 Model of light propagation

The image system model given by equation (2.2) requires some impulse response function $h(x, y)$ or optical transfer function $H(k_x, k_y) = F\{h(x, y)\}$to be complete. Light propagating in free space between two perpendicular planes separated by a distance z can be modeled in multiple ways. The three main methods applied in Fourier ptychography is the angular spectrum method, Fresnel propagation and Fraunhofer propagation[8]. The methods differ in what approximations are made to derive them, and under which circumstances they are correct. Generally, a more approximate solution is faster to compute. The method used should thus be chosen based on which of them is the cheapest that still yields accurate results for a given working distance. All the models assume a monochromatic source, meaning the illumination only contains one wavelength.

The angular spectrum method can be derived from the Rayleigh-Sommerfeld equation without approximation and is thus a precise model of light propagation in a scalar model. The fact that this method works equally well under all working distances z, means that it can be used in algorithms and models were lights is propagated over very short distances, for instance in the propagation between object slices in 3D Fourier ptychography[7]. It can be formulated as follows:

$$O_2(x,y) = F^{-1}\{F\{O_1(x,y)\} * H_{AS}(f_x,f_y)\} \tag{2.6}$$

Here O_2 is the plane propagated onto, O_1 the source plane $F\{\}$ the 2D Fourier transform, $*$ pointwise multiplication and H_{AS} the optical transfer function of free space by the angular spectrum method.

$$H_{AS}(f_x,f_y) = exp(ikz\sqrt{1-(\lambda f_x)^2-(\lambda f_x)^2}) \tag{2.7}$$

In the above expression, i is the imaginary unit, z distance between the planes and k the wavenumber (spatial frequency) of the light.

When the distance between the planes z is large enough to satisfy the inequality in (2.8), it is possible to use Fresnel propagation to model the free space propagation without significant loss in precision. To do this, the optical transfer function is set to be the function shown in equation (2.9). The same propagation can also be written using equation (2.10). Here, the constant phase factor of $\frac{e^{ikz}}{i\lambda z}$ is ignored. ξ and η are the coordinates in the source plane, while x and y are the coordinates in the target plane.

$$z^3 \gg \frac{\pi}{4\lambda}[(x-\xi)^2+(y-\eta)^2]_{max} \tag{2.8}$$

$$H_{Fres}(f_x,f_y) = exp(ikz) * exp(i\pi\lambda z(f_x^2+f_y^2)) \tag{2.9}$$

$$O_2(x,y) = exp(i\frac{k}{2z}(x^2+y^2)) * F\{O_1(\xi,\eta) * exp(i\frac{k}{2z}(\xi^2+\eta^2))\} \tag{2.10}$$

For even larger z, where the criterion in (2.11) is satisfied, computation time can be reduced further by using Fraunhofer propagation without significantly reducing precision. The Fraunhofer propagated input plane is proportional to its own Fourier transform. In implementations, the proportionality constant is often ignored[6]. This turns the propagation into a simple Fourier transform.

$$z \gg \frac{k}{2}[\xi^2+\eta^2]_{max} \tag{2.11}$$

Propagation distances where Fraunhofer propagation is a correct model are said to be **far field**, while propagation distances where Fresnel propagation is correct are called **near field**

2.1.4 Resolution limit

The resolution limit of an optical system in absence of any image system aberrations can be defined by the Rayleigh resolution limit given by formula (2.12)[8]

$$\delta r = \frac{0.61\lambda}{NA_{obj}} \tag{2.12}$$

NA_{obj} is the numerical aperture (NA) of the imaging system, λ the wavelength of the illuminating light and δr is the minimum distance between two features in the object that can be distinguished between in the output image. If the two features are any closer, they become indistinguishable in the output image. In other words, δr defines the resolution of the image.

The resolution is thus determined by illumination and NA. NA is defined to be the refractive index of the medium n multiplied by the sine of angle between the point where the optical axis and object intersect, and the edge of the objective as shown in Figure (2.1). As $n \approx 1$ in air, the formula simplifies to $NA = sin(\theta)$ when the sample is imaged in air.

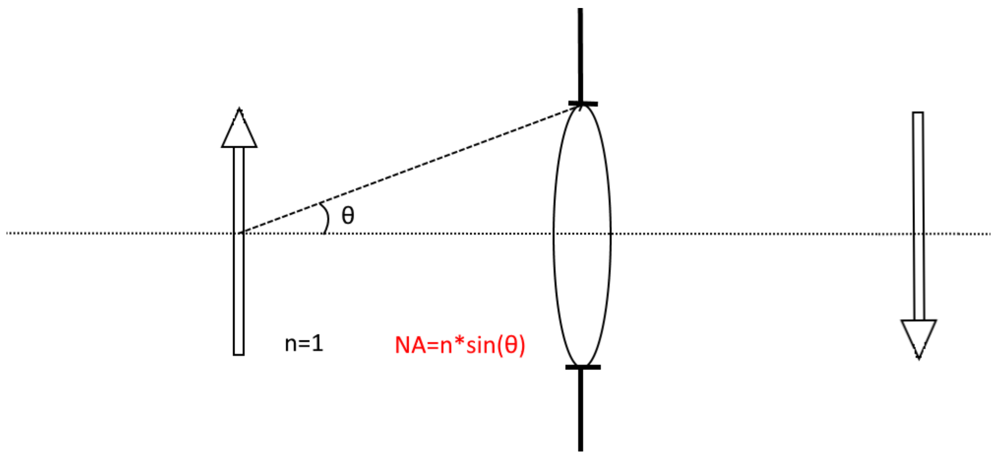


Figure 2.1: The numerical aperture of a microscope is defined to be refractive index n multiplied with the angle between the object at the optical axis and the edge of the lens

2.1.5 Space-Bandwidth Product

In general, increasing the NA of an objective lens yields an increase in the resolution, but decreases the field of view (FoV)[10]. The total amount of information transmitted through the imaging system remains constant. A quantitative measure of this information throughput is the space-bandwidth product (SPB), which can be expressed as the product of image size and bandwidth[11]. In the case of a 2D image, with dimensions X and Y in image space and F_x and F_y in frequency space, the SBP can be written using the following equation.

$$SBP = XYF_xF_y \tag{2.13}$$

2.2 Fourier Ptychography

Fourier ptychography is a super-resolution technique that uses multiple images of an object taken with illumination from different angles to construct an image with a higher space-bandwidth product than the optical system would normally permit. To construct the high-resolution image, the information from the input images must be merged to produce both the phase and the amplitude of the complex object. However, image sensors can only record the intensity of incoming light. To reconstruct the phase of the imaged object, a phase recovery method must be used. Different formulations of this problem along with different solvers exist in the literature[12, 13].

2.2.1 The Optimisation Problem

All FPM algorithms takes as inputs a number of images representing different illumination angles and an initial guess. The goal is to use the input images to reconstruct a high resolution image of the object. To define a method, two aspects must be determined: how to measure the error of the current estimate with respect to the imaging model, and how to numerically reduce this error. It has been shown that minimising the error of the image estimate amplitude is a robust approach in FPM[12]

The amplitude error is defined to be the difference between the sum of squares of the measured value and the value predicted by the current object estimate when is applied to the forward image model along with the correct Fourier shifts(2.19). Mathematically this can be formulated as the following equation[12].

$$E_a(O(\mathbf{u})) = \sum_l \sum_{\mathbf{r}} |\sqrt{I_l(\mathbf{r})} - |F^{-1}\{P(\mathbf{u})O(\mathbf{u}-\mathbf{u}_l)\}||^2 \qquad (2.14)$$

Here O is the Fourier transform of the high resolution object transmittance function, P the pupil function and I_l the measured intensity from the current low-resolution image indexed by l. r is a coordinate in image space. A small difference between the measurements and the model yields a small sum in(2.14). Minimising this equation will thus lead to an optimal reconstruction of the high-resolution image.

The approach originally used in the context of Fourier ptychographic microscopy is taken from the related field of ptychography and is based on the method of alternating projections[2]. In this technique, the working estimate is transferred between the Fourier domain and the image domain in an iterative fashion. Each time the domain is changed, a constraint is applied to project the current estimate onto the space of legal solutions in the current domain. an illustration of the general procedure is given in Figure (2.2)

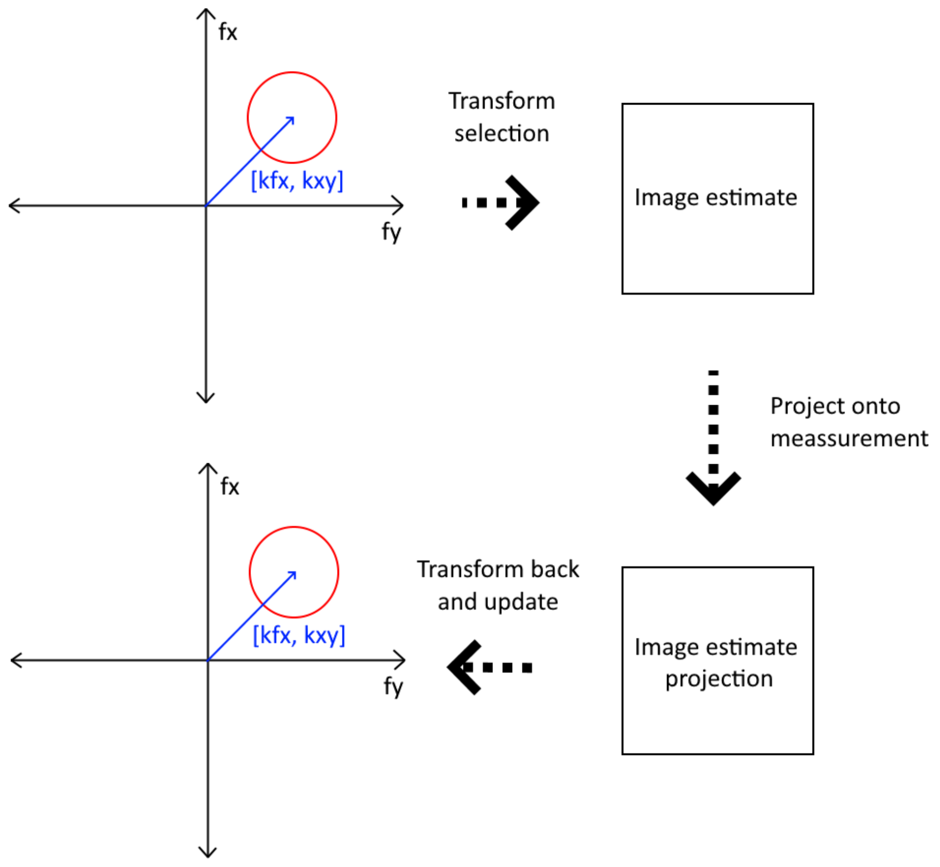

fx
[kfx, kxy]
fy
Transform selection
Image estimate
Project onto meassurement
fx
[kfx, kxy]
fy
Transform back and update
Image estimate projection

Figure 2.2: The general procedure of an the alternating projections algorithm

The Gerchberg-Saxton Algorithm[14] is an example of an alternating projections algorithm. Here the projection step in both domains consist of enforcing known constraints. When adapted for Fourier Ptychography by Zheng in 2013[2], the update formula in image space was given by enforcing the measured intensity while keeping the estimated phase (2.15) and the update in Fourier space was performed by reverse transforming the new image estimate, and overwriting the old estimate in Fourier space.

$$I_{upd} = |I_{measured}| \frac{I_{est}}{|I_{est}|} \tag{2.15}$$

A problem with this approach is that the method of alternating projections only guarantees an optimum if the sets involved are convex. This property does not hold in the FPM amplitude optimisation problem. As a consequence, the method of alternating projections can only guarantee convergence towards a local optimum[15].

A more robust but also more computationally expensive method is the Gauss-Newton method. This method replaces the update step in Fourier space, and has stronger convergence guarantees than Gerchberg-Saxton[12]. It also permits simultaneous updates of the object and pupil function. Konda[1] implements this update step as the following equation (2.16). Here the second derivative (the Hessian) is not calculated exactly, but instead estimated. This is therefore an example of a Quasi-Newton method.

$$O_{upd}(k-k_i) = O(k-k_i) + \alpha \frac{|P|}{|P|_{max}} \frac{\tilde{P}(O_{i\_upd}(k-k_i) - O_i(k-k_i))}{|P|^2 + \delta} \tag{2.16}$$

2.2.2 Free Space Propagation Model

During the iterative recovery process used in Fourier ptychography, the current estimate is transferred between different planes. In the Fraunhofer propagation model, the light field is transferred between the object plane, the Fourier plane, and the detector plane. In Fresnel propagation, the light field is transferred between the object plane, the lens plane, and the detector plane.

In the Fraunhofer propagation based model, the lens is assumed to be positioned in the Fourier plane. The Fourier plane contains the far-field diffraction pattern of the object, which is the Fourier transform of the object scaled by some proportionality constant. The aperture of the lens functions as a low pass filter, only letting the central low frequency parts of the object Fourier transform through. The radius of this aperture in the Fourier plane is given by the product of the numerical aperture of the lens, and the wavenumber (spatial frequency) of the illumination in radians[6], as show in equation (2.17). The refraction through the lens and subsequent propagation to the detective plane is then modelled as an inverse Fourier transform of the low pass filtered diffraction pattern. An illustration of this process is provided in 2.3

$$radius_{aperture} = NA * \frac{2\pi}{\lambda} \tag{2.17}$$

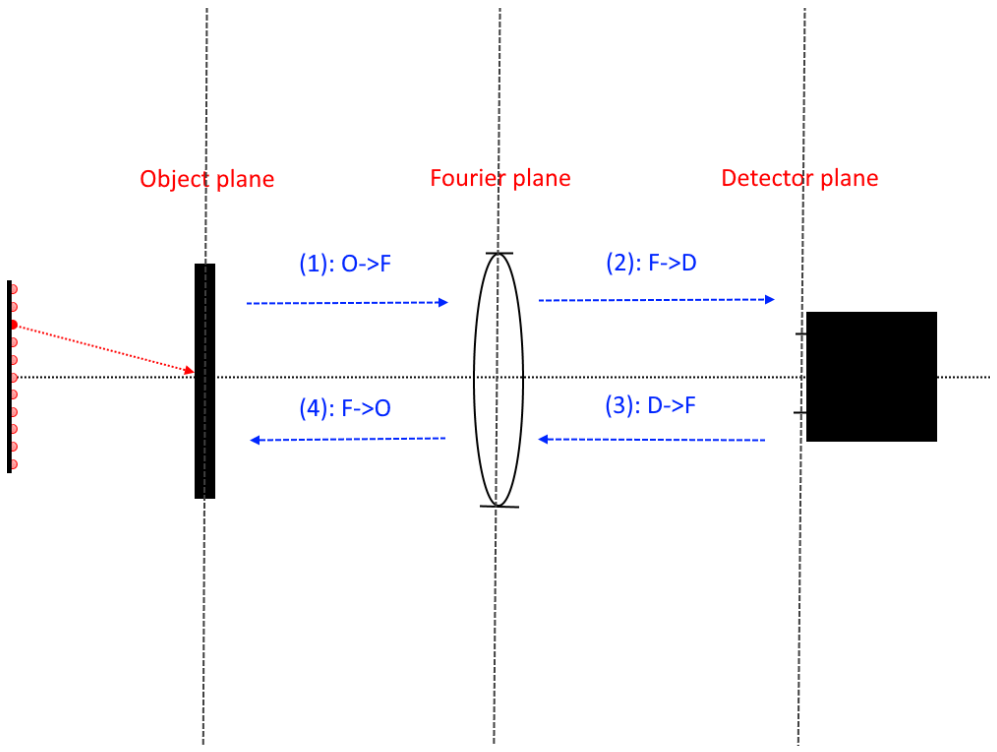


Figure 2.3: The three planes of the imaging system in a Fraunhofer model, and the propagations between them

In this Figure, propagation (1) and (3) are Fourier transforms, and propagation (2) and (4) inverse Fourier transforms. Following Zheng[2], all proportionality constants are ignored.

When using Fresnel propagation, the model changes in two main ways. First, the central plane is renamed to **the lens plane**, as it now represents the physical plane at the lens. The second is that two complex multiplications are added to the propagatation step. The first one before the Fourier transform, the second after. These complex multiplications account for changing phase across the wavefront, making it curved. This gives the model better accuracy when propagating over short distances.

Under Fresnel propagation, the pixel size changes between planes. The size change is given by equation (2.18), where λ is the wavelength of the light, z the distance between the planes and W the width of the source plane[1]. In an actual implementation however, most of the extra complex factors cancel each other, and the resulting method only differs in the calculation of pupil shift and the propagation between object and lens plane, as Konda shows[1].

$$psize_{new} = \frac{\lambda z}{W} \tag{2.18}$$

The angular spectrum method is used in 3D FPM, to propagate the light field between the slices of the 3D sample, as the distances between these are too short for Fresnel propagation. However, in 2D FPM, the distances between the planes are long enough that Fresnel propagation provides precise results at a lower computational cost. Also, since the pixel size is constant between planes when propagating with the angular spectrum method, the fact that the lens plane in FPM is much larger than the object plane becomes a problem. To sample the full aperture of the lens plane, an equally large area of the object plane must be samples, further increasing computational costs. With Fresnel propagation however, the pixel size changes with the planes in a way that ensures that if the entire object is sampled with pixels at the object plane, the entire propagated object will be sampled at the lens plane, as discussed in Kondas PhD thesis[1]

Angled Illumination

The intensity image formation process for a given image can be modelled using Fraunhofer propagation as the following forward model[1].

$$I(x,y) = |F^{-1}\{P(k_x,k_y) * O(k_x,k_y)\}|^2 \tag{2.19}$$

Here, $I(x,y)$ denotes the image, F^{-1} the inverse Fourier transform operator, $O(k_x,k_y)$ the object Fourier transform, $P(k_x,k_y)$ the pupil function/optical transfer function of the imaging system and $|.|$ the pointwise modulus (absolute value) of the complex image.

The pupil function acts as a low pass filter. This means that only a portion of the frequency spectrum of the object can pass through the optical system. Changing the angle of illumination shifts the spectrum, changing what part of it is let through. Mathematically this shift is performed by multiplying the object field with a complex exponential representing the incoming plane wave. The shift theorem(2.20), shows how this change in illumination angle leads to a shift in the Fourier spectrum of the object. For a thin lens, and with light propagation modeled according to Fraunhofer propagation, the equation (2.20) holds for all angles.

$$F\{o(x,y) * e^{iax+iby}\} = O(k_x - a, k_y - b) \tag{2.20}$$

However, if the light field is propagated between planes according to Fresnel propagation, a different equation relate illumination angle and pupil shift more precisely. Konda[1] formulates it as (2.21).

$$S_{aperture} = \frac{P_{FoV} - L_{shift}}{L_{dist}} u + P_{FoV} \tag{2.21}$$

Here $S_{aperture}$ is the shift of the pupil at the lens plane, P_{FoV} the image segment offset, L_{shift} the offset of the currently lit LED from the central LED, L_{dist} the distance between the LED matrix and the object plane, and u the distance between the object and lens plane. In the above formula, the unit used is meters. To translate into pixels, $S_{aperture}$ must be divided by the pixel size in the real space lens plane. This pixel size is then valid in both the real space lens plane, and the frequency space at the lens plane.

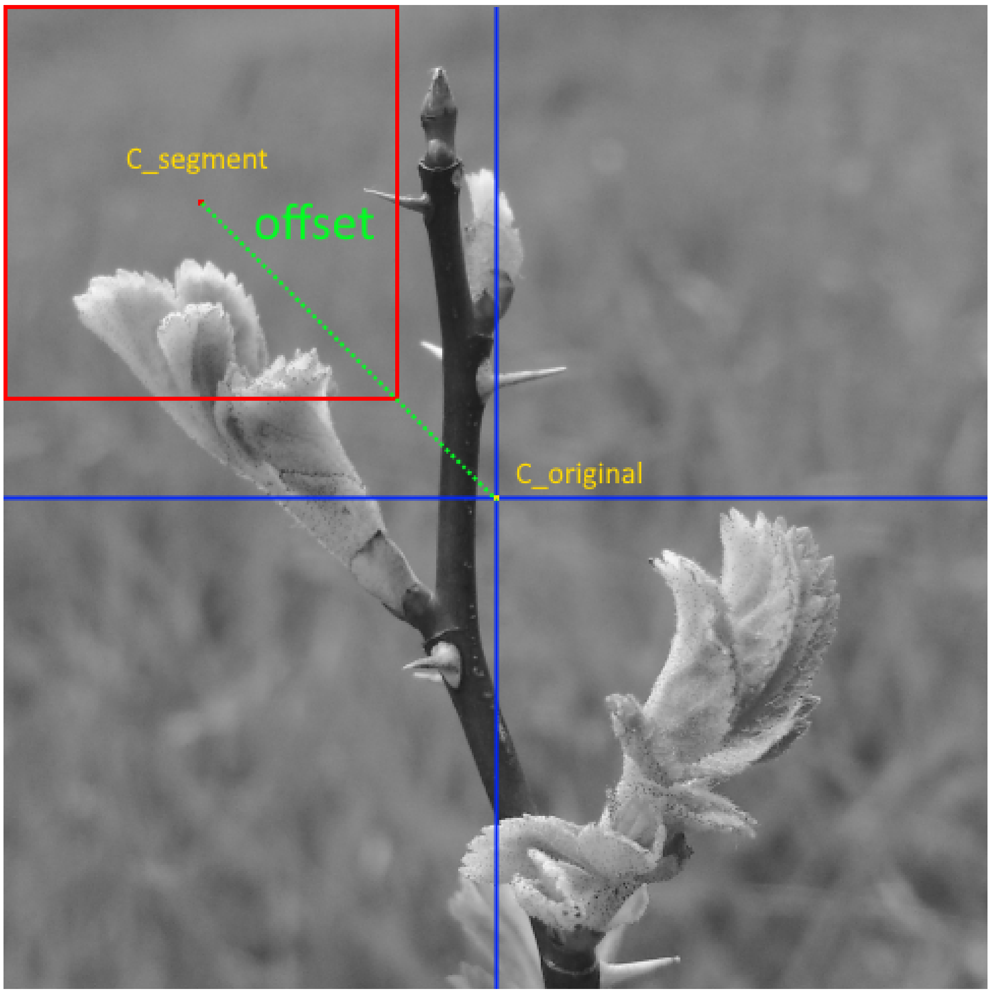


Figure 2.4: An image with a selected segment, and the offset between segment center and image center

Pupil Recovery

The pupil function of the imaging system is also recoverable with a Gauss-Newton approach. The formula for this is shown in equation (2.22). The ideal pupil is represented by a function that is zero outside of the lens aperture and one within, and is used as an initial guess.

$$P_{upd} = P + \alpha \frac{|O|}{|O|_{max}} \frac{\tilde{O}(O_{i\_upd}(k-k_i) - O_i(k-k_i))}{|O|^2 + \delta} \tag{2.22}$$

2.2.3 Noise Reduction

To improve the reconstruction quality, several implementations of FPM perform some extra pre-processing steps on the input images. It is common to perform a threshold-based noise reduction step on the input images[16]. Several advanced noise-reduction schemes have been proposed that try to minimise the amount of data lost in the denoising process[17, 18].

2.2.4 Reconstruction Process

The reconstruction process iterates over all images, applying equation (2.15) at the detector plane and (2.16) and equation (2.22) at the lens plane. Fresnel propagation is used to propagate the fields. The pupil shifts ensures that only the section of the lens plane corresponding to the pupil size and illumination angle is updated. It has been shown that the order of iteration should be close to decreasing image intensity for the best convergence[1]. Once information from all illumination angles have been incorporated into the lens plane estimate, the full iteration process is repeated until acceptable convergence.

Chapter 3

Background - Parallel computers

The second aspect important to the implementation of the recovery algorithm is the underlying implementation platform. In this work both multi core CPUs and GPUs will be examined. This chapter is a brief discussion the properties and principles of programming on these devices, and the systems incorporating them

3.1 History of parallel computers

In the first several decades of microprocessor development, the number of transistors on single integrated circuit increased exponentially, doubling every second year. This trend was predicted by Moore in 1965[19], and has been dubbed "Moores law". Borkar[20] identifies three main driving factors for this development. The first is the fact that when transistor size was reduced, such that more transistors can fit on the same area, the supply voltage could be scaled down enough that the power required to drive the chip remained constant. This is called Dennard scaling. The second is the development of more advanced microarcitectures, and the third is the use of caches to reduce the delay of memory access. However, in the 90s, as transistors got smaller and smaller, Dennard scaling stopped applying[21]. This meant that an exponential increase in transistors would from there on would lead to an exponential increase in power consumption.

The development of traditional single-core processors thus hit a power wall. Increasing performance through smaller transistors and higher clock frequencies like before quickly lead to unacceptable levels of power consumption. It was still possible to increase the number of transistors, but not to run them all at full frequency at the same time. A lot of the chip at any given time would be dark silicon[22]. This development lead to two major shifts in hardware development: from single-core to multi-core, and from homogeneous general processing units into heterogeneous, more specialised units.

This shift was dramatic not only for hardware manufacturers, but also software developers. To fully utilise a heterogeneous and parallel system, algorithms and code must be written in a different way than for a single-core machine[23]. Parallelism inherent in computational problems must be identified and exploited by multiple, often heterogeneous compute units. The reward, however, is increased performance at a lower time and energy budget.

3.2 Terminology

In this thesis, the terms **processing unit**, **program**, and **process** is defined as follows: A **processing unit** is a computer that can execute instructions. A collection of instructions is called a **program**. When a program is scheduled by the operating system to run, it is said to be **executing**. A program in execution is called a **process**. A process can be **ready** for execution, **running** on a processing unit, or **waiting** for resources. The terms thread and process are used interchangeably unless otherwise noted.

3.3 Concurrency and parallelism

A concurrent computer system is a system where multiple processes can be executing simultaneously. Even a single processor can be used for concurrent execution by sharing processor time between all the current processes. However, since only one process is actually being run on a processing unit at the processor at any given point in time, there is no actual parallelism. For there to be parallelism, there must be multiple physical processing units running different programs at the same point in time.

Concurrency provides performance benefits when processes have to wait for external resources such as different computing units or memory. As long as there is another process ready to be run anytime the current process has to wait, the shared processing unit can switch to it and avoid a situation where the processor sits idle. This increases utilisation and throughput. However, if the executing processes do not spend a lot of time waiting for resources, concurrency by itself does not increase throughput by much.

The performance increase offered by concurrency is limited by the amount of waiting time each process needs. By adding more processing units to the computer system, it is possible to increase the throughput without having to wait for the currently running process(es). This, however, only works if there are processes ready to run. If all non-running processes are waiting for external resources, adding extra processing units will not help. Additional processors increase performance if the lack of processing power is the bottleneck at a given moment, but not necessarily otherwise.

3.4 Task parallelism

Task parallelism is possible for problems where multiple streams of instructions (threads) can be executed independently. A task parallel workload maps well to a collection of independent processing units, as each unit can receive its own task to be performed. Sometimes the tasks require coordination. In this case some sort of communication mechanism must exist to allow the threads to synchronise. A web server is an example of a task parallel system, as the connections to the clients are independent and can be processed independently.

3.5 Data parallelism

Data parallel problems are problems where the same operation is performed across a large number of independent data items. These sorts of computations are efficiently mapped to sets of processing units that can perform the same operation on multiple data items at once. As all of these processors are executing the same operation, they can share control logic. This simplifies the design and makes them more energy efficient. Vector addition is an example of a data parallel operation, as it consists of a large number of identical and independent operations across different data.

3.6 Multiprocessor memory layout

Modern multiprocessor systems can be split into two types: shared memory systems and distributed memory systems. In a shared memory system, the different processing units work on the same memory, while in a distributed memory system each processing unit has its own private memory. A modern multiprocessor CPU by itself is an example of a shared memory system, with multiple cores having equal access to the same main memory. A heterogeneous computing platform consisting of a CPU and a GPU on the other hand is an example of a distributed memory system, with both the GPU and CPU having its own private memory.

Whether a system uses shared memory or distributed memory has consequences for how different processes and threads in the system cooperate. Communication between processes on distributed processing units must happen over some sort of network or connecting bus. This introduces topology dependent latencies and memory bandwidth limitations when transferring data between the local memories. A goal in in distributed processing is therefore to minimise memory movement between the different processing units.

3.7 Concurrent Programming

To program concurrent programs, the underlying system must provide access to programming abstractions such as threads, locks and other synchronisation primitives, and a way to start, stop and wait for threads. On more specialised machines like GPUs, domain specific programming abstractions like thread blocks and grids are used. In this work, threading on the CPU is programmed with the Linux thread library pthreads, while concurrent programming on the GPU is programmed with CUDA.

3.7.1 Pthreads

Pthreads is an interface for thread programming available on the Linux operating system. The interface exposes functions for creating, destroying and waiting for threads. In addition, it contains synchronisation primitives such as as mutexes, semaphores and condition variables[24].

3.7.2 Strong and weak scaling

This sub-section is in large part from the pre-project.

When given a program where some fraction p of the execution time can be arbitrarily parallelised, while the other fraction $1-p$ must be run serially, the amount of possible speedup S for a given amount of input data run on N processors is given by equation 3.1

$$S = \frac{1}{(1-p)+\frac{p}{N}} \tag{3.1}$$

This equation is known as Amdahl's law, and was presented in[25]. By letting the number of processors N approach infinity, it is clear that the given speedup of a problem with some fixed input size is proportional $\frac{1}{1-p}$, that is, the serial fraction. If for instance 25% of a program is strictly serial, then the maximum attainable speedup is $\frac{1}{1-0.75} = \frac{1}{0.25} = 4$.

However, this analysis assumes that the size of the input data remains constant. Gustafson claims in [26] that this measure is misleading. Rather than assume constant data size, Gustafson argues that it is more realistic to assume constant run time. An outline of the argument for this is can be seen by looking at an example. Let R_i be some run time on a computer with N_i cores for a data set of size D_i. If the number of cores are doubled, the running time of the problem might not be reduced by much for the same set D_i of data. However, many problems in scientific computing have the property that if you double *both* D_i and N_i, then the total running time *stays constant*. This means that problem sizes can be effectively scaled with the number of processors, even though the run time of a fixed problem might not be. Gustafson formulates this result in equation 3.2, where s denotes the serial portion $1-p$.

$$S_{scaled} = s + pN \tag{3.2}$$

If the running time for a program with input of constant size decreases linearly with the number of processors, as modelled by Amdahl's law, it is said to exhibit **strong scaling**. If instead the running time can remain constant while the number of processors and input size increases, as Gustafson describes, it exhibits **weak scaling**.

3.7.3 Load balancing

When distributing computation across different processing units, it is important for performance to distribute the work such that all involved processing units need an equal amount of time to complete their share of work. If for instance four equal CPU cores are assigned some amount of work, but the share given to the first core is much larger than the share given to the others, the first core must work for much longer than three three others, which end up finishing their work and then wait for the first. If instead all the CPU cores are given the same amount of work, no core has to wait for the others, and the total execution time is reduced, as shown in Figure3.1.

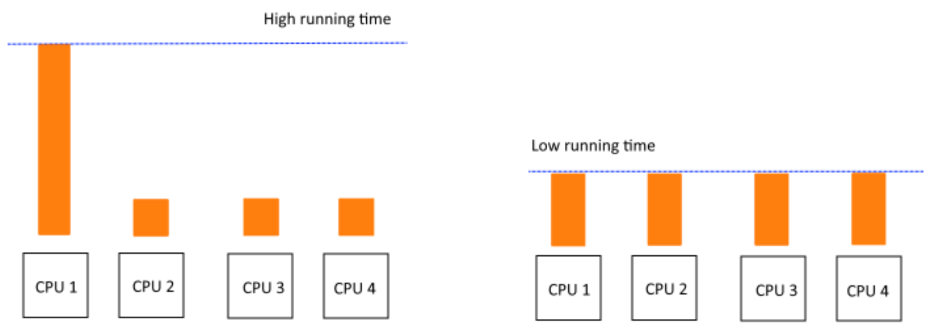


Figure 3.1: The execution time of an unevenly vs an evenly distributed workload

SIMD-processors

From pre-project

A data parallel algorithm often contains cases of the same operation being performed on multiple data elements. This makes it easy to map data parallel workloads onto processors that support performing the same operation on multiple data elements. These Single Instruction, Multiple Data (SIMD) processors have operations that takes vectors of data, and perform element wise operations on them. The SIMD processor accomplishes this by using collections of single data processors, where the input vector elements each get assigned as input to a single processor. A single processor in this vector configuration is referred to as a SIMD processor lane. All the processors in a SIMD core perform the same instruction at a given cycle. As only a single instruction is needed for multiple data elements, this approach saves power and bandwidth compared to a processor that fetches an instruction for every data element[23]. A common extension of this architecture is to allow a given single data processor to opt out of the instruction currently being performed on the SIMD processor.

3.8 General programming on GPUs

This section is large part from the pre-project

As the performance of single core computers have stagnated, further gains in compute power must come from parallel and specialised hardware[27][23]. One of the new types of hardware that have emerged over the last decade as an attractive computation platform is the GPU. GPUs use a large number of individually weak processors to create a cost- and power efficient computation unit capable of high throughput[28]. Overall, the microarchitecture of a GPU differs significantly from the microarchitecture found in CPUs. This means that programmers must approach GPUs in a different way than CPUs, and be mindful of its strengths and weaknesses to use it effectively. The terminology used to describe the concepts and microarchitectural components of a GPU varies between vendors. In this section, vendor independent descriptive names are used (inspired by the approach in[23]), unless otherwise noted.

3.8.1 Overall Architecture of a GPU

A single core in a GPU runs at a slower frequency than a core in a CPU, but since they are simpler and less power hungry it is possible to combine many more on the same chip. The cores, are organised into multiprocessors that behaves like SIMD-processors, meaning that all cores in a multiprocessor must perform the same operation. As is common, the GPU SIMD processor also allows cores in the multi-processor to opt out of a SIMD-instruction. What makes the GPU model different is that considers a sequence of single data instructions performed on a single lane as its own logical thread. This abstraction turns the GPU into a processor more similar to the general data parallel processor described in[29], especially since each of these virtual threads each get their own private register memory. In modern GPUs, the SIMD processors are often multi-threaded, allowing several SIMD threads to execute in parallel. To organise the SIMD threads, groups of SIMD threads are assigned to a single SIMD processor.

3.8.2 The GPU memory model

GPUs have a different memory organisation than CPUs[30]. Where CPUs handle the gap in processor and memory latency primarily through transparent, hardware controlled caches, GPUs have several levels of programmer controllable cache to fill this purpose. GPUs usually function as co-processors, meaning that their operation has to be orchestrated from the CPU. Any memory a program wants to modify on the GPU must be especially copied there over to the GPU via the bus (usually PCIe) connecting the CPU, the GPU and the main memory, and the operation must be initiated by the CPU. This operation has high latency, and demands significant bandwidth from the bus. To write an efficient GPU program, the number of times this operation is performed should be minimised, as the overhead incurred severely reduces the overall gain of using the GPU. The ideal situation is therefore to be able to fit the entire working set of an algorithm on the GPU. The following sections discuss the memory layout in NVIDIA GPUs, but the overall organisation is similar to AMD GPUs, as can be seen in for instance[31].

In addition to the programmer controllable caches, a GPU can *hide* latency by having enough SIMD threads scheduled that whenever one SIMD thread has to wait for memory, another SIMD thread that is ready to run can be swapped in.

Global Memory

The largest memory on the GPU is called global memory in the NVIDIA terminology. This memory comes in the form of DRAM soldered on the GPU itself, and functions as the main memory of the GPU. It is large but, compared to the other memories on the GPU, also quite slow as it is placed of chip. It is therefore usually cached[23]. When data is initially moved to the GPU, it is put in this memory. To minimise the latency of fetching data from the global memory, the GPU tries to *coalesce* the memory requests, by combining several requests to adjacent memory. To be able to do this, the memory requests must satisfy certain requirements in memory alignment, type size and relative location in memory. The specifics of these requirements vary between GPUs.

The reason coalescing is effective is the way DRAM is organised. When a memory request is made to a bank of DRAM, the address specifies both a row and a column. The row is a contiguous block of memory, and the column specifies what part of this memory block should be fetched. As soon as a row is requested, it is put in a buffer, meaning that successive accesses to this row can query the buffer rather than the full bank, decreasing latency. To make the most of this fact, modern DRAM supports burst mode, where a memory request to the row can return as much of the row buffer as the bus can handle every cycle. This makes larger, contiguous accesses to DRAM much more efficient than spread-out accesses[23][p. 87], and motivates memory coalescing as an optimisation technique.

The fact that global memory is cached means that the usual cache aware optimisations, such as blocking for matrix multiplication, still apply for accesses to global memory.

Local memory

Called shared memory in NVIDIA terminology, this memory is local to each SIMD processor. As such, it is available to all the threads in a SIMD thread block that is assigned to the SIMD processor. Each thread block is allotted its own share of local memory however, so different thread blocks running on the same SIMD processor do not share memory. Local memory is placed on-chip, and has low latency and bandwidth. However, it is rather small. Since it it shared between all SIMD threads in a thread block, it can be used for synchronisation between threads and as a common work memory.

Register Memory and Private Memory

Private to each logical thread is a set of registers. Similar to the way local memory is shared between all the SIMD threads assigned to a SIMD processor, the register memory is shared between all logical threads across all SIMD threads assigned to a SIMD processor. This means that the more logical threads a programmer assigns to a SIMD processor, the less register memory each logical thread will receive. The register memory is fast and on-chip. If a logical thread uses more memory than it is assigned, it is allotted space in a section of off-chip memory called private memory. As private memory is off-chip, it has higher access latency. This means that keeping logical thread memory within the register memory limit is an important optimisation goal. As the register memory is shared across all logical threads in a thread block, reducing the number of logical threads per block to increase the amount of memory each is given can a necessary optimisation.

3.8.3 CUDA

Concurrent programs are, in general, much harder to program than sequential programs, necessitating effective programming models and abstractions[32] To make programming their GPUs reasonable, NVIDIAs provides the GPU programming platform CUDA. CUDA is available to programmers as a software abstraction layer that gives access to the GPU as a general processing unit. Any general programming on an NVIDIA GPU must use this platform. CUDA provides most of the abstractions discussed in the previous sections, but by different names. A quick naming reference is given in Table 3.1

Table 3.1: CUDA terms

Descriptive term	**CUDA term**
Multithreaded SIMD processor	Streaming Multiprocessor(SM)
SIMD thread	Warp
Logical thread	CUDA thread
GPU memory	Global memory
SIMD processor local memory	Shared memory
Logical thread private memory	Local memory
Logial thread register memory	Thread processor register memory

Threads and Warps

CUDA calls their SIMD-threads warps, and each warp consists of some number of logical GPU threads, called CUDA threads. The number of CUDA threads in a warp is commonly 32, but can vary between architectures. The CUDA threads within a given warp can opt out of the current instruction, essentially allowing each CUDA thread to behave as an independent thread. To be able to differentiate between threads during code execution, each CUDA thread is assigned its own ID. The GPU uses the thread IDs to group the threads into warps in a predictable way, where each warp consists of consecutive thread IDs. In Volta and later architectures, each CUDA thread has its own program counter(PC), and can be run independent of what warp it belongs to. This is different from earlier architectures, where all threads in a warp were guaranteed to be run on a SM simultaneously. To guarantee synchronous warp execution, the barrier `__synchwarp()` must be used

Thread blocks and the grid

A thread block is a collection of CUDA threads specified by the programmer. The CUDA platform guarantees that all the CUDA threads in a thread block will be executed on the same Streaming Multiprocessor. This allows the CUDA threads of a single thread block to use the shared memory for communication and coordination. The grid is the structure organising the thread blocks. Within a grid, each thread block has its own unique ID, in one, two or three dimensions.

Chapter 4

Related Work

Since the introduction of the Fourier ptychographic technique in 2013, much effort has been made to improve image acquisition rates and quality through various experimental setups. In addition, several formulations of the phase recovery optimization problem have been proposed, leading to different iterative recovery methods. In the related field of ptychography, several parallel and GPU-based algorithms and frameworks have been proposed

4.1 Denoising

During the image acquisition process in FPM, the images pick up noise from various sources. The background noise of a given input image can be estimated by taking an average of the image intensity and check if it is below a pre-defined threshold. If it is below the threshold, this average is assumed to be background illumination and subtracted from the image[16]. This method however is not able to distinguish signal from noise, meaning that details from the imaged object are removed together with the noise when the background is subtracted[17].

To improve the denoising process such that it removes as much noise and as little signal as possible, an adaptive denoising scheme is proposed in Fan[17]. This scheme works by adding a noise factor to the projection step in the image plane, and estimates noise pixel by pixel for each iteration step. Pixels that are assumed to be noise can be then be ignored in the projection.

Another recent approach is a denoising step that, like the threshold denoising, happens as a pre-processing step on the input images. It does, however, not use an uniform average estimate as a threshold on the input images. Instead, the image is decomposed into different details by wavelet decomposition. Then a threshold is applied to each of the detail images, before the detail images are reverse transformed back to an intensity image[18].

4.2 Recovery algorithms

In addition to the iterative Gerchberg-Saxton-Fienup and iterative Quasi Newton methods for phase recovery, other methods are also used in the literature. Yeh[12] reviews a number of phase recovery algorithms both iterative and single-step global based on multiple formulations of the FPM optimisation problem. They conclude that the iterative Quasi-Newton method in general is the most robust and practical.

4.2.1 Parallel Ptychography

Within the field of Ptychography, where a recovery problem similar to FPM must be solved, certain parallel implementations make use of different recovery algorithms more suited to parallelisation. The Difference Map(DM) algorithm is parallel and has been successfully implemented for ptychography[33]. Dong[34] presents a parallel implementation at the sub-iteration level using the difference map algorithm where the sub-iterations are divided into contiguous groups that are each assigned its own GPU. However, a disadvantage of this method is the increased memory footprint[35], and the object update itself is not parallelised.

The extended Ptychographic Engine(ePIE) is an iterative method for solving the phase recovery problem in Ptychography. However, by dividing the workload into disjoint image areas, Nashed et.al[35] have created a parallel implementation where each disjoint image area is given its own GPU for processing. They also present techniques to combine the disjoint image areas in a way that removes artifacts. Nashed also presents a more communication heavy parallel version, where neighbouring image areas exchange edge information to reduce the edge artifacts during computation. Both these approaches might be used in FPM, but instead of partitioning the image, the Fourier spectrum of the object is partitioned. A similar approach called SHARP also exists[36], with different methods to handle to overlapping regions.

4.2.2 Parallel Fourier Ptychography

Due to the partial coherence of the illumination in FPM, an image of a large sample has to be processed in sub-images. These sub-images can be processed independently. Within the processing of these sub-images, it is possible to leverage the inherent data parallelism to reduce running times by using for instance GPUs[4]. Sub-iteration task parallelism like the schemes shown in[35] seem much more rare. Some projects using parallel image acquisition with multiple cameras combine it with parallel writing and partitioning into sub-images[37]. Xiu[38] presents a parallel algorithm where the different sub iterations are grouped into overlapping partitions that are processed independently. The claim is that the individual images produced by these can be more easily be combined than a traditional object plane partitioning, where edge artifacts between the segmenst are a known problem. The article does not detail how this is done however.

4.3 Multiplexed Image Acquisition

It is possible to reduce the number of the input images by lighting multiple LEDs at once[16]. This reduces acquisition time by a factor equal to the number of lit LEDs per image. Each iteration step in the algorithm is extended to incorporate all the information from the lit LEDs at once. This does not significantly reduce the number of computations however, as the information from each illumination angle still has to be applied one by one. This version of the algorithm thus primarily reduces image acquisition time, not reconstruction computation time.

4.4 Segment tiling

The coherence requirement in FPM reconstruction means that large images must be subdivided into segment small enough for the pixels to be illuminated with mutually coherent illumination. These segment must necessarily be processed independently, and are thus a clear and often exploited target for task parallelism. However, due to the periodic nature of the discrete Fourier transform, the hard cutoff at the segment edges leads to edge artifacts. There are techniques to even out the edge artifacts and average the intensity, such as reconstructing segments with some overlap and then average like Treland[4], or using alpha blending as in Zheng[2]. Zheng especially uses quite small segments of 150px by 150px, and a relatively large overlap of 50px between each segment. Treland on the other hand uses segments of up to 2048px by 2048px in timing experiments, but 512px by 512px in actual reconstruction.

4.5 Other applications of alternate projections

An example of an alternating projections using the DM method in a different non-convex domain is the DM based implementation of a Sudoku solver presented by Schaad[39]. This implementation enforces the constraints of sudoku by projecting onto the space of legal columns, legal rows and legal squares. As long as two constraint enforcing steps do not overlap in the working estimate, they exhibit some of the same basic behaviour as the iteration steps in FPM and ptychography phase retrieval. The paper also describes a solver for N-queens that works by the same principle. Despite the non-convexity of the sets involved in both these problems, convergence is demonstrated.

Chapter 5

Implementation

A full implementation of a Fourier ptychographic recovery algorithm usually consists of a pre-processing step and an iterative recovery step. The implementations used in this thesis primarily focus on speeding up the iterative recovery step. All implementations use a Fresnel based light propagation model. Implementations are partitioned into single-threaded and multi-threaded. This distinction refers to the amount of processor threads working concurrently. A single-threaded application using one GPU is a single-threaded application by this definition.

The algorithm consists of **iterations** that in turn consist of **iteration steps**. Each iteration contains a one iteration step for every input image. Both full iterations and iteration steps are traditionally run in sequence, but with some relaxations to the pupil recovery scheme, it is possible to run certain iteration steps in parallel while staying close to sequentially consistent with the baseline implementation. A high-level overview of the algorithm is seen in Figure 5.1

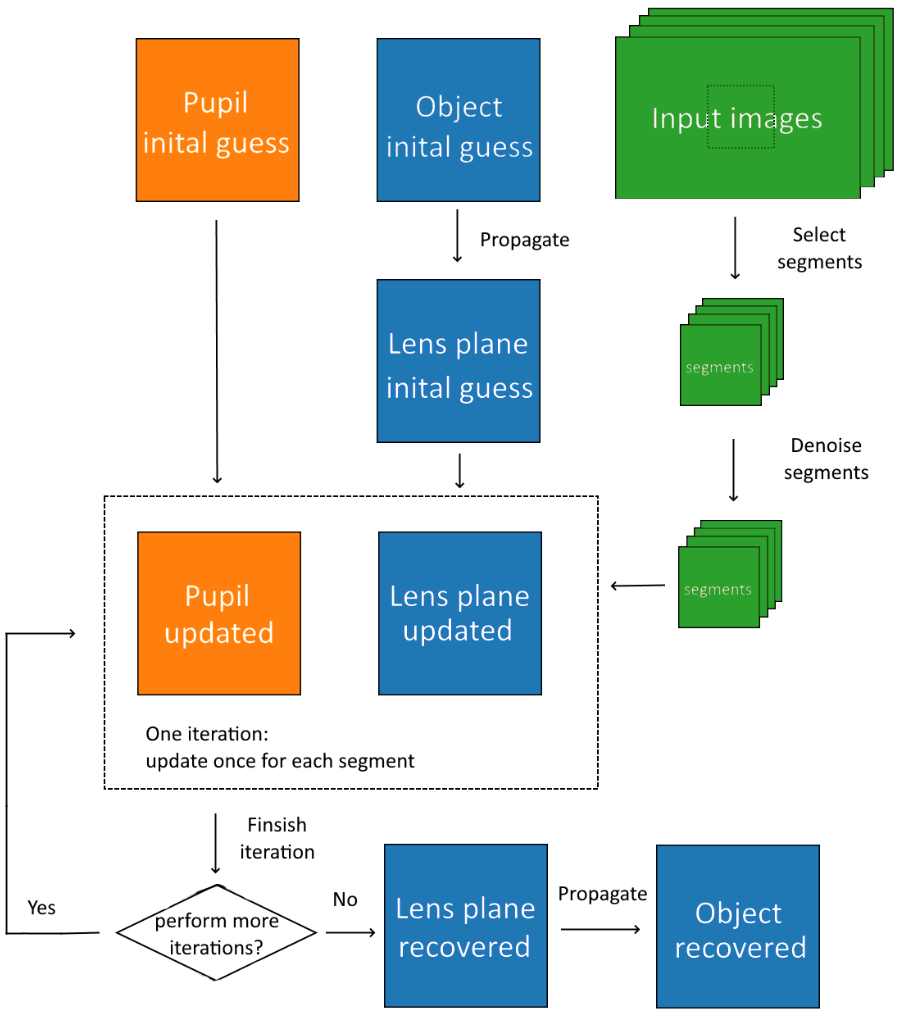


Figure 5.1: Overview of the full algorithm

5.1 Matlab baseline

The implementation used as a starting point and baseline in this thesis is developed by Professor Muhammad Nadeem Akram at the Universisty of South-Eastern Norway, and is based on the Fresnel propagation algorithm described in Kondas PhD thesis[1]. It first performs denoising and intensity adjustment of the input images, then it applies the algorithm to both recover the pupil of the imaging system and the input object. The pupil recovery uses the EPRY method[40] and is based on similar principles as the object recovery. The algorithm baseline is implemented in Matlab. It uses a single processor thread. The built in Matlab functions used exploits some data parallelism.

5.2 Baseline in c++

To more freely be able to work with different frameworks and hardware, the first implementation step was to create a c++-version that uses the same algorithm as the baseline. This required setting up a framework for buffer manipulation, simple image loading and writing, noise removal, complex arithmetic, and application of the 2D Fourier transform.

5.2.1 Image load and store

All implementations in this thesis assume 16-bit grayscale tif images, and loads them into buffers of data type double. The loading is performed by wrapping the imread function provided by OpenCV, and transferring the images from the OpenCV matrix data type to linear arrays of double precision floating point numbers (double). These arrays represent 2D buffers stored row by row (row major order).

5.2.2 Configuration by file

To more easily change algorithm and microscope parameters and design test suites for experiments, the executable is built to accept two configuration files as launch parameters. The first file specifies the microscope settings such as physical dimensions and the number and position of lit LEDs. The second file specifies algorithm settings such as input and output folders, upscale factor, segment location and other constants. The configuration file grammar is simple: each line is either a key-value pair, empty or a comment. Comment lines start with a '#', key-value pairs are written with a ':' as separator and whitespace other than newline is ignored. Invalid lines are ignored with a warning. Parameters not specified by the provided configuration files default to hard-coded values.

5.2.3 Noise removal

Following the baseline implementation, the noise removal (denoising) is implemented by a thresholding scheme. The goal of this scheme is to remove background illumination from the images. Background illumination is estimated for each image based on the average value in the 100 by 100 pixel square at the top left and bottom right corner. The estimated average value is stored for all images. When selecting segments from the images, the estimated background value for the image the segment is copied from is subtracted from all pixels in the segment, if it is below the threshold of 1600. If the average is above 1600, it is assumed to be signal and is not removed.

5.2.4 Complex numbers

Complex numbers are stored in arrays of type double, with components packed as shown in Figure 5.2. This is the format several libraries, such as CUDA and GSL expect. Array access is simplified through macros that compute the actual array index for real and imaginary components based on the logical address of the complex number. The complex values **not** wrapped in their own data type to allow for full control over data layout and implementation, and keep the representation of all buffer object explicit. This does, however, require the programmer to remember which buffers contain complex numbers and which buffers contain only reals. In future implementation, using types and structs to wrap the buffers and make them easier to work with would probably be worthwhile.

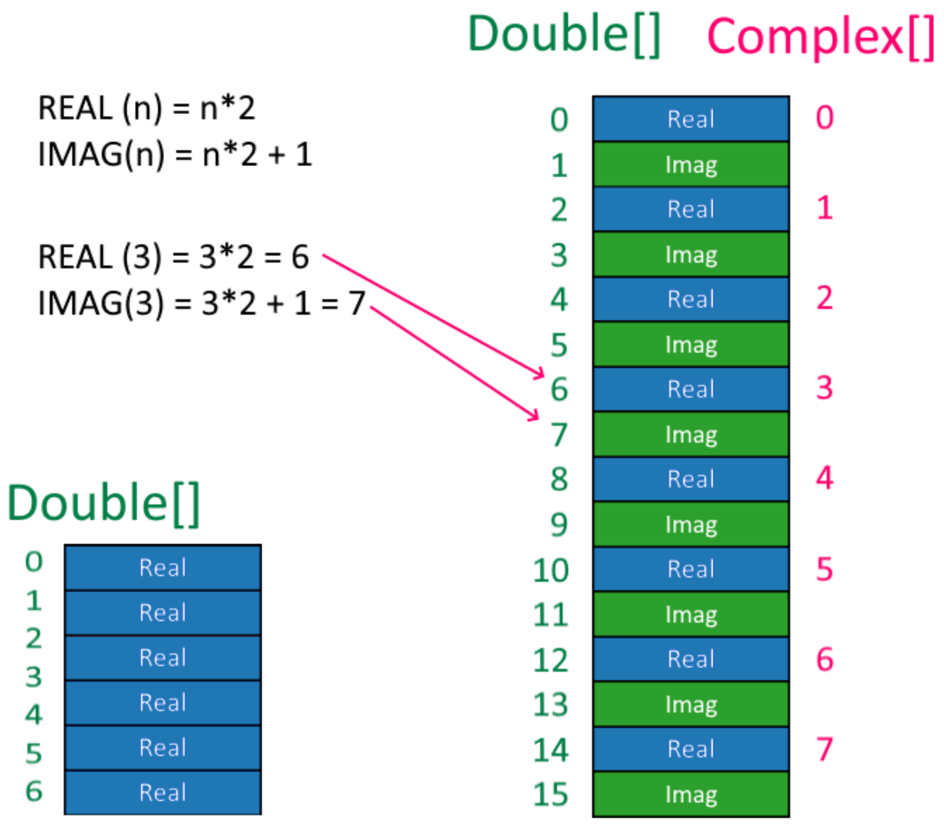


Figure 5.2: Data layout of real and complex buffers, and index calculation

5.2.5 Buffer manipulation

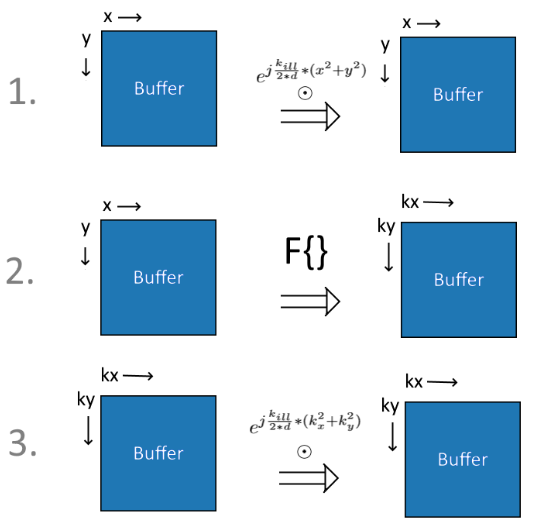
A goal throughout development has been to abstract the reconstruction process into manipulation of two-dimensional data buffers. The buffers contain either real or complex values. All buffer manipulation happens through buffer manipulation functions. These functions are specifically written to perform a (logically) single operation, and could thus be unit tested for correctness during development. The implementation contains separate buffer manipulation functions for real and complex buffers. The four main types of buffer operations are:

- Operations that copies a buffer; whole or in part
- Operations across all elements of a buffer
- Operations that transform the buffer, like DFTs
- Operations that combine two buffers

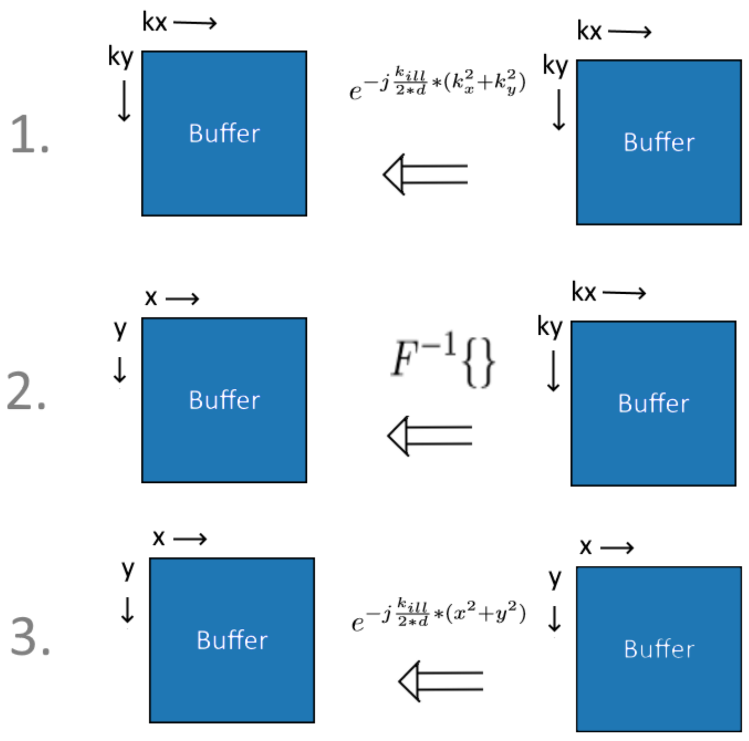
5.2.6 Propagation steps

The basic propagation operation under Fresnel propagation takes a buffer as input, multiplies it with a phase factor, Fourier transforms it, and multiplies it with a final phase factor. It is illustrated in Figure 5.3. The dot in a circle indicates element-wise multiplication.

In the FPM algorith, when propagating between object plane and lens plane only step 1. and 2. are applied. When propagating between lens and detector only step 2. is used, as the detector plane projection does not modify the phase. Also, the pupil of the imaging system low-pass filters the lens plane estimate before it is propagated to the detector plane. The propagation steps of the full algorithm is illustrated in Figure 5.4



(a) Forwards propagation



(b) Backwards propagation

Figure 5.3: Free space Fresnel propagation of buffer between planes as a series of buffer operations

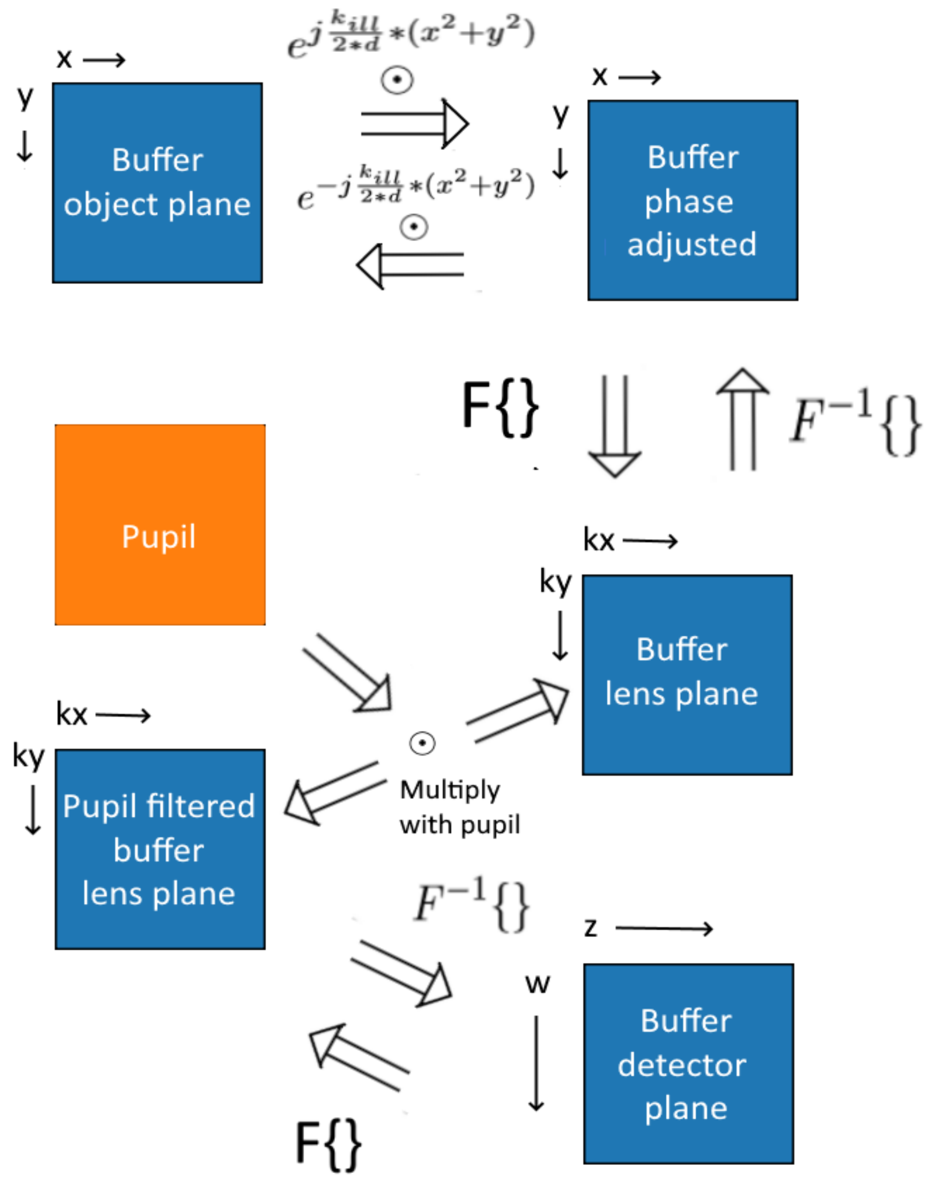


Figure 5.4: The buffer operations required for propagation in the algorithm. The circled dot represents element-wise multiplication

5.2.7 Pixel sizes and magnification

All the three planes of the imaging system (object, lens, detector) are sampled by **a grid of pixels** in the algorithm. The pixel grid at the detector plane is the size of the input low-resolution images, the pixel grids at the lens and object plane have the same dimensions as the result high-resolution image. These three pixel grids are used to store the light field as it is propagated between the planes in the recovery algorithm. All the planes have different pixel sizes, caused both by the nature of Fresnel propagation, the magnification of the imaging system and the upscaling effect of the reconstruction. The different pixel sizes are calculated as part of the algorithm setup. As an example, the pixel sizes of the imaging system used to capture the open FPM data set from Tian Labs[41] are now derived and presented in Table 5.1 and Figure 5.5

Assume the input image size in pixels is 512 by 512. This is the dimensions of the low resolution pixel grids at the detector plane. Choosing an upscale factor of 4 in each dimension yields high resolution pixel grids of 2048 by 2048 pixels at the lens and object plane. The imaging system has a magnification of $M = 8.1485$, and a camera with a sample pixel size of $p_d = 6.5 * 10^{-6}m$. The magnification, upscale factor and detector plane pixel size gives the object plane high-res pixel size $p_o = \frac{p_d}{M*4} = 2.0 * 10^{-7}m$.

To find the pixel size in the lens plane, equation (2.18) is multiplied with the low resolution pixel size at the object plane. The distance from object plane to lens plane is $d_{ol} = 0.0422 * 10^{-3}m$ and the wavelength of the illumination is $\lambda = 6.292*10^{-7}m$. The extent of the object plane $W = 2048*0.19*10^{-3} = 0.41*10^{-3}$ is the object plane pixel size times the number of pixels in a given dimension. The resulting pixel size in the lens plane is $65 * 10^{-6}m$, an order of magnitude larger than the pixels in the object plane.

Table 5.1: Size of the grids from the Tian labs example

location	pixel dimension	pixel width	plane width
Object plane	2048 * 2048	$0.19 * 10^{-6}m$	$0.41 * 10^{-3}m$
Lens plane	2048 * 2048	$65 * 10^{-6}m$	$133 * 10^{-3}m$
Detector plane	512 * 512	$6.5 * 10^{-6}m$	$3.33 * 10^{-3}m$

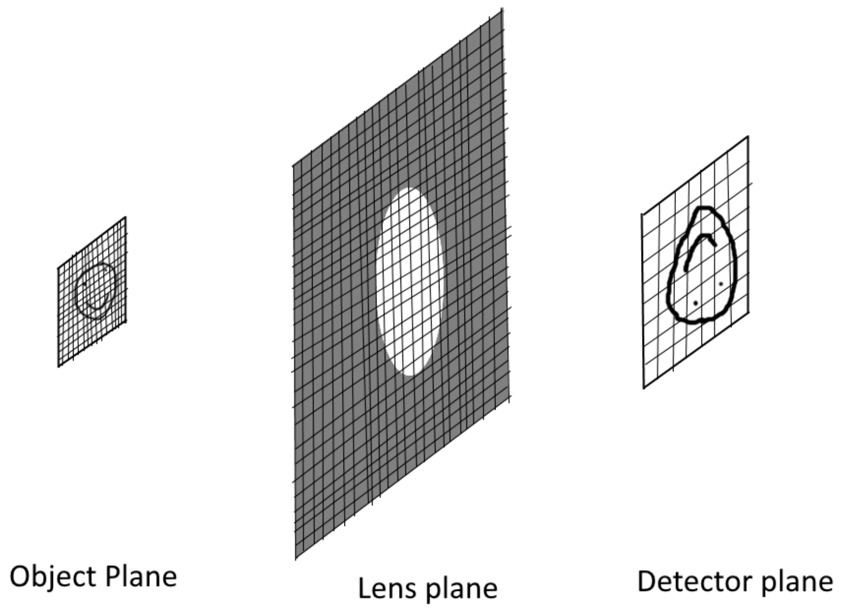
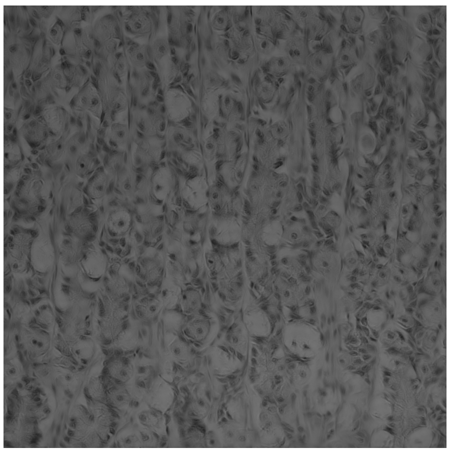
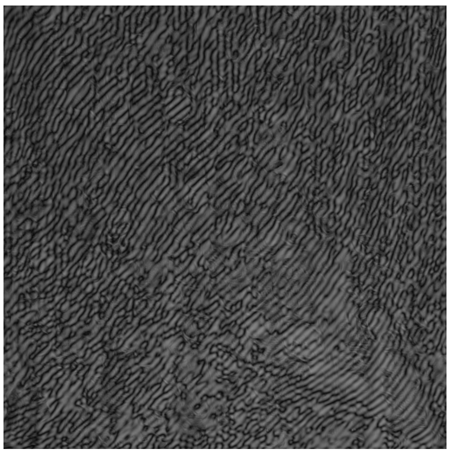


Figure 5.5: The three pixel grids used for the three planes in the algorithm. With Fresnel propagation, the pixel size varies between planes

5.2.8 Calculating offsets

Each iteration step is associated with a single selection of the lens plane recovery estimate, which is the starting point of the iteration, as well as the target for the updated segment. The offsets are determined both by the angle of illumination provided by the associated LED, as well as the offset of the center of the patch currently under reconstruction. It is important to know whether the images in the current data set are flipped or not. A single lens imaging system flips the image of the object, meaning that a segment offset of (x, y) at the result image corresponds to a shift at the object plane of $(-x, -y)$. However, some data sets have this flip corrected, in which case the segment offset coordinates must **not** be flipped. If the flip/lack of flip is not properly accounted for, the result is artifacts in the reconstruction like those shown in Figure 5.6. It is thus important to check if any given input data set is flipped or not, so the shift calculations can be updated accordingly

The offsets is determined by equation (2.21), and the pupil size is determined by the physical size of the lens aperture. Both the shift and pupil diameter is divided by the lens plane pixel size so that it is sampled by the same grid as the lens plane estimate. The final pattern for the open data set provided by Tian labs[41] is shown in Figure 5.7.

(a) Offsets based on correct segment center

(b) Offsets based on incorrect segment center

Figure 5.6: A comparison between the result of correct pupil offsets (left) and incorrect pupil offsets (right)

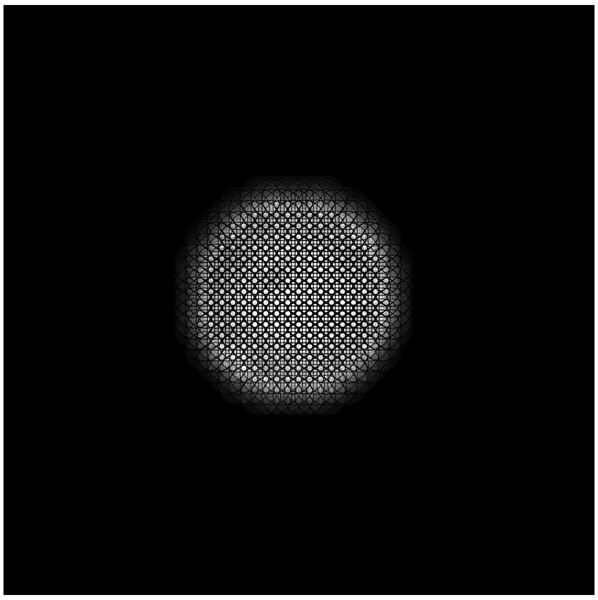


Figure 5.7: The sampling pattern of the Tian labs data

5.2.9 The 2D Fourier transform

As the recovery process requires two discrete Fourier transforms to propagate the estimate for each iteration step, it is important that it uses a fast implementation to avoid becoming a major bottleneck. For the C++ baseline, the implementation provided by GSL was deemed fast enough, as it yielded similar single-core running time to the Matlab implementation. The transform is normalised in the backwards direction, that is, when transforming from a spectral to a spatial representation.

When low-pass filtering the lens plane estimate for propagation to the detector plane, the pupil function filter assumes that the lens plane is Fourier shifted. A Fourier shift of a spectrum exploits the fact that the discrete Fourier transform is periodic to move the location of the zero frequency of the 2D spectrum from the upper left corner to the center. This reinterprets all frequencies above the Nyquist limit as negative frequencies, by using the fact that given a sample rate SR and any integer k, any sine wave at a frequency of F is indistinguishable from (or "aliased by") a sine wave at a frequency of $F + (k * SR)$ [42]. A the effect on a 1D signal with sampling rate of 6 is shown in Figure 5.8, while a 2D example from the algorithm is shown in Figure 5.9.

The convention used in this implementation is to shift the Nyquist and all higher frequencies to the negative side, following the behaviour of the matlab functions `fftshift` and `ifftshift`. The Fourier transform functions and most buffer operations expect the zero frequency to be in the upper left, and the filtering operations expects the zero frequency to be in the center.

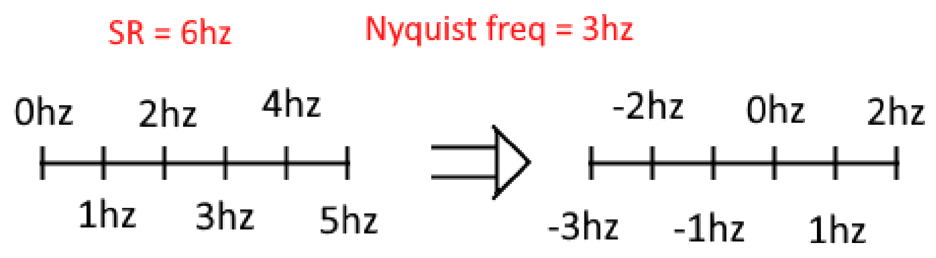


Figure 5.8: one dimensional shift. Nyquist frequency and above are shifted into closest negative aliases. 4hz to -2hz, 5hz to -1hz and so on

When working with DFTs, it is important to keep in mind where the results are normalised. In GSL, the forward transform multiplies all output pixels up by N*M, where N is the number of rows and M the number of columns, and the reverse transform divides all the pixels with N*M. If a sub section of a buffer that has been forward transformed by this DFT implementation is to be reverse transformed, the ratio between pixels in the full buffer and pixles in the the sub-buffer must be included in the energy adjustment performed in the inverse DFT. An illustration of the required scaling is provided in Figure 5.10

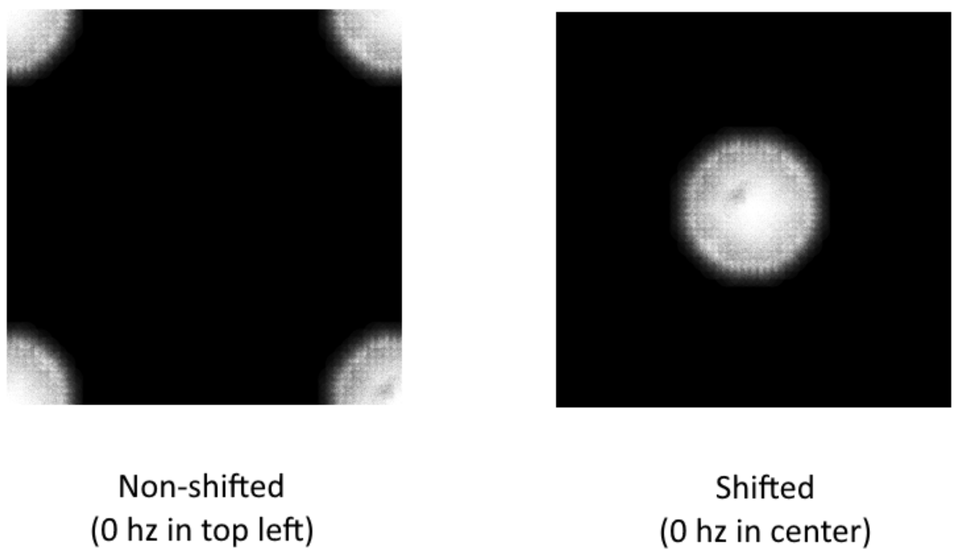


Figure 5.9: Non-shifted and shifted versions of the lens plane estimate

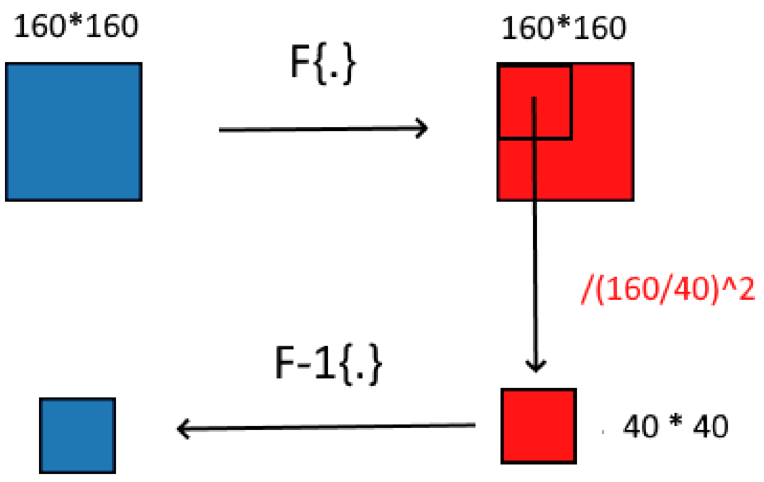
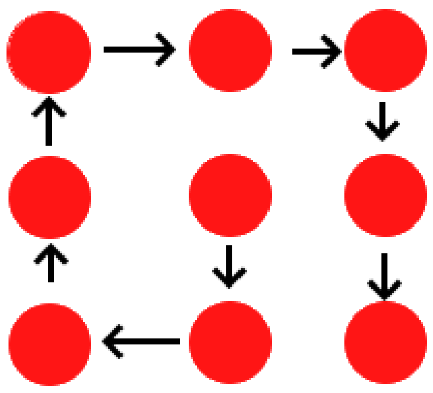


Figure 5.10: Difference in scaling when reverse transforming sub sections of a previously transformed buffer. Illustration not to scale

5.2.10 Iteration Step Sequence

The sequence of iteration steps should ideally follow be sorted by the intensity of the input images associated with each iteration step. A common heuristic to achieve this without actually measuring the intensity of every single input image is to assume that the input image illuminated by the central LED has the highest intensity. This is because the central LED illuminates the sample at no or only a very slight angle. As a consequence, the part of the spectrum that is let through


(a) LED sequence

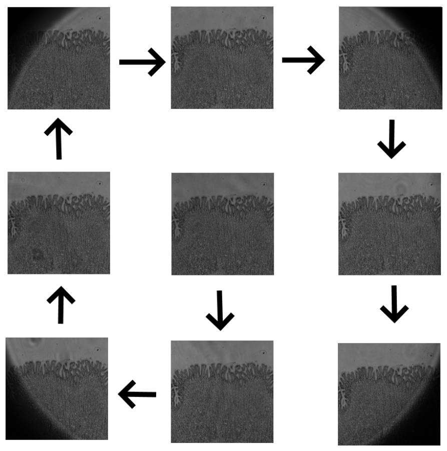

(b) Images corresponding to LEDs from the Tian labs data set

Figure 5.11: Iteration step sequence from LED positions. Initial LED is the one illuminating the image at the smallest angle. The image intensity drops quickly as the iteration progresses away from the center

the low pass filter of the lens contains the low frequency information of the original image. Usually, it is the low frequencies that contain the most intensity [6].

Since smaller illumination angles lets the low frequencies pass, and large angles lets high frequencies pass, the illumination angle and intensity of the images will be related. This means that if a sequence of images is constructed, ordered by the angle of the illumination, an order close to descending intensity order is acquired. A common way of doing this is to start at the image illuminated by the central LED, and spiral outwards as illustrated in Figure 5.11.

In the implementation, an illuminating LED and its associated input image can be identified by one of 3 indices. The **LED index** runs row by row from the top left LED. The **image index** also runs row by row from the top left, but only counts lit LEDs. Finally, the **sequence index** follows the location of the image/LED in the iteration sequence. They are all 0-indexed. To indicate not applicable values, for instance the sequence index of an image not in the sequence, the index -1 is used. A small example is shown in Figure 5.12, illustrating the three indices.

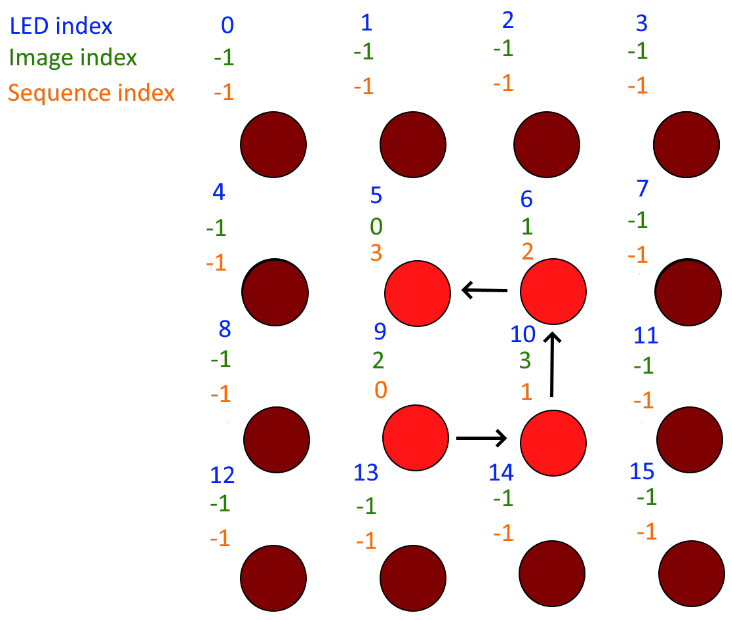


Figure 5.12: An example of a 4 by 4 LED grid with some lit LEDs (in bright red) and an iteration sequence. The indices are written at the top left of each LED, and each LED corresponds to one image

5.2.11 Initial guess

The recovery algorithm needs a starting point for the lens plane estimate. The most common method is to upscale the low resolution input image illuminated by the central LED, but a simpler alternative is to initialise the lens plane estimate to be $1+0i$ at every pixel. In the c++ baseline version, the lens plane is initialised as $1+0i$, as it works better than a very naive implementation attempt of an image upscale. One major problem with the implemented upscaling is that is introduces impossible frequencies in the Fourier spectrum of the lens plane estimate that significantly reduces output quality, as shown in Figure 5.13. Note that the spectra are shifted to show the zero frequency in the middle rather than to the left. This could probably be improved by using an image library for better upscaling and removing the impossible frequencies, but initialising with 1 works well enough that other initialisation is left as future work.

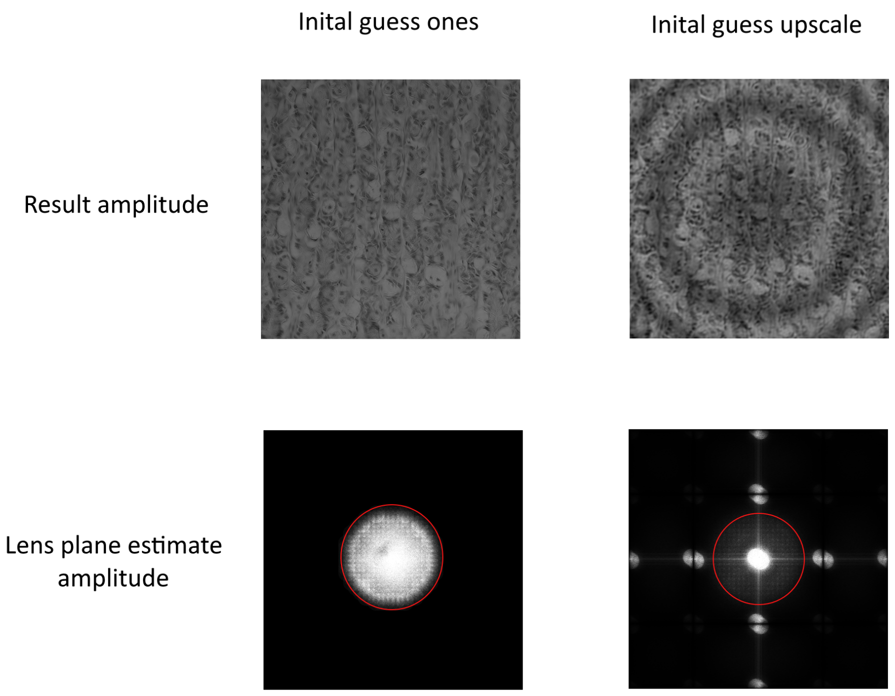


Figure 5.13: Result amplitude in object and lens plane for the same algorithm parameters under different initial guesses. The red circle indicates roughly the area of the lens spectrum where legal values can show up. Any values outside are artifacts of the reconstruction process

5.2.12 The recovery process

The c++ baseline performs the following buffer operations in a single iteration step:

1. Find location of segment selection corresponding to illumination shift and copy it.
2. Low pass filter this selection with the pupil function.
3. Reverse Fourier shift to move zero frequency to the top left
4. Reverse Fourier transform to propagate the selection to the detector plane
5. Project low resolution image onto selection, by replacing the amplitude from the estimate with the amplitude (square root of intensity) from the image as shown in equation (2.15)
6. Fourier transform to propagate adjusted selection to the lens plane
7. Use the old selection and the image adjusted selection together with the pupil function to create an updated version of the selection through a quasi Gauss-Newton method as shown in equation 2.16
8. Subtract the part of the lens plane within the shifted pupil

9. Replace the subtracted part of the lens plane with the updated selection
10. Use the old selection and the image adjusted selection as well as the old pupil estimate to update the pupil estimate. This is performed using equation (2.22)

5.3 Multi-threaded

When creating a multi-threaded implementation of the iteration step in Fourier ptychography, the main challenge is how to safely decouple the iterations of the algorithm. For a given image segment, the two factors introducing dependencies is the overlap in Fourier spectrum information between images from different illumination angles, and the iterative estimation of the pupil function. Coordination between threads can be organised in a centralised or distributed manner. The centralised approach is realised using a main thread controlling a team of worker threads fetching from a priority queue. The distributed approach partitions the input, and relies on border exchanges to ensure consistency in overlapping regions. In this work, the centralised approach is used.

5.3.1 Iteration step overlap

For each iteration step in the linear version of algorithm, a single image and corresponding area of the high-resolution lens plane is used. To parallelise the algorithm on a iteration step level, the steps must be distributed across multiple threads. To ensure good convergence even in the presence of noise and aberrations, the overlap between the lens plane areas of the images should be 60-80% of the area[12, 43]. This requirement does, however, introduce dependencies between iteration steps that access the same parts of Fourier space. If the algorithm works in parallel on dependent iteration steps, race conditions may occur.

The race conditions happen in the overlapping pixels of the segments associated with the concurrently executed iteration steps. The lens plane estimate is only read and written once for each iteration step. Intermediate calculations happens in work buffers private to each thread, and do not themselves contain any race conditions. The most likely form of race condition is thus that the thread that finishes its iteration step last gets to overwrite all other pixels in the overlapping region with its own result. This reduces the data redundancy of the overlapping pixels, and might lead to slower convergence. To at all be able to solve the phase recovery problem, each pixel must be adjusted at least two times for each full iteration with data from at least two images[1]. However, the more times a given pixel is iterated over in the lens plane estimate, the more robust the reconstruction process becomes to noise and microscope miscalibration[12]. Permitting some overlap in the execution may thus increase the number of full iterations needed to get a good reconstruction. The more errors in the imaging system, the more pronounced this effect might get

The precise overlap varies between datasets and is dependent on the specific physical characteristics of the microscope that captured the images as well as the layout of the illuminating LEDs. As an example, one of the freely available data sets from Tian labs that shows a histology slide[41] gives images with an area overlap of circa 70% in Fourier space. This means that each image in this data set in general overlaps with 42 others. However, many of the overlaps are very small. Permitting some overlapping concurrent work on the different iteration steps can significantly increase the available concurrency without introducing a significant amount of race conditions. The difference in number of iteration steps depending on the central iteration step considering 100% and 70% of the pupil radius is illustrated in Figure 5.14.

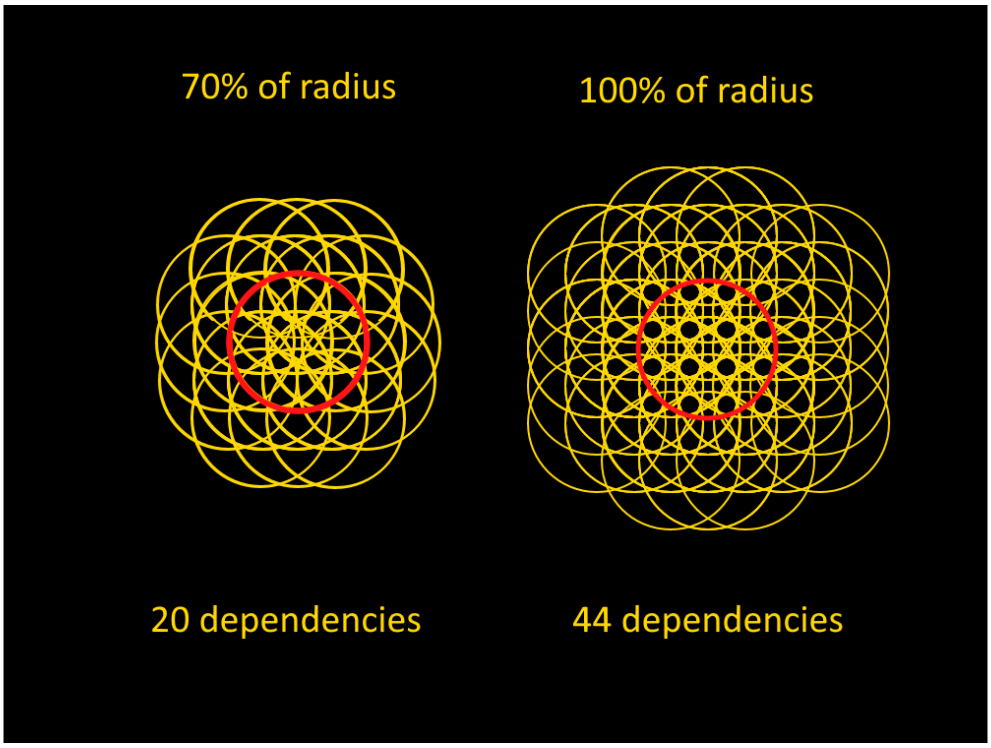


Figure 5.14: The difference in dependent iteration steps when considering 100% of the pupil radius vs 70% of the pupil radius

To create a mechanism that makes sure no dependent iterations are computed concurrently, a graph where each image is a node and each edge indicates a dependency/overlap is set up. An example graph for a synthetic data set with much lower degree of overlap than found in the Tian labs data sets is shown in Figure5.15. Using this graph, it is possible to create a thread safe priority queue containing all the images that only lets an image be taken out if no dependent images are currently being worked on.

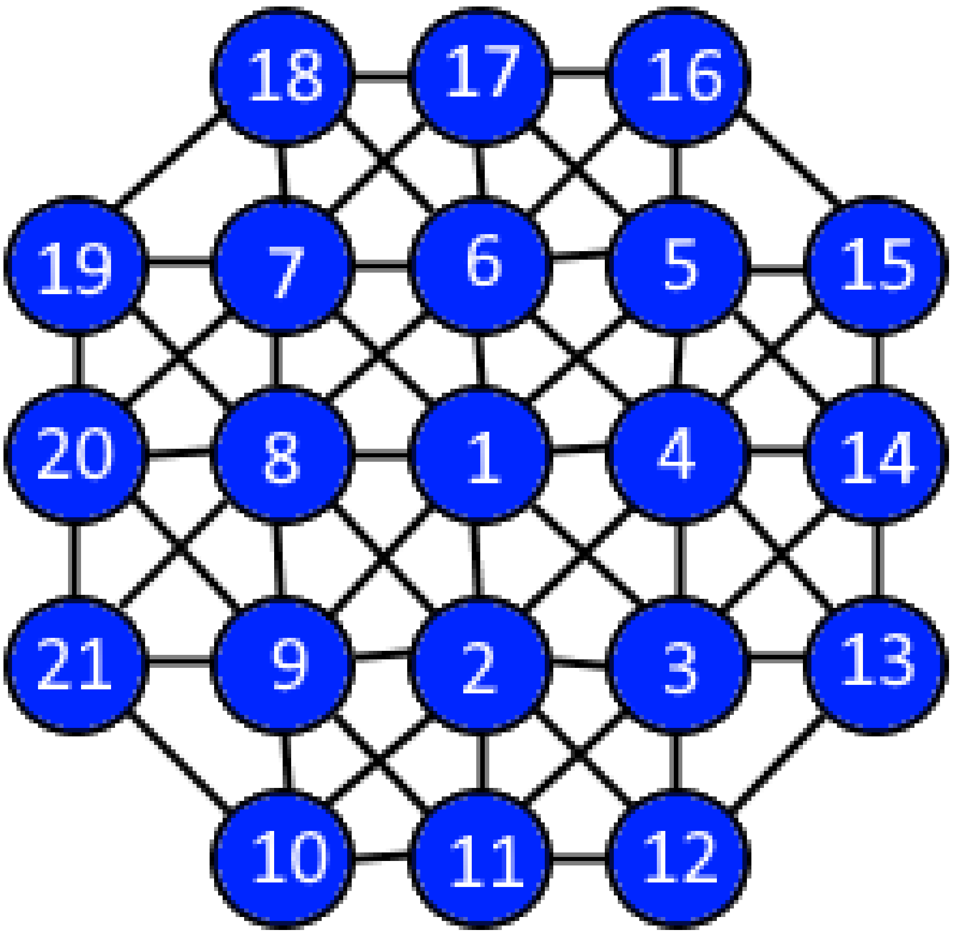


Figure 5.15: An example of a dependency graph from a synthetic data set. In real data sets, the nodes are generally connected to more neighbours

Multi-threaded spiral

The first method is to use the single-threaded queue and distribute a set of iteration steps for the current thread group to run. The distribution process follows the spiral order, but respects the overlaps. The more threads, the more this sequence must deviate from the original. The closer to the center, the greater the effect, as iteration steps corresponding to high intensity input images are fewer in number and overlap more than the steps corresponding to low intensity images.

Multi-threaded spiral with deviation tolerance

To avoid the effect where the early, high intensity iteration steps get performed far out of order, a tolerance parameter can be added to the priority queue. This parameter can the be used to make sure the queue only lets the work threads fetch iteration steps to work on if is not to far out of order with respect to the single-threaded sequence. This tolerance is implemented by counting the number of complete predecessors an iteration step has in the baseline order, and dividing this by how many of them are complete. This ratio is compared with a deviation threshold. This deviation metric is stricter for iteration steps early in the order, which might be an advantage, as much of the intensity and information resides in the initial, central iteration steps. The first element is defined to have a ratio of 1.

5.3.2 Pupil Function Estimation Relaxation

In the ideal case the pupil function of the optical system is a simple low-pass filter. However, no imaging system is perfect. Applying the phase recovery algorithm on a data set from a real microscope with the assumption of an ideal pupil will thus lead to errors in the reconstruction. Correct recovery requires that the coherent transfer function of the pupil model used in the recovery procedure models the aberrations of the real system. If the aberrations are known a priori, a suitable mathematical model can be applied to the coherent transfer function during setup[6]. However, in this work the pupil function is estimated iteratively as the algorithm progresses according to the principles from[40].

The baseline implementation includes an update of the pupil estimate every iteration step. This introduces a common dependency between all iterations that prohibits any concurrent iteration step execution.

To decouple the iteration steps, the pupil estimate update regime must be relaxed. This can be achieved by pre-computing the pupil and reusing it for later image reconstructions, or by reducing the frequency of pupil estimate updates such that only some iteration steps update it. A third option is to fully estimate the pupil function for certain tiles and reusing this pupil function in neighbouring tiles under the assumption that the pupil aberrations vary very slowly with space[9]. A fourth option is to partition Fourier space and estimate a different pupil function for each partition, analogous to the asynchronous parallel approach described in Nashed et al.[35],

All of these options will reduce the quality of the pupil reconstruction, but without them, the algorithm is strictly iterative. Striking the right balance between pupil estimation accuracy and parallelism is important. In this implementation, the pupil update frequency is reduced. [According to Dag, this is reasonable, as the pupil still gets plenty of updates. How to cite the conversation?]

5.3.3 Work Distribution by Priority Queue

To run multiple iterations steps concurrently, a data structure is required that can provide an independent set of iterative steps from the dependency graph. In this implementation this structure is realised as C++ class that both creates the dependency graph and provides methods for checking out, prioritising and completing iteration steps. The checkout method returns an iteration step that is guaranteed to be independent from all other iterations steps that are currently being executed. If an independent iteration step is requested and none are available, the index -1 is returned. A queue working on a synthetic dataset is illustrated in Figure 5.16. Here the base single threaded order is a spiral running from the centered, with nodes numbered in this order. Four threads are currently working on node 1, 10, 11 and 12. This has locked most other nodes in the graph, but node 16 through 20 are still available if another thread request work.

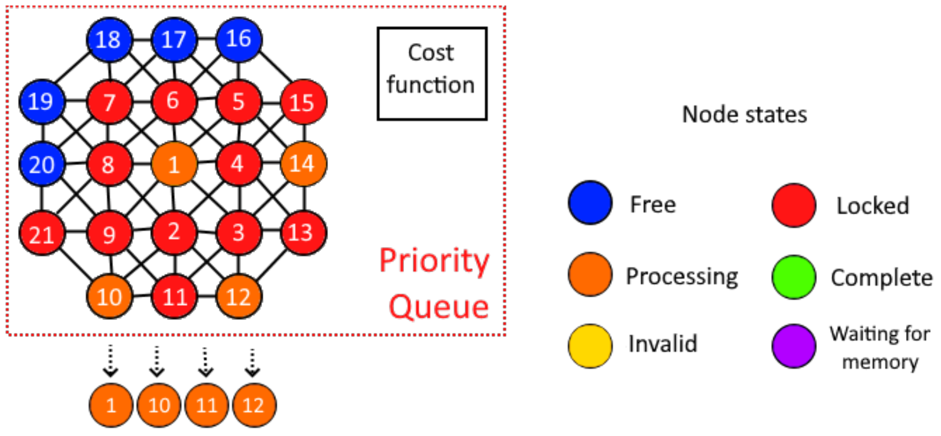


Figure 5.16: An example of a priority queue working on a synthetic data set

The priority queue is able to serve independent iteration steps by using both the dependency graph and an array of locks. The locks are integers, one for each iteration step, that counts how many currently executing iteration steps that depend on a given iteration step. When a thread requests a new independent iteration stuff to work on, the priority queue finds an iteration step with lock counter value of zero, increases the lock counter of the selected iteration step and all dependent iteration steps with 1 and returns the independent iteration step. Once a thread is done with an iteration step, it can call a method on the queue to unlock all its dependencies. The completed item, however, is not unlocked. This ensures each item is only operated on once until the queue is reset.

The work queue can provide as many independent iteration steps as requested. In the implementation, this is used to identify and run batches of concurrent iteration steps. For a given dependency graph, the number of independent nodes will be limited. This puts a hard limit on how many iteration steps can be processed concurrently. To increase this limit, the priority queue can be instructed to create a dependency graph that permits some overlap and thus increasing the number of independent nodes. The downside of this is that the overlapping pixels will get fewer iterations. Disregarding the overlap might also lead to race conditions when different iteration steps work concurrently on overlapping lens plane estimate regions.

5.3.4 Thread coordination

Threads are launched in synchronised batches. Each batch consists of a fixed number of threads which are all provided with independent iteration steps to execute. The iteration step independence is important, as all the threads update the same lens estimate memory buffer. Once all threads in a batch are done, a pupil update is performed using the work buffers of the thread with the lowest ID in the batch (this choice is arbitrary, any thread will do). As a consequence, using more threads per batch will reduce the update frequency of the pupil.

5.3.5 Changing the Iteration Step Order

Running independent iteration steps concurrently changes the order of execution. Where a single threaded implementation processes neighbouring iteration steps one by one, two concurrently executing iteration steps cannot be next to each other without introducing large overlaps and potential race conditions. However, a strictly enforced no-overlap policy makes the concurrent iteration step order deviate quite a bit from the original order for just a few threads, and significantly for a larger number of threads. This can severely affect the quality of reconstruction. To ensure good reconstruction quality, a balance between strict enough overlap management and the number of concurrent threads must be struck to keep both the reconstruction quality and the available parallelism high.

5.4 GPU acceleration

As GPU acceleration and other exploitation of the inherent data parallelism in the buffer operations can be applied independently of the task parallel methods used in this implementation, the original plan was to implement both for even greater speedups. However, as the task parallel implementation was the main focus and time was limited, the GPU functionality was not fully implemented, and is not part of the experiments performed.

Chapter 6

Methodology

The purpose of the experiments is to validate that the multi threaded FMP implementation produces correct results, and how much speedup can be attained without significantly reducing image quality. A higher concurrent iteration step count will lead to greater deviation from the optimal intensity order. Different strategies to minimise this effect are tested.

6.1 Optimising the baseline

The baseline single threaded version was optimised by iteratively using perf to identify the most computationally intensive functions in the code, and speeding them up. Running time was measured using the bash utility `time`. The optimisation continued until the processor running time was similar to that of the Matlab baseline for 5 iterations on the same data set ($\tilde{1}$ minute and 30 seconds). The Matlab implementation was still faster in wall clock time, but this was due to data parallel processing. As data parallel processing was not the main focus of this thesis, the equal processor running time was considered more important than the unequal wall clock time. For this thesis, the attainable speedups without data parallel processing was the main focus.

6.2 The computer

The computer used in this experiment is a LAB computer at the NTNU HPC lab. Specifications are given in Table 6.1.

6.3 Reconstruction quality

The reconstruction quality of a given experiment can be evaluated qualitatively through determining the smallest resolvable feature in a USAF resolution target, or quantitatively through model match, mean square error(MSE) and structural

Table 6.1: Computer specification

Part	Name	Details
CPU	AMD Ryzen 7 5800X	8 cores 16 threads
RAM		DDR4 2667MHz 2x8GiB
GPU	GTX 1080 Ti x 2	11 GB GDDR5X memory
Operating System	Ubuntu	20.04.2
Secondary memory		500GB M.2 SSD

similarity index (SSIM) relative to some ground truth. A single threaded, 20 iteration run of the baseline algorithm is used as a ground truth in this thesis. Model match is implemented as part of the algorithm in C++. MSE and SSIM methods are implemented in python. The python code and installation instructions are provided as snippets in the appendices.

6.3.1 Result representation

Final and intermediary results from experiments are written to 16-bit tif images for storage. Each solution is represented by two images, one representing amplitude, the other representing phase. Amplitude and phase are chosen as they show more information about the sample than real and imaginary rectangular coordinates. The amplitude images are scaled by a configurable constant (200 unless otherwise noted). This is to make the amplitude images visible when viewed in an image viewer. The absolute value of the phase is scaled to $[0, 65535]$ when written to file. This destroys information about whether the offset from 0 phase is positive or negative, but yields more useful images, as shown in Figure 6.1 and is in line with the rendering process used in the Matlab baseline. When using these images as input to other algorithms, these conversions must be reversed to recover the original image.

6.3.2 Model Match

Using the calculated estimate with the forward model 2.19 and comparing the result with the recorded images makes it possible to quantify how close the current estimate and forward model fits the recorded images data. This can be calculated during (at a significant computational cost) or after execution to measure convergence and reconstruction quality. This is done by taking the absolute difference between the amplitude of the recorded images and the predicted images according to the lens plane estimate, as in equation (2.14). The difference is divided by number of images and number of pixels to show the average pixel error across all input images.

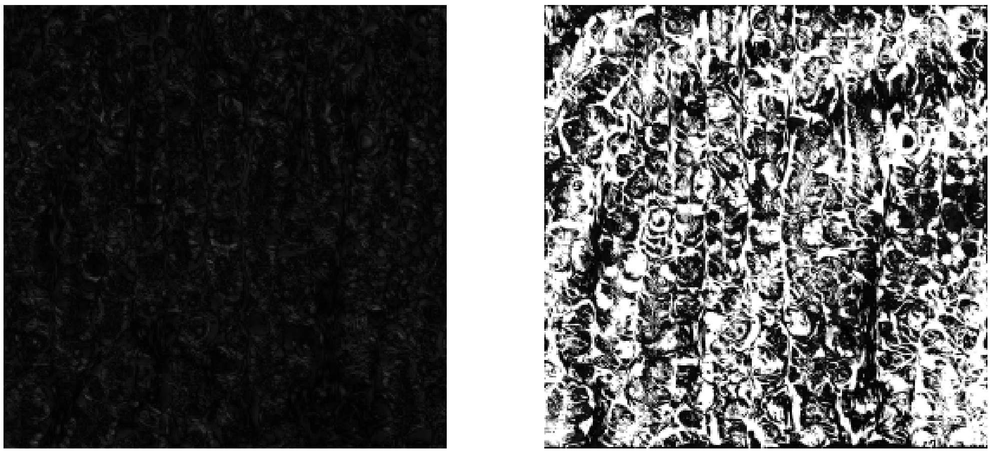

(a) Absolute value of $[-\pi, \pi]$ **(b)** 0 to 2π

Figure 6.1: Different renderings of phase from recovered images

6.3.3 Similarity indexes

To compare obtained results to some ground truth, both Mean Squared Error(MSE) and Structural SIMilarity index (SSIM) are used. Both metrics are normalised to a range of $[0, 1]$. MSE is implemented in python and uses the squared magnitude of the difference between two complex images. The implementation of SSIM is also in python, but uses a library implementation from scikit-image[44]. SSIM is used on phase and amplitude separately, as they can be seen to converge at different rates.

6.4 Algorithm parameters

The main parameters of interest in this thesis are the number of threads, degree of overlap relaxation and iteration step order. The parameters used in the Gauss-Newton iterative method follows the Matlab baseline. Image parameters such as size, upscale factor, segment size and segment location are fixed across all experiments using the same data set.

6.4.1 Sequence difference

As the behaviour of the algorithm is influenced by the order of iteration steps, a metric for the difference between two different sequences can be used to compare the difference in reconstruction quality to the difference in iteration step order. The metric used in this work is the ratio between possible completed predecessors in the base iteration order and actually completed predecessors. This gives a metric between 0 and 1 that measures how out of order the selected iteration actually is. The first element has no possible predecessors, and is assigned an order metric of 1.

6.5 Datsets and setup

The data sets used are from Tian labs. They image a dog stomach and a USAF target respectively. Both data sets are obtained using the same microscope, and contain the same amount of images of the same size. The parameters are shown in Table 6.2

Table 6.2: Tian labs data parameters

Parameter	**Tian labs**
Flipped	yes
Dimensions	2560px * 2160px

6.5.1 Microscope setup

The microscope parameters are obtained from the same repository as the data sets. As the open data sets were published with Fraunhofer propagation based recovery in mind, they do not contain information about the distances between object, lens and detector. It is, however, possible to recover these numbers by using the other parameters. The final numbers used in the algorithm are the derived parameters used in the baseline matlab code provided by Muhammad Nadeem Akram. The microscope settings are shown in Table 6.3.

Table 6.3: Microscope parameters

Microscope parameters	**Tian labs**
LED matrix	32 x 32
LED pattern	circle
LED pattern diameter	19
Center LED(row, col)	(13, 14)
LED separation	4mm
Magnification	x8.1485
Numerical Aperture(NA)	0.1
Illumination wavelength	629.2nm
LED to object	67.53mm
Object to lens	42.2mm
Lens to detector	343.9mm
Detector pixel size	6.5μm

6.5.2 Segment size

The segment size used is determined by the coherence width as given by the formulas in Section 2.1.1. For the Tian labs microscope, the values obtained for the parameters is given by Table 6.4. The most difficult parameter is the width or area of the LED. Schnell[8] use red LEDs (625nm) with an area of $130\mu m * 130\mu m$. As all data sets in this thesis are taken using red light LEDs, we assume the LED dimensions are comparable. The segment size is set to be **512 pixels in each dimension**. This is a little above the coherence lengths obtained, but the images produced show little to no artifacts.

Table 6.4: The max number of pixels in each dimension that maintain mutual coherence

Source	Expression	Tian labs result
Konda 2018	$I_c = 1.22 * \frac{\lambda z}{w}$	500px
Konda 2020	$I_c = \frac{\lambda z}{w}$	409px
Schnell 2019	$A_c \approx \frac{l^2 \overline{\lambda}^2}{A_s}$	409px
Chung 2019	$L = 0.61 \frac{\lambda z}{a}$	500px

6.6 Experiments

The experiments are run in different suites. The suites are "baseline", "multi threaded with strict overlap handling", "multi threaded with relaxed overlap handling", "multi threaded with stricter reordering limits", "consistency" and "performance". All test suites are run with 20 iterations, except "performance", which is run with 10. All experiments are run over all data sets unless otherwise noted.

6.6.1 Baseline

Two baselines are created, one using a USAF target data set, the other using an open source stained histology data set. Both data sets are from Tian labs and acquired with the microscope described in the Tian labs columnt of Table 6.3. The baseline reconstructions are preformed by a single thread, to represent a conventional Fresnel based recovery algorithm. At each full iteration, the image produced by the current estimate is saved. The algorithm settings for the baseline is shown in Table 6.5.

6.6.2 Multi-threaded

The main way to alter the multi threaded algorithm is to vary the **number of threads** and the **distribution of iteration steps**. Altering either of these will change the order of iteration steps. This may have a significant effect on algorithm

Table 6.5: Baseline algorithm parameters

Algorithm parameters	Baseline
Full image	2560px x 2160px
Input image format	16-bit grayscale tif
Segment	512px x 512px
Upscale factor	4
Segment upper left(x,y)	(800, 824)
Threads on CPU	1
Iterations	20
Lens projection method	Gauss Newton
Iteration order	anticlockwise spiral
Noise threshold	1600
gnNumStability constant	1
gnStepFactor	0.1
Significant overlap	100%
Sequence deviation tolerance	100%

convergence and thus reconstruction quality.

Strict queue, no deviation penalty

These experiments are run with the same parameters as the baseline, with the exception of the number of threads. Work items are distributed in a way that respects the overlap between iteration steps, with no penalty for reordering the original sequence. The setup is shown in Table 6.6.

Table 6.6: Parameters for the strict queue experiments

Experiment name	Threads	Iterations	Significant overlap	Deviation tolerance
strictQueue 1	2	20	100%	100%
strictQueue 2	4	20	100%	100%
strictQueue 3	8	20	100%	100%
strictQueue 4	16	20	100%	100%

Non-strict queue, no deviation penalty

The introduction of a **significant overlap** parameter permits a priority queue that can yield iteration steps that tolerate some overlap with other concurrently executing iterations steps. This parameter is in the range $[1,0]$, where 1 represents full respect for overlap and 0 no respect for overlap. Adjusting this parameter towards 0 permits the execution of iterations closer in the lens plane, at the cost of reduced redundancy. The reduced redundancy is a result of the race conditions in the overlapping pixels between concurrently executing iteration steps. Only one of the overlapping iteration steps will get to perform the final write to a given overlapping pixel.

The reduced overlap is a potential disadvantage, but reducing the deviation from the preferred single-threaded order should be an advantage, as it has been shown that the spiral, approximately intensity order that the single threaded implementation follows provides better results than for instance a random ordering[6]. The purpose of the experiments in this section is to investigate how this trade-off affects the convergence and quality of the reconstructions. The parameters for these experiments are given in Table 6.7.

The experiments were run in two iterations, with the experiments prefixed with "2-" being the follow ups to investigate the effect of increasing overlap tolerance further when running 16 threads. As the overlap percentage refers to overlapping radius and not overlapping area, it is expected to behave non-linearly in terms of area overlap. For instance, changing "significant overlap" from 0.5 to 0.4 is a greater change than changing it from 0.6 to 0.5.

Table 6.7: Parameters for experiments with overlap tolerance

Experiment name	Threads	Iterations	Significant overlap	Deviation tolerance
lessStrictQueue 1	2	20	75%	100%
lessStrictQueue 2	2	20	50%	100%
lessStrictQueue 3	4	20	75%	100%
lessStrictQueue 4	4	20	50%	100%
lessStrictQueue 5	8	20	75%	100%
lessStrictQueue 6	8	20	50%	100%
lessStrictQueue 7	16	20	75%	100%
lessStrictQueue 8	16	20	50%	100%
lessStrictQueue 2-1 (9)	16	20	25%	100%
lessStrictQueue 2-2 (10)	16	20	0%	100%

Non-strict queue, deviation penalty

To get more control of how far the execution order deviates from the single-threaded order during a multi-threaded execution, a new parameter called "Deviation tolerance" is introduced. Where the "significant overlap"-parameter adjusts what iterations are available to choose from for the priority queue, the "deviation tolerance" adjusts what available iteration step the priority queue chooses on the basis of how out of order it would be with respect to the single-thread order. Reducing the deviation tolerance should improve reconstruction quality, as it enforces a better iteration order, but it also reduces the available concurrency. The extreme case is when the deviation tolerance is set to 0. In this case, the algorithm degenerates back into the single threaded algorithm. The experiments are performed with a number of threads and a "significant overlap" parameter that has shown diverging behaviour in the previous experiments, to see if the new "deviation tolerance" parameter can help improve their convergence. The tested parameters are shown in Table 6.8

Table 6.8: Parameters for deviation tolerance experiments

Experiment name	Threads	Iterations	Significant overlap	Deviation tolerance
deviation 1	8	20	100%	75%
deviation 2	8	20	100%	50%
deviation 3	8	20	100%	25%
deviation 4	8	20	75%	75%
deviation 5	8	20	75%	50%
deviation 6	8	20	75%	25%
deviation 7	8	20	50%	75%
deviation 8	8	20	50%	50%
deviation 9	8	20	50%	25%
deviation 10	16	20	100%	75%
deviation 11	16	20	100%	50%
deviation 12	16	20	100%	25%
deviation 13	16	20	75%	75%
deviation 14	16	20	75%	50%
deviation 15	16	20	75%	25%
deviation 16	16	20	50%	75%
deviation 17	16	20	50%	50%
deviation 18	16	20	50%	25%

6.6.3 Consistency

As long as no overlap is permitted, the same parameter combination should yield the same result image. However, when overlap is permitted, race conditions between overlapping concurrent dictate the final result. To get an impression of how much variation this causes, "lessStrictQueue8" and "lessStrictQueue2-1" are each run 4 times for the Tian USAF target data. These two configurations are chosen since they both use 16 threads with a high tolerance for overlap (50% and 25% respectively). Then their convergence graphs relative to the baseline will be presented and compared. For comparison, "strictQueue4" is also run four times on the Tian

USAF data set and presented the same way. This experiments has the same number of threads, but 100% "significant overlap". This experiment should have no race conditions, and is expected to behave deterministically with equal convergence graphs.

6.6.4 Convergence and quality

The convergence metrics are based on similarity to a baseline. However, the precise mapping between convergence metric and quality is not obvious. To get some quantitative measurement of these, phase images of the Tian labs USAF target with SSIM in specific ranges are collected and displayed for visual inspection. The images are cropped to only show the smallest group of the resolution target. The goal is to find images with phase SSIM in the ranges [95, 100], [90, 95], [85, 90], [80, 85], 75 and below, with 2 images per group.

6.6.5 Running time and utilisation

When running the convergence experiments above and obtaining the convergence results, it can be seen that some experiments converge smoothly and close to the baseline, while others converge in a jerky and random fashion to optima further away from the baseline. We dub the convergence of the first group **well behaved**. When testing running time and utilisation, only the experiments with well behaved convergence are used. Selecting experiments that are well behaved for all data sets with a final minimum SSIM compared to baseline of 0.85 yield the experiments shown in Table 6.9. Each experiment is run 3 times with 10 iterations on the Tian USAF data set. This number of iteration is set based on the observation that most of the result convergence has already happened at this point. To better be able to compare this to the baseline, an experiment with the baseline settings at 10 iterations is run and presented for comparison. The linux utility `time` is used, and the results are the average values of `real` and `user` respectively. The results are presented with thread utilisation, running time and final convergence quality metrics.

To isolate the performance effect of reducing the number of pupil updates, two baseline experiments are run with 10 iterations, and pupil updated every 8th and 16th batch respectively. These experiments are named "baseline-8" and "baseline-16"

Table 6.9: Parameters for the performance experiments

Experiment name	Threads	Iterations	Significant overlap	Deviation tolerance
strictQueue 1	2	10	100%	100%
strictQueue 2	4	10	100%	100%
lessStrictQueue 1	2	10	75%	100%
lessStrictQueue 2	2	10	50%	100%
lessStrictQueue 3	4	10	75%	100%
lessStrictQueue 4	4	10	50%	100%
deviation 1	8	10	100%	75%
deviation 2	8	10	100%	50%
deviation 3	8	10	100%	25%
deviation 5	8	10	75%	50%
deviation 6	8	10	75%	25%
deviation 9	8	10	50%	25%
deviation 11	16	10	100%	50%
deviation 12	16	10	100%	25%
deviation 14	16	10	75%	50%
deviation 15	16	10	75%	25%
deviation 17	16	10	50%	50%
deviation 18	16	10	50%	25%
baseline	1	10	100%	100%
baseline-8	1	10	100%	100%
baseline-16	1	10	100%	100%

Chapter 7

Results

The results are presented first with the convergence and image quality of the different methods. This is followed by collections of reconstructions using the same parameters and data. Finally a comparison in running time and utilisation is provided.

7.1 Reconstruction quality

This section contains the results of the convergence experiments. Both result images and convergence graphs are acquired for each experiment. In sections with many experiments, only a selection of the result data is provided. In general, convergence plot data is prioritised over image results.

7.1.1 The baselines

Figure 7.1 shows a sub selection of the full baseline image of an USAF 1951 target, reconstructed with data from Tian labs. In this thesis in general, the selections of USAF target images cover an area of around 320 by 320 pixels after upscaling, or around $60\mu m$ according to the pixel sizes from Table 5.1. All the lines in the sample are clearly separated. The smallest lines are approximately 4 pixels wide, indicating a resolution of at least $0.8\mu m$. Phase and amplitude images are shown separately The dog stomach sample is shown in full in Figure 7.2.

The convergence of both baselines is shown in Figure 7.4. The plot curves show convergence towards final result for the phase image measured by SSIM (green), for the amplitude image measured by SSIM (orange), and for the amplitude image measured by normalised MSE (blue). The MSE plot is displayed as 1 - MSE, to make it more easy to compare to the SSIM measurement.

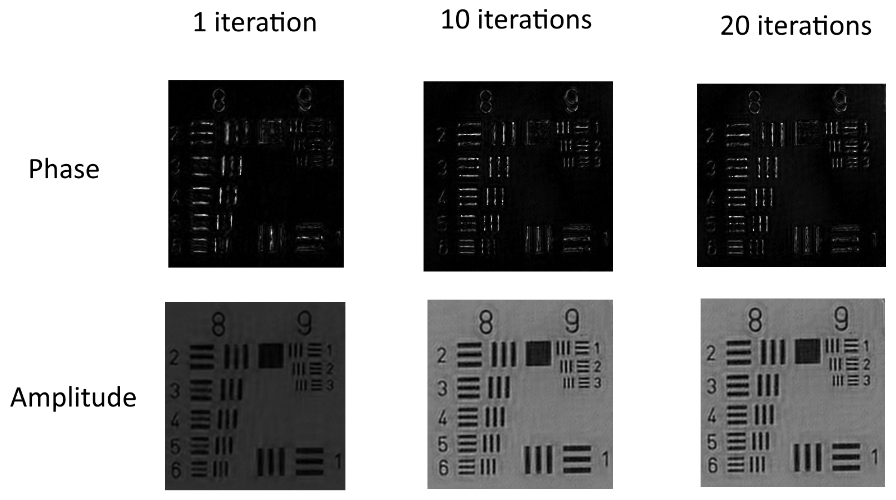


Figure 7.1: The single threaded baseline at 1, 10 and 20 iterations. Selection of Tian labs USAF target

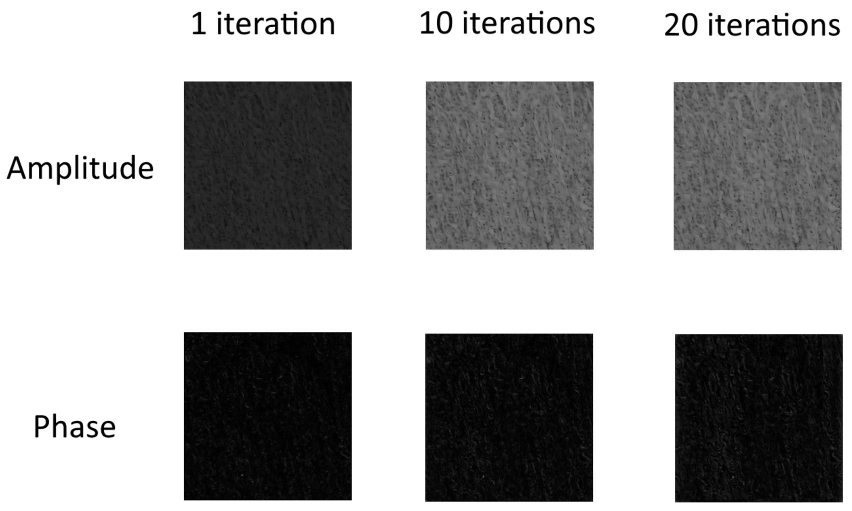


Figure 7.2: The single threaded baseline at 1, 10 and 20 iterations. Tian labs dog stomach

Figure 7.3 compares the reconstruction at 10 and 20 iterations of the same selection of the USAF resolution target shown in Figure 7.1 to one of the low resolution images in the corresponding input data set. Note that the intensity of the amplitude image has been increased to be better visible. In reality the intensity of the input image is approximately the square of the amplitude image.

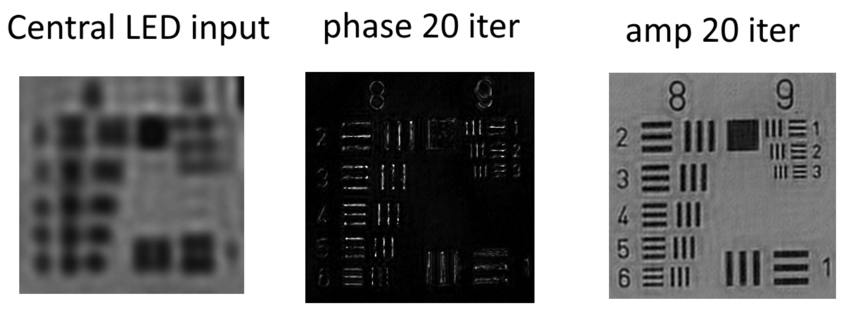
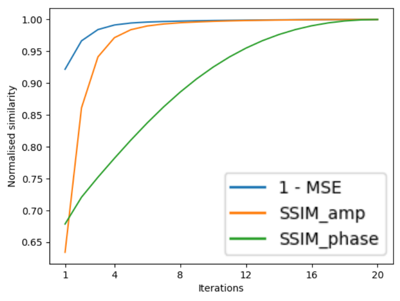
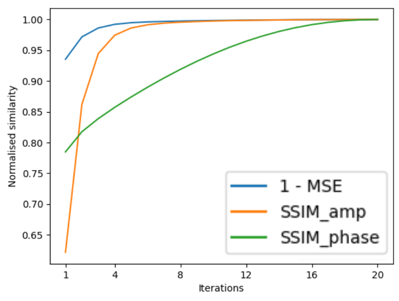


Figure 7.3: Resolution before and after reconstruction



(a) USAF 1951 resolution target



(b) Dog stomach

Figure 7.4: Convergence to result for baseline algorithm settings

7.1.2 Multi threaded

The following experiments were all performed using multiple threads. All convergence graphs are labeled with the experiment they originated from. The blue curve plots (1 - MSE) of the amplitude image relative to the baseline amplitude image. The orange curve plots the SSIM of the baseline and experiment amplitude image. The green curve plots the SSIM of the baseline and experiment phase image.

Strict queue

Phase and amplitude images from the strict queue experiments are shown in this section. Figures 7.5 and 7.6 shows the results for the USAF and dog stomac data sets. Figure 7.8 shows convergence graphs from the dog stomach data set, Figure

7.7 convergence graphs from the USAF data set. The colors in the plots are read as follows: green is SSIM of phase image, orange is SSIM of amplitude image, blue is (1 - MSE) of amplitude image. All plots are relative to the baseline of the relevant data set.

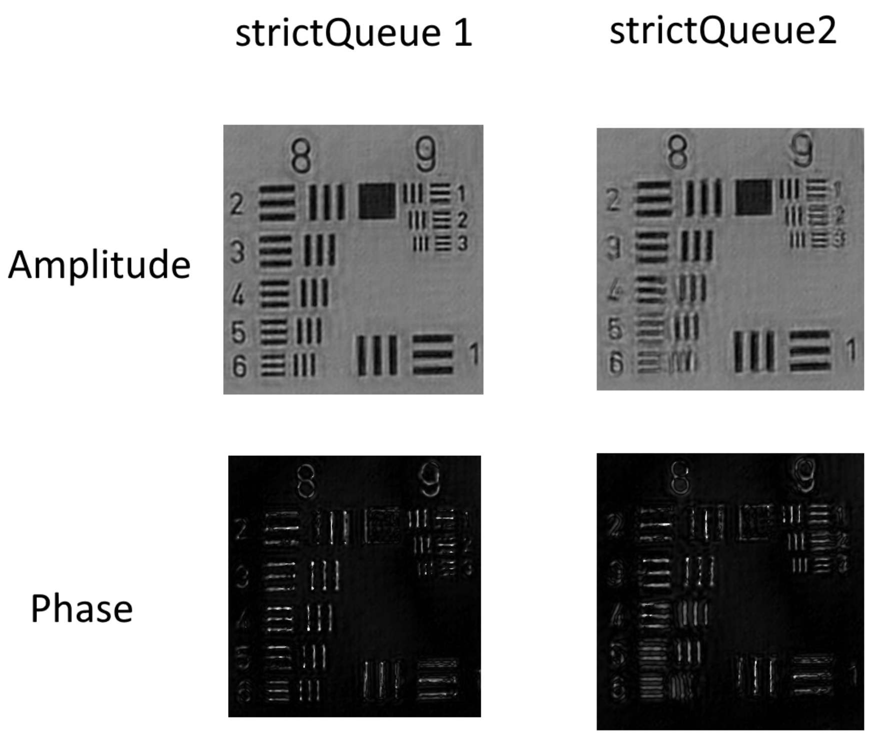


Figure 7.5: The "strictQueue" experiment results for the Tian labs USAF 1951 data at 2 and 4 threads

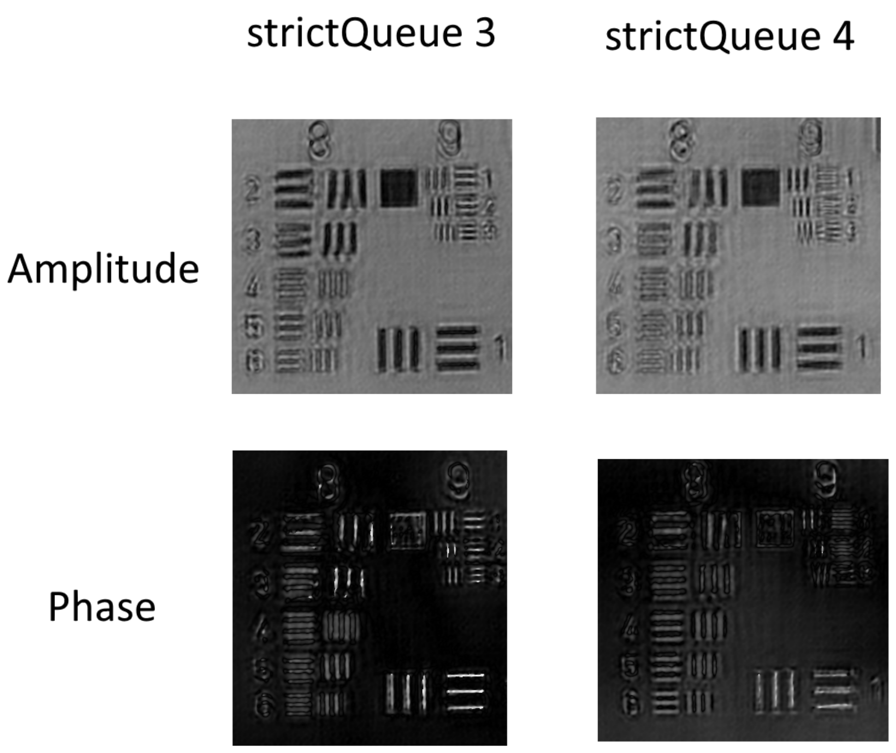


Figure 7.6: The "strictQueue" experiment results for the Tian labs USAF 1951 data at 8 and 16 threads

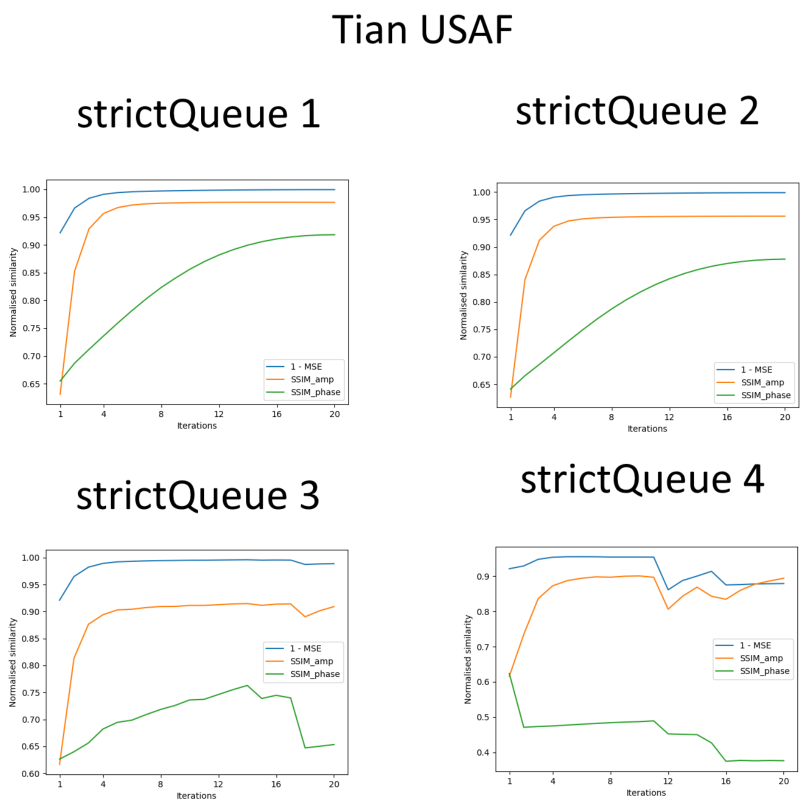


Figure 7.7: Convergence toward baseline for all "strictQueue" experiments on the Tian labs USAF 1951 data set

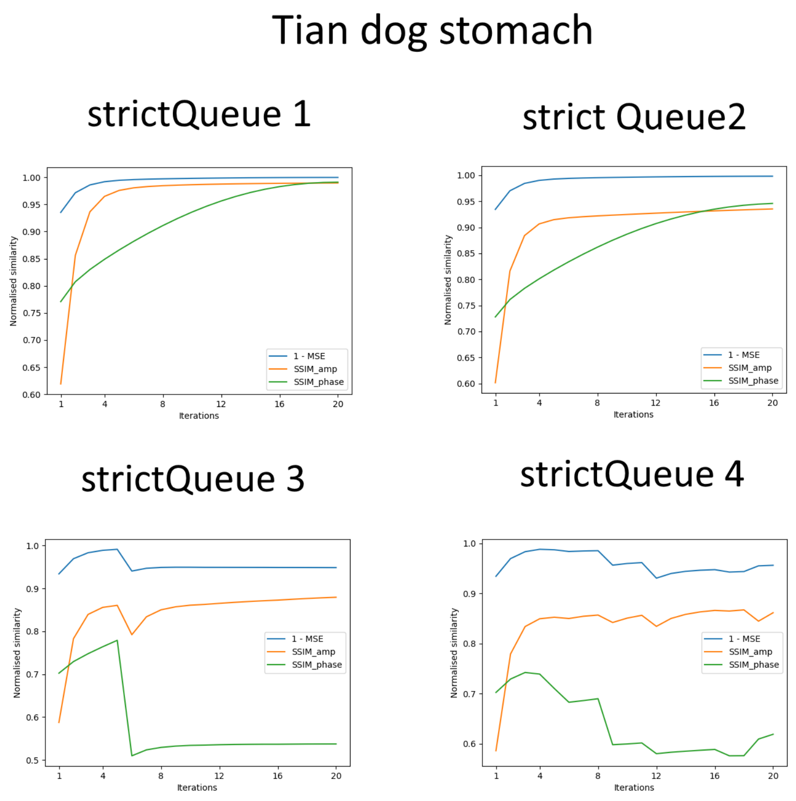


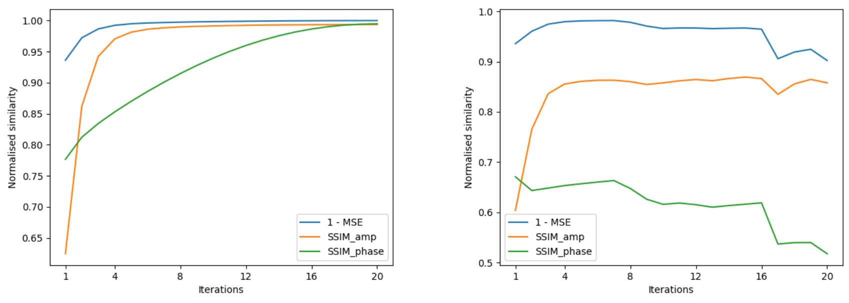
Figure 7.8: Convergence towards baseline for all "strictQueue" experiments on the Tian labs dog stomach data set

Non-strict queue, no deviation penalty

This experiment suite ("lessStrictQueue") contains a large amount of experiments. Therefore, only a selection of the convergence graphs are presented, with the full set of plots available in appendix B. The selected plots are shown in Figure 7.9, and illustrates the difference in convergence between experiment "lessStrictQueue 1" and "lessStrictQueue 8".

Non-strict queue, deviation penalty

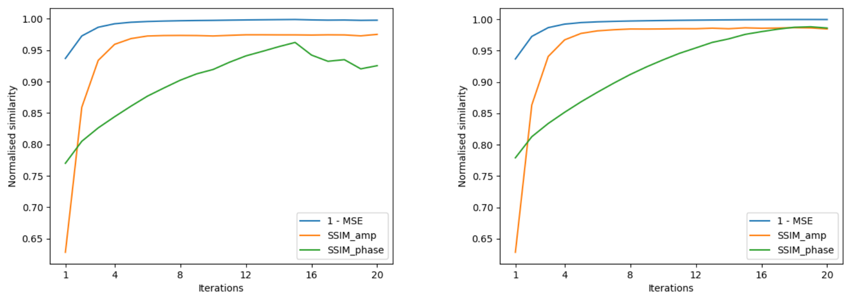
This experiment suite ("deviation"), like the "lessStrictQueue" suite, contains a large amount of experiments. The full collection of convergence graphs are available in appendix B. To illustrate the change in convergence behaviour caused by the "deviation tolerance" parameter, the convergence graph of "deviation 16" and "deviation 17" as run on the USAF data are selected from the full data and displayed in Figure 7.10. These two experiments have the same settings as "lessStrictQueue 8" except for the values of the "deviation tolerance" parameter.



(a) lessStrictQueue 1 showing good convergence

(b) lessStrictQueue 8 showing poor convergence

Figure 7.9: Example of good and poor convergence in the lessStrictQueue experiment suite



(a) deviation 16 showing poor convergence at 75% deviation tolerance

(b) deviation 17 showing poor convergence at 50% deviation tolerance

Figure 7.10: Examples to illustrate the effect of the "deviation tolerance" parameter. "Deviation tolerance" parameter set to 100%

7.2 Consistency

When overlapping iteration steps are performed concurrently, race conditions occur. These experiments were run to see what effect these race conditions have on the consistency of the convergence, that is, how much the convergence plots differ for the same algorithm settings across multiple runs. All experiments in this section had 16 threads and 100% deviation tolerance. The results for an experiment with 25% significant overlap is shown in Figure 7.11. The results for an experiment with 50% overlap is shown in Figure 7.12. The results of an experiment with 100% significant overlap, and thus ideally no race conditions, are shown in Figure 7.13. The colors in the plots are read as follows: green is SSIM of phase image, orange is SSIM of amplitude image, blue is (1 - MSE) of amplitude image.

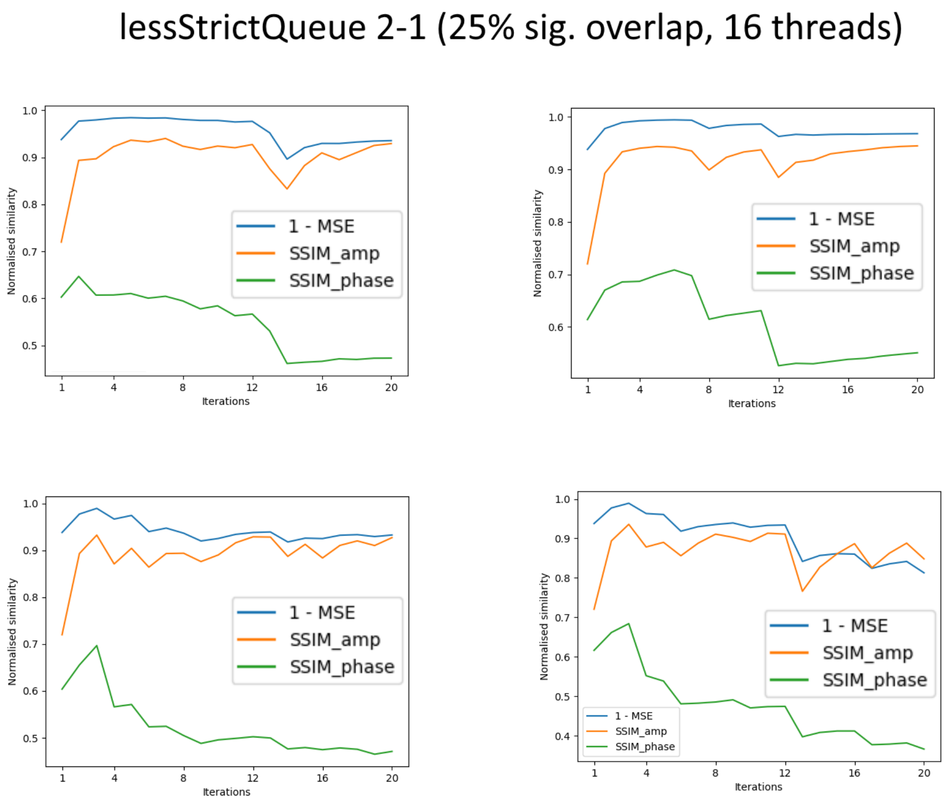


Figure 7.11: Convergence of 4 runs with 25% significant overlap

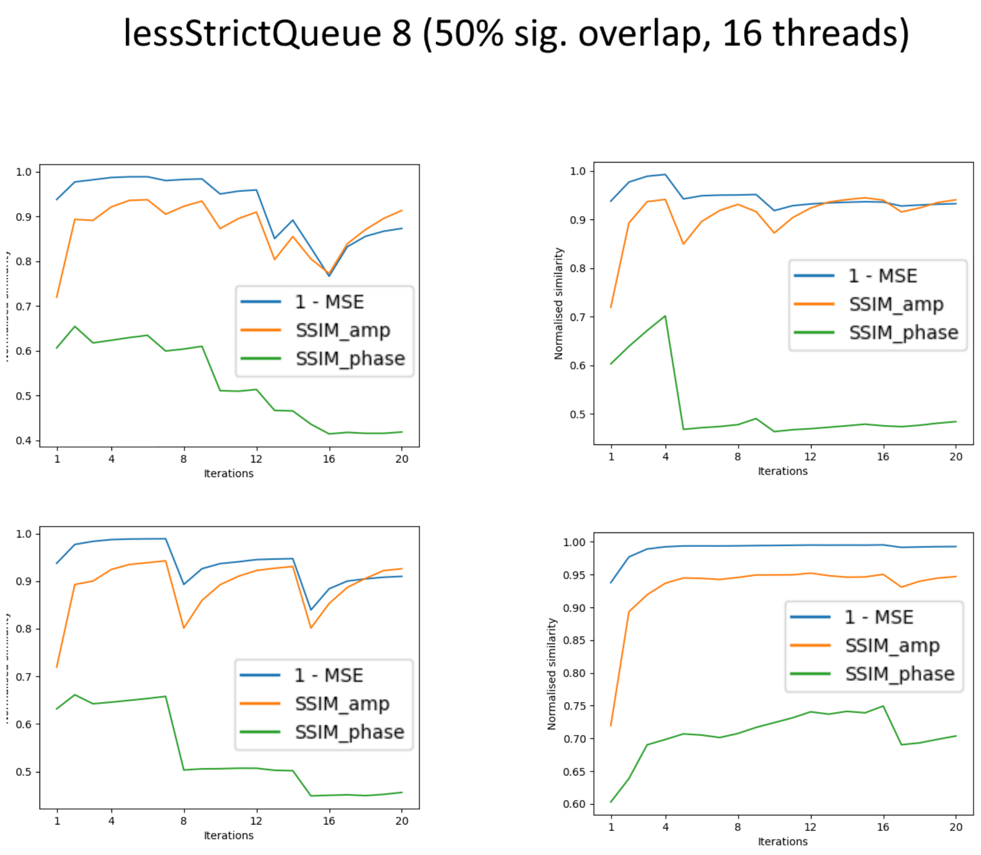


Figure 7.12: Convergence of 4 runs with 50% significant overlap

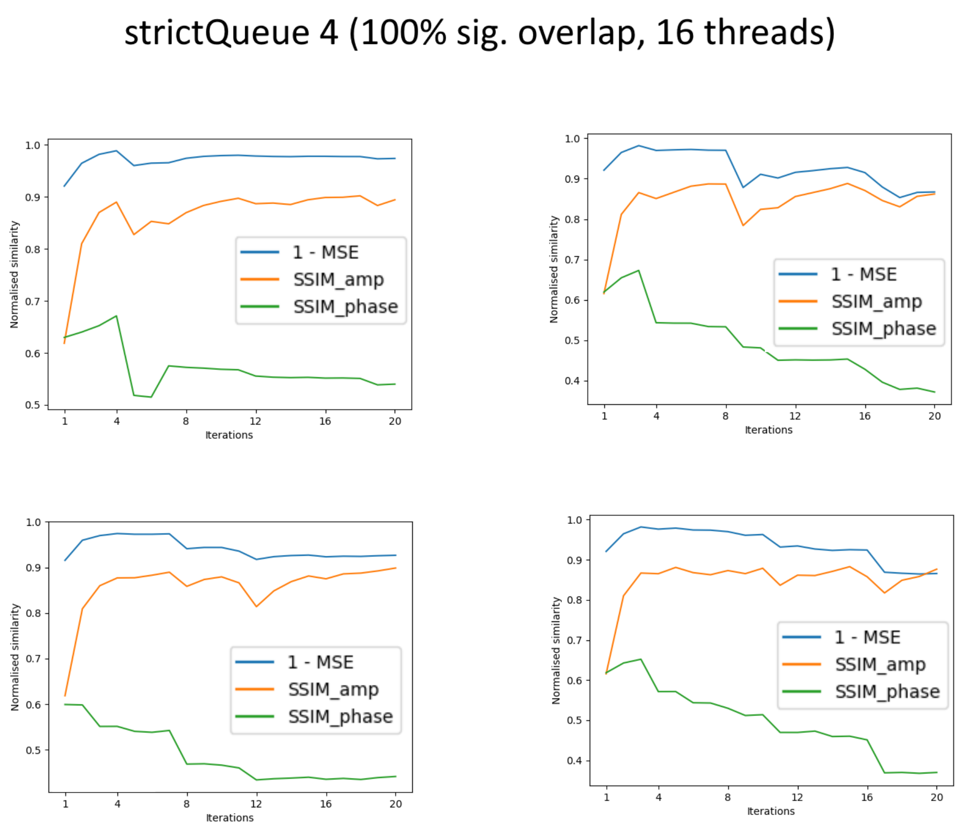


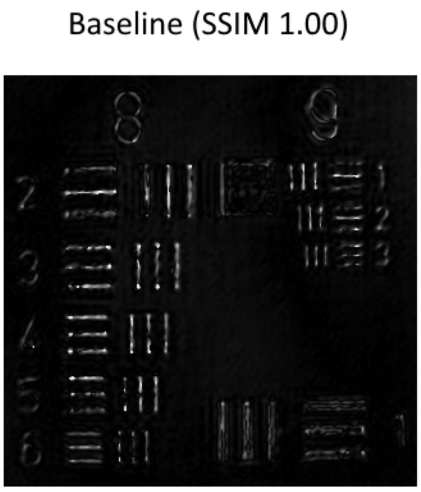
Figure 7.13: Convergence of 4 runs with 100% significant overlap

7.2.1 Convergence and quality

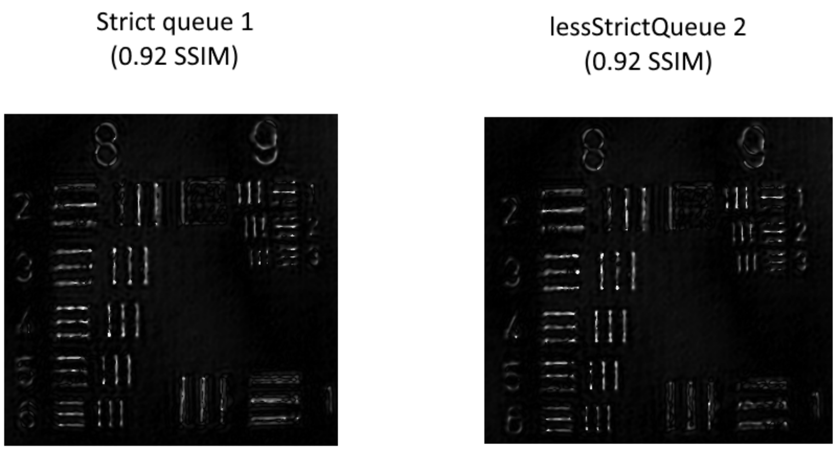
To get a better idea of how resolution, perceived quality and the image quality metrics (SSIM and MSE) compare, a selection of USAF target phase images were selected from the data produced by the experiments above, and sorted according to SSIM. Sample phase images with SSIM in the interval [90, 100] are shown in Figure 7.14. Phase images with SSIM of 90 and below are shown in Figure 7.15.

7.3 Running time and utilisation

This section shows running times and utilisation (threads per iteration step batch) for all the performance experiments. Table 6.9 shows all timing, utilisation and phase SSIMs for all performance experiments. Figure 7.16 shows the central selections of the phase image recovered from the performance experiment with the best and with the worst SSIM relative to baseline, together with the SSIM of the baseline experiments using 10 iterations and 20 iterations (the 20 iterations baseline is the baseline used for all experiment, and thus has a SSIM of 1).

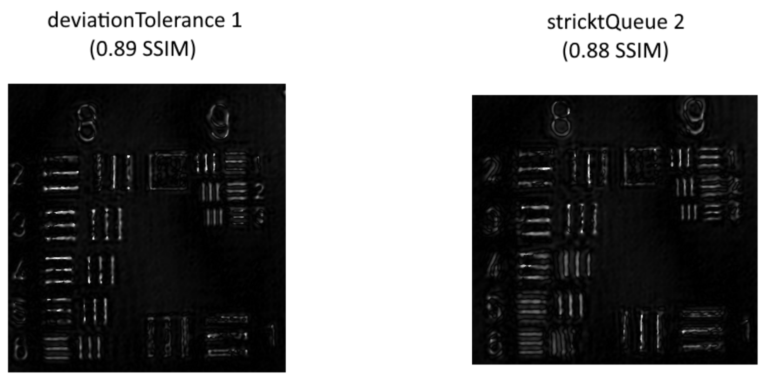


(a) SSIM in the range [0.95, 1.00] (Only result fitting criteria)

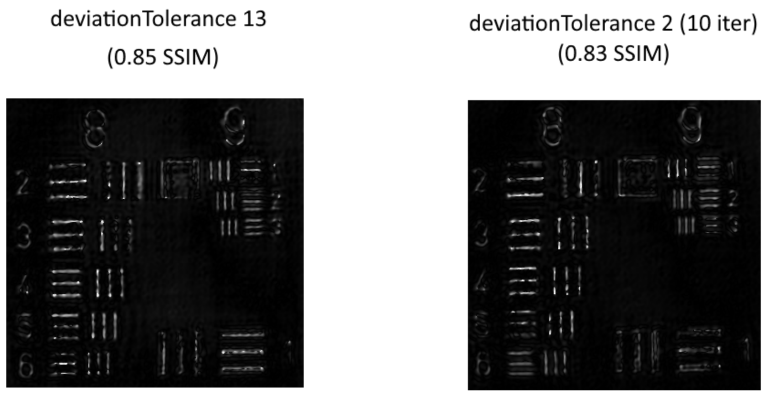


(b) SSIM in the range [0.90, 0.95]

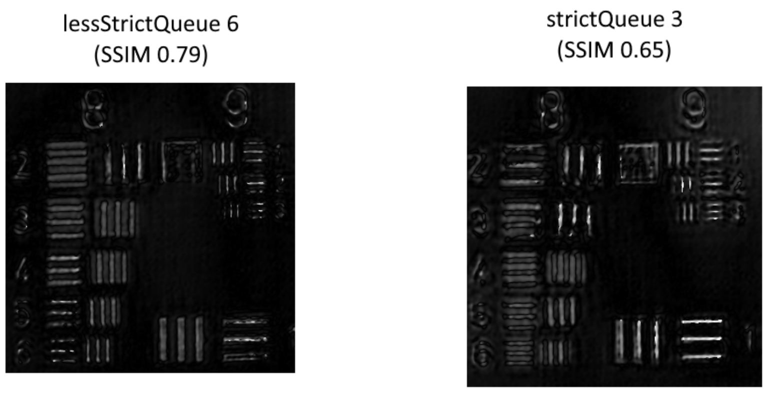
Figure 7.14: Examples of phase images with SSIM within [0.90, 1.00]



(a) SSIM in the range [0.85, 0.90]



(b) SSIM in the range [0.80, 0.85]



(c) SSIM below 80

Figure 7.15: Examples of phase images with SSIM below 0.90

Table 7.1: Results for the performance experiments, sorted by descending wall clock time.

Experiment name	**Wall clock time**	**Processor time**	**Max threads**	**Avg. threads**	**SSIM phase**
devitaion 17	30.56	288.21	16	11.27	0.85
devitaion 14	34.04	255.97	16	8.88	0.84
devitaion 18	34.79	277.50	16	6.98	0.85
deviation 15	39.51	230.43	16	6.98	**0.87**
deviation 5	40.81	179.48	8	6.37	0.85
deviation 11	41.68	197.81	16	6.23	0.85
deviation 12	41.70	197.69	16	6.23	0.85
deviation 9	42.85	182.58	8	6.10	0.86
deviation 1	42.92	183.33	8	6.10	0.83
deviation 2	44.91	180.59	8	5.64	0.85
deviation 6	44.94	182.30	8	5.63	0.86
deviation 3	50.10	180.58	8	4.80	**0.87**
lessStrictQueue 3	54.15	166.08	4	3.96	0.83
lessStrictQueue 4	54.22	166.37	4	3.96	0.82
strictQueue 2	54.71	165.04	4	3.90	0.81
lessStrictQueue 2	99.55	175.47	2	1.99	0.86
strictQueue 1	103.42	175.27	2	1.98	0.85
lessStrictQueue 1	103.67	175.90	2	1.99	0.86
baseline	203.33	203.33	1	1	**0.89**
baseline-8	159.51	159.18	1	1	**0.89**
baseline-16	154.26	153.89	1	1	**0.89**

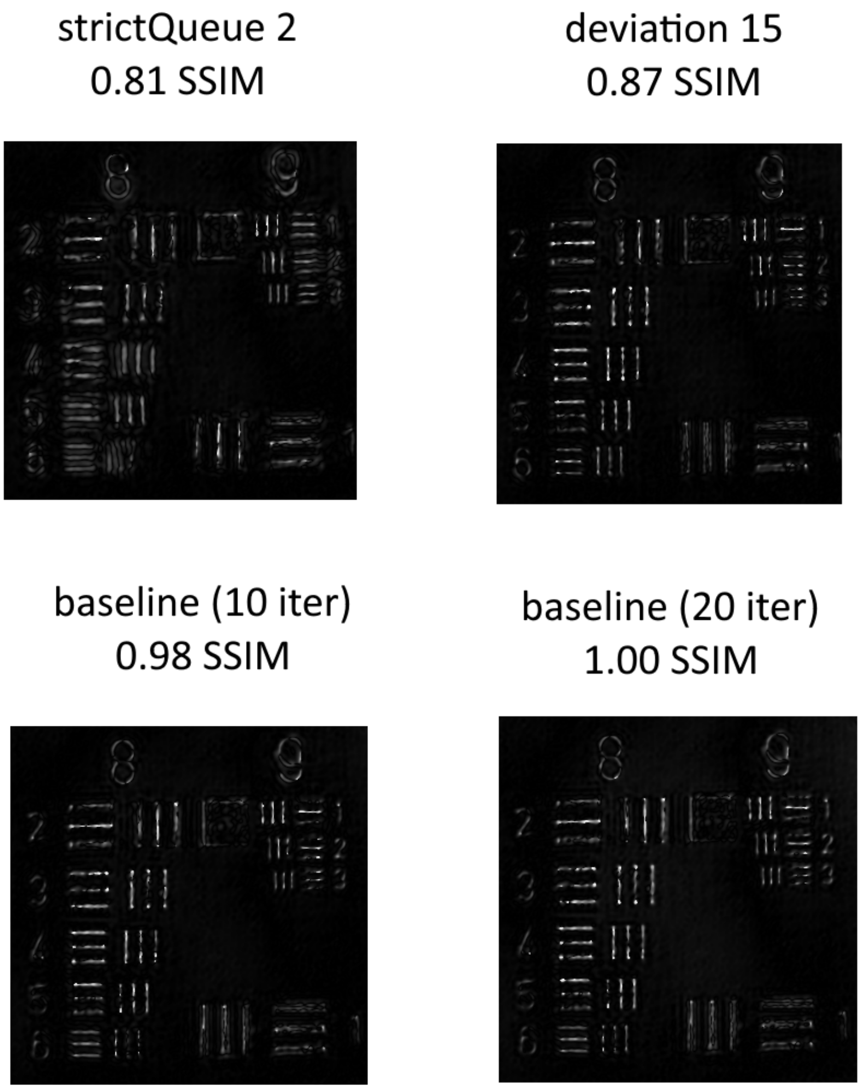


Figure 7.16: Selection of phase results from strictQueue 2 (left) and deviation 15 (right) at 10 iterations. Baseline at 10 and 20 iterations shown below

Chapter 8

Discussion

In this chapter, the experimental results are discussed experiment by experiment, with each discussion relating the current experiment to the previously discussed experiments.

8.1 Baseline

Looking at the baseline results, it is clear that 20 iterations of the algorithm with a single thread produces good results. Both MSE and SSIM show that the amplitude does not converge by much after around 5 iterations. The phase, however, follows a flatter curve that benefits from the fact that a full 20 iterations were performed, though the difference between 10 and 20 iterations is quite small. The USAF resolution target from the Tian labs data is quite clearly resolved down to the smallest target already at 10 iterations, though the smallest phase details are somewhat clearer at 20 than at 10 iterations.

8.2 Multi-threaded

As soon as concurrent execution is introduced, the order of iteration steps changes, as the original sequence contains only neighbouring iteration steps. Unless we permit large overlap between concurrent iteration steps, two neighbouring iteration steps will never be run in parallel. Concurrent steps are run in batches with a number of threads given as algorithm parameter. It is important to note that if there are fewer threads than valid iteration steps in a batch, some threads will stay idle during batch execution. The stricter the queue is with respect to overlap between iteration steps and deviation from the single threaded iteration step order, the fewer iteration steps will be available for processing in a given batch on average. Also, only one pupil update is performed per thread batch, meaning that a higher number of concurrent threads reduces the pupil update frequency. This might be a significant drawback for systems with large aberrations, but seemingly not for the two data sets from the Tian labs microscope used in this thesis.

8.2.1 Strict queue

The initial implementation of the queue ensures no overlapping iteration steps can be executed concurrently. This round of experiments examines what increasing the number of threads does to quality and reconstruction speed. It is clear from the convergence graphs that a higher amount of threads lead to worse results. This is likely due to the fact that finding a large amount of independent iterations will result in iteration steps from many different locations in the dependency graph, which leads to iterations being performed far out of order with respect to the single-threaded version. This effect is most pronounced in the central iteration steps. These steps are both high in intensity, containing a lot of information about the full image, and close together. With strict respect to overlap, very few of these central steps will be performed in parallel. Instead, they are spread across multiple thread batches, leading to a large deviation from single-threaded order in the central iteration steps.

This proportionality between how much the iteration steps are out of order, and how many threads are run per batch is clearly visible in the quality of the reconstructions. Two or four threads per batch does not alter the order by much, and the result converges close to the baseline. When 8 or 16 threads are run, the reconstruction quality is reduced, especially in the phase image. It can be seen on the convergence graphs that in these cases, while the amplitude still seem to converge close to the baseline, the phase does not. It is worth noting that this behaviour is similar in both data sets from the Tian labs microscope.

8.2.2 Less strict queue

The purpose of these experiments were to see what effect reducing the strictness of the dependency graph had on the reconstruction quality. Like with the 100% experiment from Section 8.2.1, the convergence towards baseline end up at much lower similarity and becomes much more erratic when the number of threads are set to 8 or 16. The 2 and 4 thread experiments, on the other hand, maintain the smooth convergence in all cases, and converges close to baseline

In the case of 8 iterations, the reduced overlap strictness seems to improve the image of the dog stomach when looking at **strictQueue 3** alongside **lessStrictQueue 5** and **lessStrictQueue 6** for the dog stomach data. The Tian labs USAF data also shows some improvement, at least in the 50% overlap experiment **lessStrictQueue 6**, which shows a relativly smooth converence in the direction of the baseline, with the exception of the jump between 12 and 13 iterations. The 16 iteration experiments have been run with 100, 75, 50, 25 and 0 percent overlap strictness. However, it does not seem to help, as the phase keeps converging to other optima than the baseline.

The 2 and 4 thread experiments show that the algorithm works well in its basic form with few threads. The focus of the experiments in Section 8.2.3 were thus to investigate what a stricter policy on how much the order of iteration steps can deviate from the single thread order can do to improve the convergence even when many concurrent threads are used.

8.2.3 Deviation tolerance

The main observation from this group of experiments is that enforcing an iteration step order closer to the single threaded baseline does improve quality, and can turn parameter combinations that previously led to ill-behaved convergence into well-behaved convergence. An example is "lessStrictQueue 8" with a phase SSIM ending up at around 0.40 vs "deviation 18" with a phase SSIM relative to baseline of 0.90. Both experiments have 16 threads and "significant overlap" of 50%, but "deviation 18" has 25% deviation tolerance, while "lessStrictQueue 8" has 100% deviation tolerance. "Deviation 17" has 50% deviation tolerance, which also gives a reconstruction with phase SSIM of 0.85. However "deviation 16" with 75% deviation tolerance is ill behaved, and ends up with a phase SSIM of 0.72. This indicates that the iteration step order produced by the priority queue when the deviation tolerance is 75% or higher produces iteration steps that are significantly out of order and greatly reduces reconstruction quality. However, as soon as these are removed, further restrictions in iteration step order have less effect on the reconstruction quality.

One explanation for this behaviour is that reordering iteration steps early in an iteration damages convergence more than reordering later. In the spiral order used as a base in all experiments starts at the center of the lens plane and works its way outwards in an anticlockwise spiral. Iterations early in this sequence use high intensity input images that have a large effect on the reconstruction relative to the low intensity images associated with iterations further from the center of the lens plane. Following the findings in previous works, the iteration step order should go from high intensity images to low intensity images. However, forcing the task-parallel algorithm to run a batch of for instance 8 iterations at once while respecting the overlap between iteration steps will lead to the first thread getting the center iteration, but following threads will get iteration steps much further out from the center than the spiral order suggests. This is because all predecessors of that center iteration step get locked when the first iteration step is marked as processing. This leads to the central, early iteration being separated by many, out of order iteration steps in the created order.

Thus, forcing the priority queue to serve null-operations when an iteration step would be too out of order leads to the central iterations being performed one by one. This increases reconstruction quality. However, as the iteration step order progresses to lower intensity images, the queue allows for concurrent iteration steps that are more out of order to increase available parallelism. The reason this does not reduce reconstruction quality in the same way as reordering early iteration steps most likely has to do with both the lower intensity, and way the spiral order achieves the ideal of an order of descending intensity images.

The intensity of the input images tend to be proportional to the angle of illumination, with direct illumination as the most intense, and illumination at great angles the least. This means that the intensity will be roughly equal for iterations corresponding to pupil shifts of the same magnitude. The spiral order exploits this by first traversing the center, then the LEDs in a square around the center, then the LEDs around that square and so on, until all iteration steps are performed. The estimation inherent in traversing in squares rather than circles does not seem to be significant, as it has been shown in previous work[2] to match the true intensity order well in terms of reconstruction quality. The main observation is: since the objective is to perform iteration steps in intensity order, **iteration steps in the same square can be reordered**. This means that reordering late iterations from large squares of similar intensity should not matter in terms of reconstruction quality. As large squares span a large amount of iteration step sequence indices, late indexes in a spiral sequence can be more freely reordered. As they more seldom overlap, they can also be performed concurrently to a larger degree.

8.3 Consistency

It can actually be seen that even the experiments with 16 threads and no overlap show different convergence between runs. Both the trajectory of convergence and the end results differ. It makes sense that the experiments with many threads and many race conditions in overlapping lens plane pixels differ, but the difference in four runs of "strictQueue 4" are harder to explain. This means that interpreting data from individual experiments must be done with some care, however when running experiments on two data sets, the overall behaviour of both (ill vs well behaved) seems to usually be equal across the two data sets. Execution time seems to be more stable, only varying with a few seconds between equal experiments. Thread utilisation seems constant for a given experimental setup.

8.4 Convergence and quality

Inspecting the selected Tian labs USAF target phase results shows that the smallest resolvable lines and numbers does seem to follow SSIM relative to baseline, but SSIM does not exactly correspond to exactly what numbers and lines are clearly resolved. Clear quality loss is apparent when SSIM gets lower than 80, but even in the case of "strictQueue 3", with a SSIM of 0.65, all lines are still somewhat visible. It is important to keep in mind that these examples are sub-segments of the full experiment result images. Some of the difference in terms of SSIM might stem from low-frequency details or details not visible in the sub-segments. Overall though, it seems using SSIM to compare a baseline and other experiments does give a usable indication of the resolution of the experimental results.

8.5 Running time and utilisation

When looking at the performance results, two things are apparent. The first is that the wall clock time seems to be proportional to the average number of active threads per batch in the algorithm. The second is that the reconstruction quality (represented by the SSIM of the phase image relative to to baseline at 20 iterations), stays within a fairly narrow range across the selected experiments. Slacking the "deviation tolerance" parameter from 25% to 50% is seen to improve the running time at low or no cost to convergence. This holds when comparing "deviation 17" and "deviation 18", as well as when comparing "deviation 14" and "deviation 15" or "deviation 11" and "deviation 10". This can be explained by the principles discussed in Section 8.2.3. As the primary job of the deviation tolerance metric is to hinder large reorderings of early iteration steps, it needs to be set strict enough to deny early large reorderings. Setting the deviation tolerance any stricter, however, primarily reduces available parallelism in later iteration steps, where the reordering matters less. As the metric for "out of order" is relative to the base sequence order of the iteration step under consideration, even a low deviation tolerance will permit some concurrent execution of iteration steps late in the base order. The tipping point for the Tian labs microscope and the Tian labs USAF target data seems to be be between 50% "deviation tolerance" and 75% "deviation tolerance".

When interpreting the speedups, it is important to note that the pupil adjustment, which is an expensive operation, is performed only once per thread batch. If an experiment runs at on average 8 threads per batch, then the pupil is updated 8 times less than a single threaded version. This will further reduce running time, and potentially also quality, compared to a single threaded experiment. Comparing "baseline-1" with "baseline-8" and "baseline-16", it is clear that reducing the pupil update frequency does reduce the running time. As "deviation 17" on average updates the pupil every 11.27 iteration steps, and "baseline-16" updates the pupil every 16 iteration steps, comparing these experiment over-compensates for the reduced computation time associated with the pupil estimation. Still, comparing "baseline-16" and "deviation 17" does reveal a speedup of about 5 times.

8.6 Contributions to state of the art

The current state of the art of FPM reconstruction does contain both methods to exploit data parallelism between independent sub segments of large input images, and the data parallelism available at the iteration step level. However, little work exists on exploiting task parallelism at the iteration step level in FPM, even though there exists multiple parallel solvers at this level for ptychography. This Section will compare the priority queue based solver in this thesis to other parallel solvers, especially in the parallel ptychography literature, and highlight similarities and differences, as well as discuss the applicability of the priority queue parallelisation in a broader context.

8.6.1 Expanding the FPM priority queue

Comparing the priority queue approach with the independent path-approach described in Xiu[38], one advantage of the queue is that maintaining a single recovery object ensures that the phase remains constant across the full recovery object, as recovering independent recovery object segments can lead to arbitrary constant phase shifts. The results presented in Xiu do not include phase images for the FPM reconstructions. Another advantage of the priority queue is that it can control the overall iteration order, while the different paths in the independent-path approach can progress in arbitrarily different orders relative to intensity, and is thus most likely a method that works better in ptychography than FPM. A third advantage of the priority queue is that is handles overlap in a controllable way that preserves a configurable amount of redundancy, while the independent path-algorithm partitions the iteration steps such that the redundancy in the border regions disappears.

Another example of a solver for the phase recovery problem that is parallel at the sub-iteration level has been demonstrated in ptychography using the difference map method as base by Dong[34], it is not entirely parallel at the object update step precisely because the implementation divides the sub-iterations into partitions that are unaware of the overlaps. In the case of ptychography, a priority queue aware of the overlap constraint could dynamically distribute iterations that could be written to the object safely, for both distributed and non distributed implementations. FPM is in this case an even more restrictive case, as it contains iteration step order as a constraint as well as overlap.

Broader applicability

However, while the priority queue based approach as it is described in this thesis requires a shared memory, many of the related methods do not. This allows them to be more easily parallelisable across a distributed system. However, while the priority queue and thus iteration step control is centralised, it does not mean that memory has to be. Using a batch based method such as in this thesis would allow for a synchronisation step where each threads, instead of writing to the shared memory, performs a write to the memories of all other nodes. As the overlap constraint is maintained, this write can be set to be as race condition free as desired.

It does thus seem that the concept of a dynamic, central structure that can accommodate multiple constraints and cost functions to distribute and manage multi threaded solvers of iterative phase recovery has potential both in FPM and ptychography, providing a new tool for parallelising algorithms in this family. The ability of the priority queue to monitor the overall state of the executing algorithm permits cost functions that can respect multiple constraints. In the case of ptychography, it can respect the overlap between iteration steps. In FPM, the addition of a cost function for out-of-order execution improves recovery quality while still allowing exploitation of task parallelism.

The overall structure of the phase retrieval problem in ptychography and FPM, with multiple inputs in different domains that all constrain the solution to be recovered, stems from the origins of the technique, which is the Gerchberg-Saxton algrorithm [45]. This algorithm was further generalised by Fienup[14], and shown to be related (equivalent in the case of a single intensity measurement) to a steepest descent method. What makes it parallelisable in the case of phase retrieval is the fact that the search direction gradient is only computed for a sub region of the recovery object: in the case of FPM, this sub region is defined by the shifted pupil. As long as the search gradient of two different iteration steps do not overlap, they can both be applied to the lens plane estimate at the same time. No strict dependency between iterations arise before one iteration must read or write to an overlapping region in the lens plane. As has been shown in the experiments, when performing phase recovery in FPM, the order of iterations also matters.

However, as the fundamental structure of the alternating projections method underlying both FPM and ptychography is what allows the priority queue method presented in this thesis to work, any other algorithms based on the same principles will be potential candidates for parallelisation using this method. Not all algorithms in this family will be parallelisable however, a counterexample is the algorithm for pupil estimation, where all constraints are used to update the whole of the pupil, and thus are all dependent. To work around it in the context of FPM with pupil recovery, the pupil update algorithm had to be changed to update the pupil less. In our data sets this did not reduce quality by any significant amount, according to the performance experiments. However, this might be a more expensive trade-off when reconstructing an object imaged by a more aberrated imaging system.

Chapter 9

Conclusion and future work

In this thesis, a new task-parallel approach to solving the phase recovery problem in Fourier Ptychography has been presented, developed and validated. It has been shown that task parallelism at the sub-iteration level can be exploited in iterative FPM recovery schemes, albeit within certain constraints. A task parallel FPM implementation must contain a way of distributing work. If this method respects the overlaps between sub-iterations, it is shown to introduce a reordering of the sub-iterations when compared to the single threaded baseline. As the order of iterations are important, this can cause visual artefacts and reduced reconstruction quality. By restricting how far out of order sub-iterations are allowed to be performed, the balance between reconstruction quality and available concurrency can be adjusted. Experiments performed shows that concurrency can be combined with only a small reduction in reconstruction quality.

The fact that it is possible to parallelise the iterative FPM algorithm illustrates that even though the mathematical formulation of an algorithm does not permit concurrency, it might still exist upon more careful inspection of the problem. This concurrency might come at a price in terms of the quality of the output. In some cases, this quality degradation might cause the parallelised algorithm to become unusable. However, by identifying what factors in the concurrent implementation causes result degradation compared to the iterative one, they can be managed and balanced to the point where the attained speedup is more beneficial than the reduced quality. For instance, in the experiments in this thesis, a speedup of 6.65 was attained when running the algorithm with 16 threads rather than 1, and the difference in SSIM relative to baseline between the 16 thread version and the 1 thread version was only 0.04.

When considering iterative algorithms however, this trade-off must be compared to the speedup/quality trade-off associated with running more/less iterations. For instance, to get a speedup of 5, one could also reduce the number of run iterations down by a fifth. However, as the number of iterations in our timing experiments are 10, this would mean running only one or two iterations. Looking at the shape of the convergence graphs, it is clear that the low starting point and initial rapid convergence per iteration makes this trade-off compare unfavourably in this case

Another advantage of identifying new ways of parallelise iterative algorithms at the task level is that it applies to a wider range of computers than just data parallel machines. This thesis has only examined a task-parallelised version of FPM recovery, and the shown speedups at a low computational cost are obtained not on a GPU, but on a multi core CPU. However, this does not mean that the task-parallel implementation doesn't benefit GPUs. Rather, it serves as another source of parallelism that is orthogonal to the more commonly exploited data parallelism. This means that both forms of parallelism can be exploited at the same time. In the case of FPM, the amount of data parallel work that can be offloaded is limited by the size of the segments. With more threads, and more in-flight segments, the amount of available work for the GPU grows, making it more likely that the GPU can be run at capacity.

There are many ways to implement FPM recovery algorithms. There are multiple different iterative methods, imaging system models, aberration correction and denoising schemes. The work load can be divided between compute units in by different strategies, both at segment level and iteration level. This thesis is a contribution to the toolbox of techniques that can be used to implement efficient, high quality FPM reconstruction, and hopefully a step on the path towards a true near real-time or real-time FPM microscope system.

9.1 Future work

A disadvantage of the implementation presented in the thesis is that require a lot of reading and writing into a common buffer (the lens plane estimate). This works well on a single GPU or a single CPU, but if the computations are distributed, the latency of moving memory from the central buffers to the distributed processors might introduce considerable overhead. Finding a way of allocating batches of sub-iterations to different processors such that each batch only needs to read and write to the main buffer once would reduce the memory overhead, and make the algorithm more suited for distributed systems. Also, investigating the role the pupil recovery plays, and the effect of reducing the frequency of pupil updates should be a priority. This has not been a focus in this work, even though all multi threaded experiments reduce the pupil update frequency to increase available parallelism.

Seeing how essential the order of sub-iterations are for the quality of the final reconstruction, an investigation of different recovery step orders, with focus on developing schemes that are parallelisable might yield more efficient scheduling schemes that permit more concurrency at a lower quality cost.

Another future goal is to combine the GPU acceleration techniques demonstrated in Trelands master[4] and the task parallel methods used in this thesis, to benefit from the speedups for both of them. Furthermore, a method of segmentation and recombination of large input images due to coherence requirements and reconstruction quality considerations should be part of the implementation.

As subdividing the main image into segments unlocks task parallelism, an investigation into the parallelism/result quality trade-off in aggressively subdividing the input into a large, task parallel collection of small segments would also be interesting to research. How small can the segments be made before serious edge effects or other problems arise?.

Also, as mentioned in chapter 4, several more inherently parallel formulations of the phase recovery problem exist and are used in Ptychography. Implementing these for FPM and examining ways to exploit the parallelism inherent in them in the context of Fourier ptychography should also be considered.

To help evaluate and validate future work on FPM recovery algorithms, it would be useful to have a benchmark suite with validation experiments using data from multiple microscopes, exact ground truths and multiple quality metrics. As is, any work presenting a new version of FPM recovery must contain its own test suites, making it harder to compare the quality of experiments and the quality of the results. A suite like this could also be a useful reference for different methods both FPM image acquisition and FPM image recovery.

9.1.1 Framework for optimising and parallelising ptychographic reconstructions

To lift method up to a more broadly applicable level, it would be valuable to try and formulate priority queues for the iterations in other ptychographic and Fourier ptychographic algorithms, as well as investigating whether the structure other iterative algorithms based on enforcing multiple constraints in one or more domains like for instance tomography can be mapped and exploited by similar techniques. Taking the sudoku solver described in Schaad [39] as an example, the method for parallelising with the priority queue method looks as follows:

1. Identify the domain we are reconstructing. In the sudoku case, it is a 9 by 9 grid each containing a column with nine spaces.
2. Identify in what domains the constraints are enforced. In FPM this is the lens/Fourier plane and the image planes, in the sudoku solver all constraints are placed in the sudoku domain
3. Now the overlaps between constraint adjustments must found. In the FPM

case this is derivable from the pupil size in the lens plane and the shifts between each iteration step. In the sudoku case, the dependencies are determined by which row/column/3by3 group overlaps.

4. With the overlaps in place, other cost functions related to the desired iteration order can be introduced. In FPM this is the order metric, while the sudoku solver does not have any obvious order requirements. During the implementation process, however, factors that should be included in a cost function might become apparent.
5. With the priority queue set up with overlaps and cost function, the next step is to decide on in what manner threads should be assigned work from it. There are several options:
   - Launch threads in batches with a fixed target number of threads, then synchronise the updates. This is how the FPM implantation in this thesis is set up
   - Make the queue thread safe and set up a pool of worker threads that can check out and check in work items in a distributed fashion. This requires a stricter control of the common recovery object.
   - Divide work items and recovery object across multiple compute units, with synchronisation directed by the centralised priority queue. Care must be taken to load balance the different compute units, and the partition should be such that as little synchronisation as possible is necessary.

   Once all this is in place, the implementation can be performed on any suitable parallel machine or framework.

Bibliography

[1] P. C. Konda, 'Multi-aperture fourier ptychographic microscopy: Development of a high-speed gigapixel coherent computational microscope,' Ph.D. dissertation, University of Glasgow, 2018.

[2] C. Zheng G. Horstmeyer R. & Yang, 'Wide-field, high-resolution fourier ptychographic microscopy,' *Nature Photon*, 2015.

[3] P. C. Konda, L. Loetgering, K. C. Zhou, S. Xu, A. R. Harvey and R. Horstmeyer, 'Fourier ptychography: Current applications and future promises,' *Opt. Express*, 2020.

[4] A. Treland, 'Gpu accelerated fourier ptychography,' M.S. thesis, NTNU, 2020.

[5] J. W. Goodman, *Introduction to Fourier optics*, 3rd edition. Roberts and Company Publishers, 2005.

[6] G. Zheng, *Fourier ptychographic imaging: a MATLAB tutorial*. Morgan & Claypool Publishers, 2016.

[7] L. Tian and L. Waller, '3d intensity and phase imaging from light field measurements in an led array microscope,' *Optica*, vol. 2, no. 2, pp. 104–111, Feb. 2015. [Online]. Available: http://www.osapublishing.org/optica/abstract.cfm?URI=optica-2-2-104.

[8] K. O. Schnell, 'Fourier ptychography for computationally enhanced imaging in two and three dimensions,' M.S. thesis, NTNU, 2019.

[9] J. Chung, 'Computational imaging: A quest for the perfect image,' Ph.D. dissertation, CALIFORNIA INSTITUTE OF TECHNOLOGY, 2019.

[10] A. Pan, C. Zuo and B. Yao, 'High-resolution and large field-of-view fourier ptychographic microscopy and its applications in biomedicine,' *Reports on Progress in Physics*, vol. 83, no. 9, p. 096 101, Aug. 2020. DOI: 10.1088/1361-6633/aba6f0. [Online]. Available: https://doi.org/10.1088/1361-6633/aba6f0.

[11] D. Claus, D. Iliescu and P. Bryanston-Cross, 'Quantitative space-bandwidth product analysis in digital holography,' *Appl. Opt.*, vol. 50, no. 34, H116–H127, Dec. 2011. DOI: 10.1364/AO.50.00H116. [Online]. Available: http://ao.osa.org/abstract.cfm?URI=ao-50-34-H116.

[12] L.-H. Yeh, J. Dong, J. Zhong, L. Tian, M. Chen, G. Tang, M. Soltanolkotabi and L. Waller, 'Experimental robustness of fourier ptychography phase retrieval algorithms,' *Optics express*, vol. 23, no. 26, pp. 33 214–33 240, 2015.

[13] P. Enfedaque, H. Chang, H. Krishnan and S. Marchesini, 'Gpu-based implementation of ptycho-admm for high performance x-ray imaging,' in *Computational Science – ICCS 2018*, Y. Shi, H. Fu, Y. Tian, V. V. Krzhizhanovskaya, M. H. Lees, J. Dongarra and P. M. A. Sloot, Eds., Cham: Springer International Publishing, 2018, pp. 540–553.

[14] J. R. Fienup, 'Phase retrieval algorithms: A comparison,' *Appl. Opt.*, vol. 21, no. 15, pp. 2758–2769, Aug. 1982. DOI: 10.1364/AO.21.002758. [Online]. Available: http://ao.osa.org/abstract.cfm?URI=ao-21-15-2758.

[15] Y. Shechtman, Y. C. Eldar, O. Cohen, H. N. Chapman, J. Miao and M. Segev, 'Phase retrieval with application to optical imaging: A contemporary overview,' *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 87–109, 2015. DOI: 10.1109/MSP.2014.2352673.

[16] L. Tian, X. Li, K. Ramchandran and L. Waller, 'Multiplexed coded illumination for fourier ptychography with an led array microscope,' *Biomed. Opt. Express*, vol. 5, no. 7, pp. 2376–2389, Jul. 2014. DOI: 10.1364/BOE.5.002376. [Online]. Available: http://www.osapublishing.org/boe/abstract.cfm?URI=boe-5-7-2376.

[17] Y. Fan, J. Sun, Q. Chen, M. Wang and C. Zuo, 'Adaptive denoising method for fourier ptychographic microscopy,' *Optics Communications*, vol. 404, pp. 23–31, 2017, Super-resolution Techniques, ISSN: 0030-4018. DOI: https://doi.org/10.1016/j.optcom.2017.05.026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0030401817304054.

[18] N. Hussain, M. Hasanzade, D. W. Breiby and M. N. Akram, 'Wavelet based thresholding for fourier ptychography microscopy,' in *2020 35th International Conference on Image and Vision Computing New Zealand (IVCNZ)*, 2020, pp. 1–6. DOI: 10.1109/IVCNZ51579.2020.9290707.

[19] G. E. Moore, 'Cramming more components onto integrated circuits,' *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[20] S. Borkar and A. A. Chien, 'The future of microprocessors,' *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[21] T. N. Theis and H. .-. P. Wong, 'The end of moore's law: A new beginning for information technology,' *Computing in Science Engineering*, vol. 19, no. 2, pp. 41–50, 2017. DOI: 10.1109/MCSE.2017.29.

[22] M. B. Taylor, 'Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse,' in *DAC Design Automation Conference 2012*, 2012, pp. 1131–1136.

[23] J. L. H.
bibinitperiod D. A. Patterson, *Computer Architecture - a Quantitative Approach*. Morgan Kaufmann, 2019.

[24] (). 'Pthreads(7) — linux manual page,' [Online]. Available: `https://man7.org/linux/man-pages/man7/pthreads.7.html` (visited on 11/08/2021).

[25] G. M. Amdahl, 'Validity of the single processor approach to achieving large scale computing capabilities,' in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485, ISBN: 9781450378956. DOI: `10.1145/1465482.1465560`. [Online]. Available: `https://doi.org/10.1145/1465482.1465560`.

[26] J. L. Gustafson, 'Reevaluating amdahl's law,' *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988, ISSN: 0001-0782. DOI: `10.1145/42411.42415`. [Online]. Available: `https://doi.org/10.1145/42411.42415`.

[27] K. Olukotun and L. Hammond, 'The future of microprocessors,' *Queue*, vol. 3, no. 7, pp. 26–29, Sep. 2005, ISSN: 1542-7730. DOI: `10.1145/1095408.1095418`. [Online]. Available: `http://doi.acm.org/10.1145/1095408.1095418`.

[28] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland and D. Glasco, 'Gpus and the future of parallel computing,' *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011.

[29] j. W. D. Hillis G. L. Steele, 'Data parallel algorithms,' *Communcations of the ACM*, 1986.

[30] Nvidia. (2020). 'Programming guide :: Cuda toolkit documentation,' [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visited on 14/12/2020).

[31] AMD, *"vega" instruction set architecture - reference guide*, https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf, Accessed: 09.11.2019, 2018.

[32] H. Sutter and J. Larus, 'Software and the concurrency revolution,' *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005, ISSN: 1542-7730. DOI: `10.1145/1095408.1095421`. [Online]. Available: `http://doi.acm.org/10.1145/1095408.1095421`.

[33] E. B. and T. P., 'A computational framework for ptychographic reconstructions,' 2016.

[34] Z. Dong, Y.-L. L. Fang, X. Huang, H. Yan, S. Ha, W. Xu, Y. S. Chu, S. I. Campbell and M. Lin, 'High-performance multi-mode ptychography reconstruction on distributed gpus,' in *2018 New York Scientific Data Summit (NYSDS)*, 2018, pp. 1–5. DOI: `10.1109/NYSDS.2018.8538964`.

[35] Y. S. G. Nashed, D. J. Vine, T. Peterka, J. Deng, R. Ross and C. Jacobsen, 'Parallel ptychographic reconstruction,' *Opt. Express*, no. 26, pp. 32 082–32 097, 2014.

[36] S. Marchesini, H. Krishnan, B. J. Daurer, D. A. Shapiro, T. Perciano, J. A. Sethian and F. R. N. C. Maia, '*SHARP*: a distributed GPU-based ptychographic solver,' *Journal of Applied Crystallography*, 2016. DOI: 10.1107/S1600576716008074. [Online]. Available: https://doi.org/10.1107/S1600576716008074.

[37] 'Parallel fourier ptychographic microscopy for high-throughput screening with 96 cameras (96 eyes),' 2019.

[38] X. W. et al, 'A parallel ptychographic iterative engine with a co-start region,' *Journal of Optics*, vol. 22, no. 7, 2020.

[39] J. Schaad, 'Modeling the 8-queens problem and sudoku using an algorithm based on projections onto nonconvex sets,' Ph.D. dissertation, University of British Columbia, 2010. DOI: http://dx.doi.org/10.14288/1.0071292. [Online]. Available: https://open.library.ubc.ca/collections/ubctheses/24/items/1.0071292.

[40] X. Ou, G. Zheng and C. Yang, 'Embedded pupil function recovery for fourier ptychographic microscopy,' *Opt. Express*, no. 5, pp. 4960–4972, 2014. DOI: 10.1364/OE.22.004960. [Online]. Available: http://www.opticsexpress.org/abstract.cfm?URI=oe-22-5-4960.

[41] T. Labs. (). 'Open source | tian labs,' [Online]. Available: http://sites.bu.edu/tianlab/open-source/ (visited on 04/01/2021).

[42] R. G. Lyons, *Understanding digital signal processing, 3/E*. Pearson Education India, 2004.

[43] O. Bunk, M. Dierolf, S. Kynde, I. Johnson, O. Marti and F. Pfeiffer, 'Influence of the overlap parameter on the convergence of the ptychographical iterative engine,' *Ultramicroscopy*, vol. 108, no. 5, pp. 481–487, 2008.

[44] (). 'Scikit-image: Image processing in python - scikit-image,' [Online]. Available: https://scikit-image.org/ (visited on 23/08/2021).

[45] S. Marchesini, Y.-C. Tu and H.-T. Wu, 'Alternating projection, ptychographic imaging and phase synchronization,' *Applied and Computational Harmonic Analysis*, vol. 41, no. 3, pp. 815–851, 2016, ISSN: 1063-5203. DOI: https://doi.org/10.1016/j.acha.2015.06.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1063520315000913.

Appendix A

Appendix A – Code listings

Selected sub-segments of the full implementation are provided in this section. For access to the full code, contact anne dot elster at gmail dot com.

A.1 The priority queue

Code listing A.1: Generating the dependency graph

```
// create the dependency graph
void PriorityQueue::createDependencyGraph() {

    // Decimal pixels
    // lens radius scaled by how much of it we want to take into
    // account when determining overlap
    double lensDiameterPixelsSquared =
        pow((this->params->lensDiameter / this->params->lensPixelSize) *
            significantOverlap
            , 2);
    double curDistanceSquared = 0;

    unsigned int curEdgeListStartIndex = 0;

    unsigned int LEDindexStart = 0;
    unsigned int LEDindexEnd = 0;

    // For each node, check all others for overlap
    for(unsigned int startNode = 0; startNode < this->seqLength; startNode++) {
        this->indexList[startNode] = curEdgeListStartIndex;
        LEDindexStart = baseSequence[startNode];
        for(unsigned int endNode = 0; endNode < this->seqLength; endNode++) {
            // To find LED index: translate from sequence index
            // to image index to LED index
            LEDindexEnd = baseSequence[endNode];

            curDistanceSquared =
            pow(this->offsetsX[LEDindexStart] - this->offsetsX[LEDindexEnd], 2) +
            pow(this->offsetsY[LEDindexStart] - this->offsetsY[LEDindexEnd], 2);

            if(curDistanceSquared < lensDiameterPixelsSquared
            && startNode != endNode) {
                this->neighbourLists.push_back(endNode);
                curEdgeListStartIndex += 1;
            }
        }
    }
}
```

Code listing A.2: Selecting an available node for work

```
// Returns sequence index of first free non-completed
// item in the queue. Returns -1 if none are available
// Locks all dependencies
int PriorityQueue::checkoutItem() {
  // if no valid item is found, return -1 to indicate a noop
  int resultNode = -1;
  double orderMetric = 0;

  // Linear search of the node list
  for(unsigned int searchIndex = 0;
      searchIndex < this->seqLength;
      searchIndex++) {

    // Found an unlocked item
    if(lockList[searchIndex] == 0) {
      orderMetric = getOrderMetric(searchIndex);

      // Only check out if the order metric is within tolerance.
      // Note that a tolerance of 1 makes this if statement
      // always true
      if(orderMetric >= (1 - this->seqDeviationTolerance)) {
            lockDependencies(searchIndex);
            resultNode = searchIndex;

            this->numNotStarted -= 1;
            this->numProcessing += 1;

            break;
      }

      // Exhaust the list, just in case. Keeps looping either until
      // a valid work item is found, or the list is empty
    }
  }


  return resultNode;
}
```

Code listing A.3: Calculate the order metric

```
// For a given node, calculate how out of order processing it now would be
double PriorityQueue::getOrderMetric(unsigned int node) {
  // Count finished up to but not including node
  int numFinished = 0;

  for(int nodeI = 0; nodeI < node; nodeI++) {
    // completeList is 0 for unfinished and 1 for finished
    numFinished += this->completeList[nodeI];
  }

  // First node cannot have any completed predecessors
  if(node == 0) {
    return 1;
  }
  double result = 0;
  result = (double)numFinished / (double)(node);
  return result;
}
```

A.2 Iteration step

Code listing A.4: A full iteration step, without pupil estimate update

```
void performIterationStepCPU(int imgSeqI, iterationInfo *iterInfo,
                             iterationBuffers *workBuffers, double **grids) {

  // First, we must fetch and copy the current part of the high
  // res spectrum. For this we use a helper function that
  // needs the offset at the top left corner, as well as the size
  int LEDindex = iterInfo->iterationSequence[imgSeqI];
  int imageIndex = iterInfo->LEDiToIMGi[LEDindex];


  experimentParams *params = iterInfo->params;

  // Offsets are relative to the center of the lens plane
  // We explisitly round, as the pupilOffset arrays contains deciaml pixel offset
  int cornerX;
  int cornerY;
  cornerX = ceil
    (iterInfo->pupilOffsetsX[LEDindex] -
     (double)(params->segmentWidthLowRes/2));
  cornerY = ceil
    (iterInfo->pupilOffsetsY[LEDindex] -
     (double)(params->segmentHeightLowRes/2));

  // High res segment has same number of pixels across planes
  int lensPlaneCenterX = params->segmentWidthHighRes / 2;
  int lensPlaneCenterY = params->segmentHeightHighRes / 2;

  // These coordinates refer to the pixel position in image pixels
  // (origin in top left corner)
  cornerX += lensPlaneCenterX;
  cornerY += lensPlaneCenterY;

  // Copy the area of intrest into the work buffer estSelection
  copySubBufferSquare(workBuffers->workingEstimate,
                      params->segmentHeightHighRes,
                      params->segmentWidthHighRes,
                      workBuffers->estSelection,
                      params->segmentHeightLowRes,
                      params->segmentWidthLowRes,
                      cornerX, cornerY);
  // With the buffer copied, we need to apply the pupil
  multiplyBuffers(workBuffers->estSelection,
                  workBuffers->pupilEstimate,
                  workBuffers->estSelection,
                  params->segmentHeightLowRes, params->segmentWidthLowRes);


  // Before propagating, we store a copy of the selection to use
  // in the lens plane projection
  copySubBufferSquare(workBuffers->estSelection,
                      params->segmentHeightLowRes,
                      params->segmentWidthLowRes,
                      workBuffers->estSelectionOld,
                      params->segmentHeightLowRes,
                      params->segmentWidthLowRes,
                      0, 0);


  // Then we can propagate it to the detector plane
  propagateField(workBuffers->estSelection,
                 workBuffers->estSelection, grids, LtoD,
                 params->segmentWidthLowRes, params->segmentHeightLowRes,
                 params);


  double *curSegmentPointer =
    iterInfo->allImageSegments +
    imageIndex * params->segmentWidthLowRes * params->segmentHeightLowRes;

  // Once propagated, we apply the first projection. This updates
  // the estimate amplitude to match the amplitude from the image,
  // while leaving phase unchanged

  projectOntoImage(workBuffers->estSelection, curSegmentPointer, params);


  // Then back
  propagateField(workBuffers->estSelection,
                 workBuffers->estSelection, grids, DtoL,
                 params->segmentWidthLowRes, params->segmentHeightLowRes,
                 params);

  // Store the image adjusted pupil
  copySubBufferSquare(workBuffers->estSelection,
                      params->segmentHeightLowRes,
                      params->segmentWidthLowRes,
                      workBuffers->estSelectionImageAdjusted,
                      params->segmentHeightLowRes,
                      params->segmentWidthLowRes,
                      0, 0);


  // Then we use the result to adjust the lens plane estimate
  projectOntoLensPlane(workBuffers->workingEstimate,
                       workBuffers->pupilEstimate,
                       workBuffers->estSelection,
                       workBuffers->estSelectionOld,
                       cornerX, cornerY, params);

}
```

A.3 Propagation and projection

Code listing A.5: Propagate light field between planes

```
// Caller must supply output buffer
// Can be performed in-place
void propagateField(double *planeBufferIn, double* planeBufferOut,
                    double **grids, propDir direction,
                    int width, int height,
                    experimentParams *params) {

  double k0 = params->k0; // Illumination wavenumber (spatial period)
  double dObjectLens = params->dObjectLens;
  int flatIndex = 0;

  double realIn; double imagIn;
  double xOffset; double yOffset;

  // Different final formulas for different planes
  // They are all following the Fresnel model,
  // but some have been simplified following Konda 2018

  if(direction == LtoD) {
    // first we copy the input over to the output
    copySubBufferSquare(planeBufferIn, height, width,
                        planeBufferOut, height, width,
                        0, 0);
    ifft2shift(planeBufferOut, height, width);
    ifft2(planeBufferOut, height, width);
  }

  else if(direction == DtoL) {
    // first we copy the input over to the output
    copySubBufferSquare(planeBufferIn, height, width,
                        planeBufferOut, height, width,
                        0, 0);
    fft2(planeBufferOut, height, width);
    fft2shift(planeBufferOut, height, width);
  }

  else if(direction == OtoL){
    std::cout << "propagate␣from␣object␣plane␣to␣lens␣plane\n";
    // Calculate for each destination pixel
    // L(z,w) = fft{O(x,y) * e^(k/2u * (x^2 + y^2))}


    for(int row = 0; row < height; row++) {
      for(int col = 0; col < width; col++) {
        // For now: perform in rectangular coordinates
        // Exponential function written with eulers identity
        flatIndex = row*width + col;
        realIn = REAL(planeBufferIn, flatIndex);
        imagIn = IMAG(planeBufferIn, flatIndex);

        // The object coordinates are first in the grid array
        xOffset = grids[0][flatIndex];
        yOffset = grids[1][flatIndex];

        // Setup phasor (phase factor)
        // i* k/2u * (x^2 + y^2)
        double phaseFactor =
          (k0 / (2*dObjectLens)) *
          (xOffset*xOffset + yOffset*yOffset);

        double phasorReal = cRectRe(1.0, phaseFactor);
        double phasorImag = cRectIm(1.0, phaseFactor);

        // Then we multiply the phasor(magnitude = 1)
        double resultReal = cMultRe(realIn, imagIn,
                                    phasorReal, phasorImag);
        double resultImag = cMultIm(realIn, imagIn,
                                    phasorReal, phasorImag);

        REAL(planeBufferOut, flatIndex) = resultReal;
        IMAG(planeBufferOut, flatIndex) = resultImag;

      }
    }
    // Then a fourier transform
    fft2(planeBufferOut, height, width);


    // And a shift
    fft2shift(planeBufferOut, height, width);

  }

  else if(direction == LtoO) {

    // first we copy the input over to the output
    copySubBufferSquare(planeBufferIn, height, width,
                        planeBufferOut, height, width,
                        0, 0);

    // Then we can ifftshift the output to make it reverse
    // Fourier transformable
    ifft2shift(planeBufferOut, height, width);
    ifft2(planeBufferOut, height, width);

    //TODO phase looks weird in end result.
    // Comment out until resolved

    // Then we multiply the inverse of the initial phase factor
    for(int row = 0; row < height; row++) {
      for(int col = 0; col < width; col++) {
        // For now: perform in rectangular coordinates
        // Exponential function written with eulers identity
        flatIndex = row*width + col;
        realIn = REAL(planeBufferOut, flatIndex);
        imagIn = IMAG(planeBufferOut, flatIndex);

        // The object coordinates are first in the grid array
        xOffset = grids[0][flatIndex];
        yOffset = grids[1][flatIndex];
        // Setup phasor (phase factor)
        // -i * k/2u * (x^2 + y^2)
        double phaseFactor =
          -1 *
          (k0 / (2*dObjectLens)) *
          (xOffset*xOffset + yOffset*yOffset);
        // DEBUG: testing phaseFactor
        phaseFactor *= 1;

        double phasorReal = cRectRe(1.0, phaseFactor);
        double phasorImag = cRectIm(1.0, phaseFactor);

        // Then we multiply the phasor(magnitude = 1)
        double resultReal = cMultRe(realIn, imagIn,
                                    phasorReal, phasorImag);
        double resultImag = cMultIm(realIn, imagIn,
                                    phasorReal, phasorImag);

        REAL(planeBufferOut, flatIndex) = resultReal;
        IMAG(planeBufferOut, flatIndex) = resultImag;

      }
    }
  }
}
```

Code listing A.6: Projecting the current estimate onto the space of solutions consistent with a given input image

```
// Store results back into buffer "estimate"
void projectOntoImage(double *estimate, double *measurement,
                      experimentParams *params) {

  unsigned int rows = params->segmentHeightLowRes;
  unsigned int cols = params->segmentWidthLowRes;
  unsigned int flatIndex = 0;

  // Use the standard atan convention: -pi < angle < pi
  double estimatePhase = 0;
  double estimateAmp = 0; // for debugging
  double measuredAmp = 0;

  double newRe = 0;
  double newIm = 0;

  // NOTE: measurement is real, estimate is complex
  for(int row = 0; row < rows; row++) {
    for(int col = 0; col < cols; col++) {
      flatIndex = row*cols + col;

      // Find phase and amp from estimate and recorded low
      // res image respectively
      estimatePhase =
        cArg(REAL(estimate, flatIndex), IMAG(estimate, flatIndex));
      estimateAmp =
        cMag(REAL(estimate, flatIndex), IMAG(estimate, flatIndex));
      measuredAmp = measurement[flatIndex];

      // Energy adjustment
      // (due to difference in scaling constant size between
      // low resolution segment and high resolution estimate:
      // all coefficients have been scaled by upscaleFactor^2
      // more in the high resolution transform than the low
      // resolution one)
      measuredAmp *= (params->upscaleFactor *
                      params->upscaleFactor);

      newRe = cRectRe(measuredAmp, estimatePhase);
      newIm = cRectIm(measuredAmp, estimatePhase);

      // Translate back into rectangular coordinates and store
      REAL(estimate, flatIndex) = newRe;
      IMAG(estimate, flatIndex) = newIm;
    }
  }
}
```

Code listing A.7: Uses the image projection to create a search direction for the lens plane estimate. Several methods for calculating step length is provided, but the quasi Gauss-Newton is the one used in this thesis

```
// Stores result in the newEstimate buffer
// Makes no assumptions about pupil size,
// iterates over all pixels
void projectOntoLensPlane(double *workingEstimate, double *pupilEstimate,
                          double *curSelection, double *curSelectionOld,
                          int cornerX, int cornerY, experimentParams *params) {

  unsigned int rows = params->segmentHeightLowRes;
  unsigned int cols = params->segmentWidthLowRes;
  unsigned int flatIndex = 0;

  // Simplest scheme, just replace the old segment with the new
  // only guarantees local convergence, but hey, it’s something
  if(params->lensProjectionMethod == PROJECTION_COPY) {
    // Simply pupil filter the result of the projection
    multiplyBuffers(curSelection, pupilEstimate, curSelection,
                    params->segmentHeightLowRes, params->segmentWidthLowRes);

  }

  // GSF. If pupil is non-ideal, this will provide a better step size (?)
  // If the pupil is ideal, this reduces to PROJECTION_COPY
  else if(params->lensProjectionMethod == PROJECTION_GSF) {
    // O_upd = O_old + (conj(P) / max(abs(P))^2) * (O_img - O_old)

    double pupilAbsMax = findAbsMaxBuffer(pupilEstimate, rows, cols);
    double pupilAbsMaxSquareInverse = 1/(pupilAbsMax * pupilAbsMax);
    double diffRe = 0;
    double diffIm = 0;

    double pupilConjRe = 0;
    double pupilConjIm = 0;
    double scaledDiffRe = 0;
    double scaledDiffIm = 0;

    // Go pixel by pixel
    for(int row = 0; row < rows; row++) {
      for(int col = 0; col < cols; col++) {
            flatIndex = row*cols + col;

            //Difference between image-projected estimate and old estimate
            // (O_img - O_old). This is the direction vector of the recovery
        diffRe =
          REAL(curSelection, flatIndex) -
          REAL(curSelectionOld, flatIndex);
        diffIm =
          IMAG(curSelection, flatIndex) -
          IMAG(curSelectionOld, flatIndex);

        // Conjucate of pupil for scaling
        // conj(P)
        pupilConjRe = REAL(pupilEstimate, flatIndex);
        pupilConjIm = -IMAG(pupilEstimate, flatIndex);

        // Scale difference by conjugate pupil
        // conj(P) * (O_img - O_old)
        scaledDiffRe = cMultRe(pupilConjRe, pupilConjIm, diffRe, diffIm);
        scaledDiffIm = cMultIm(pupilConjRe, pupilConjIm, diffRe, diffIm);

        // Scale further by inverse of squared absolute pupil
        // (conj(P) / max(abs(P))^2) * (O_img - O_old)
        scaledDiffRe *= pupilAbsMaxSquareInverse;
        scaledDiffIm *= pupilAbsMaxSquareInverse;

        // Write updated version
        // O_upd = O_old + (conj(P) / max(abs(P))^2) * (O_img - O_old)
        REAL(curSelection, flatIndex) =
          REAL(curSelectionOld, flatIndex) + scaledDiffRe;
        IMAG(curSelection, flatIndex) =
          IMAG(curSelectionOld, flatIndex) + scaledDiffIm;
      }
    }
  }

  // Estimation of gauss newton method for non-linear optimisation
  // Good balance between computation time and reconstruction quality,
  // robust to noise and system errors
  else if(params->lensProjectionMethod == PROJECTION_GAUSS_NEWTON) {
    // O_new = O_old +
    //    alpha * (abs(P)/max(abs(P))) * conj(P) * 1/(abs(P)^2 + delta) *
    //    (O_img - O_old)

    double pupilAbsMax = findAbsMaxBuffer(pupilEstimate, rows, cols);
    double pupilAbsCur = 0;
    double pupilConjRe = 0;
    double pupilConjIm = 0;

    double diffRe = 0;
    double diffIm = 0;

    double scaledDiffRe = 0;
    double scaledDiffIm = 0;
    for(int row = 0; row < rows; row++) {
      for(int col = 0; col < cols; col++) {
        flatIndex = row*cols + col;

        // Difference between image-projected estimate and old estimate
        // (O_img - O_old)
        diffRe =
          REAL(curSelection, flatIndex) - REAL(curSelectionOld, flatIndex);
        diffIm =
          IMAG(curSelection, flatIndex) - IMAG(curSelectionOld, flatIndex);

        // Abs(P)
        pupilAbsCur = cMag(REAL(pupilEstimate, flatIndex),
                           IMAG(pupilEstimate, flatIndex));

        // Pupil conjugate at this pixel
        // conj(P)
        pupilConjRe = REAL(pupilEstimate, flatIndex);
        pupilConjIm = -IMAG(pupilEstimate, flatIndex);



        // Scale difference by conjugate pupil
        // conj(P) * (O_img - O_old)
        scaledDiffRe = cMultRe(pupilConjRe, pupilConjIm, diffRe, diffIm);
        scaledDiffIm = cMultIm(pupilConjRe, pupilConjIm, diffRe, diffIm);

        // Scale difference by alpha
        scaledDiffRe *= params->gnStepFactor;
        scaledDiffIm *= params->gnStepFactor;

        // Scale by abs(P)/max(abs(P))
        scaledDiffRe *= (pupilAbsCur/pupilAbsMax);
        scaledDiffIm *= (pupilAbsCur/pupilAbsMax);

        // Scale by 1/(abs(P)^2 + delta)
        scaledDiffRe *= 1/(pupilAbsCur*pupilAbsCur + params->gnNumStabilityConst);
        scaledDiffIm *= 1/(pupilAbsCur*pupilAbsCur + params->gnNumStabilityConst);


        // Write updated version
        REAL(curSelection, flatIndex) =
          REAL(curSelectionOld, flatIndex) + scaledDiffRe;
        IMAG(curSelection, flatIndex) =
          IMAG(curSelectionOld, flatIndex) + scaledDiffIm;
      }
    }
  }

  /*
  // Make sure it's properly pupil filtered
  multiplyBuffers(curSelection, pupilEstimate, curSelection,
                  params->segmentHeightLowRes, params->segmentWidthLowRes);
  */

  // After projection, replace old section with new
  // Subract the old selection
  subtractSubBufferSquare(curSelectionOld,
                          params->segmentHeightLowRes,
                          params->segmentWidthLowRes,
                          workingEstimate,
                          params->segmentHeightHighRes,
                          params->segmentWidthHighRes,
                          cornerX, cornerY);


  // Add the new
  addSubBufferSquare(curSelection,
                     params->segmentHeightLowRes,
                     params->segmentWidthLowRes,
                     workingEstimate,
                     params->segmentHeightHighRes,
                     params->segmentWidthHighRes,
                     cornerX, cornerY);

}
```

Code listing A.8: Uses the image projection to create a search direction for the pupil plane estimate, and a quasi Gauss-Newton method for step length

```
void updatePupilEstimate(double *pupilEstimate,
                         double *workingEstimate,
                         double *curSelectionOld,
                         double *curSelectionImageAdjusted,
                         experimentParams *params) {
  int rows = params->segmentHeightLowRes;
  int cols = params->segmentWidthLowRes;
  unsigned int flatIndex = 0;

  double alpha = params->gnStepFactor;
  double delta = params->gnNumStabilityConst;

  // Direction of recovery
  double dirVecRe = 0; double dirVecIm = 0;

  // Scaling factors from lens plane function
  double lensAbs = 0;
  double lensAbsMax =
    findAbsMaxBuffer(workingEstimate,
                     params->segmentHeightHighRes,
                     params->segmentWidthHighRes);

  // abs(L) / max(abs(L))
  double lensAbsOverMax = 0;
  // 1/(abs(P)^2 + delta)
  double inverseLensSquarePlusDelta = 0;

  double lensConjRe = 0;
  double lensConjIm = 0;

  for(int row = 0; row < rows; row++) {
    for(int col = 0; col < cols; col++) {
      flatIndex = row*cols + col;

      // Setup direction vector
      dirVecRe =
        REAL(curSelectionImageAdjusted, flatIndex) -
        REAL(curSelectionOld, flatIndex);
      dirVecIm =
        IMAG(curSelectionImageAdjusted, flatIndex) -
        IMAG(curSelectionOld, flatIndex);

      // Mulitply the difference by the lens plane conjugate
      lensConjRe = REAL(workingEstimate, flatIndex);
      lensConjIm = -IMAG(workingEstimate, flatIndex);

      dirVecRe = cMultRe(dirVecRe, dirVecIm, lensConjRe, lensConjIm);
      dirVecIm = cMultIm(dirVecRe, dirVecIm, lensConjRe, lensConjIm);

      // Then the scalars
      dirVecRe *= alpha;
      dirVecIm *= alpha;

      lensAbs = cMag(REAL(workingEstimate, flatIndex),
                     IMAG(workingEstimate, flatIndex));

      lensAbsOverMax = lensAbs/lensAbsMax;
      dirVecRe *= lensAbsOverMax;
      dirVecIm *= lensAbsOverMax;

      inverseLensSquarePlusDelta = 1/(lensAbs*lensAbs + delta);
      dirVecRe *= inverseLensSquarePlusDelta;
      dirVecIm *= inverseLensSquarePlusDelta;

      // Write result
      REAL(pupilEstimate, flatIndex) =
         REAL(pupilEstimate, flatIndex) + dirVecRe;
      IMAG(pupilEstimate, flatIndex) =
        IMAG(pupilEstimate, flatIndex) + dirVecIm;
    }
  }
}
```

A.4 Setup and experiments

Code listing A.9: Configuration file for the microscope used in the deviationTolerance 13 experiment

```
# Microscope configuration for the Tian labs dataset
LEDgap: 0.004

numLEDx: 32
numLEDy: 32

LEDcenterX: 14
LEDcenterY: 13

# All LEDs in a circle WITHIN a diameter 19 LEDs are lit
diaLED: 19


illLambda: 0.0000006292
magnification: 8.1485
NA: 0.1


# Meters
dLEDobject: 0.06753
dObjectLens: 0.0422

focalLength: 0.045

# camera pixels, 6.5 micrometers
detectorPixelSize: 0.0000065;
```

Code listing A.10: Configuration file for the algorithm settings used in the deviationTolerance 13 experiment

```
outputDir: experiments/deviationTolerance/USAF/13/images
inputDir: ../img/tianUSAF

imageFlipped: true
# perform model fit test during execution
meassureConvergence: true
# save the image for each iteration
saveAllIterations: true

resultAmpScaleFactor: 200

inputFullWidth: 2560
inputFullHeight: 2160

segmentWidthLowRes: 512
segmentHeightLowRes: 512

upscaleFactor: 4

# Place segment in center
segmentXstart: 800
segmentYstart: 824

bgNoiseThreshold: 1600
gnNumStabilityConst: 1
gnStepFactor: 0.1

lensProjectionMethod: PROJECTION_GAUSS_NEWTON
baseIterationOrder: SPIRAL_ANTICLOCKWISE

significantOverlap: 0.75
seqDeviationTolerance: 0.75
numThreadsCPU: 16
numThreadsGPU: 0
numIter: 20
```

Code listing A.11: Master script for the deviationTolerance experiment suite

```
echo "Run␣the␣deviationTolerance␣experiments"
echo "Working␣directory" $(pwd)
echo "Running␣tian␣USAF␣target␣data"
NUM_EXP=18
echo "Running" $NUM_EXP "experiments␣over␣2␣datasets"
for i in $(seq 1 $(($NUM_EXP)))
do
    CUR_DIR=./experiments/deviationTolerance/USAF/$i
    echo "Running␣recovery..."
    $CUR_DIR/runExperiment.sh

    echo "Calculating␣convergence..."
    ./utilScripts/calcConvergence.sh $CUR_DIR/images \
                                     experiments/baseline/images \
                                     20 \
                                     > $CUR_DIR/convergence.txt

    echo "Generating␣plots..."
    python3 ./utilScripts/plotConvergence.py \
            $CUR_DIR/convergence.txt \
            $CUR_DIR/convPlot.png
done

echo "running␣tian␣dog␣stomach␣data"
for i in $(seq 1 $(($NUM_EXP)))
do
    CUR_DIR=./experiments/deviationTolerance/dog/$i
    echo "Running␣recovery..."
    $CUR_DIR/runExperiment.sh

    echo "Calculating␣convergence..."
    ./utilScripts/calcConvergence.sh $CUR_DIR/images \
                                     experiments/baselineDog/images \
                                     20 \
                                     > $CUR_DIR/convergence.txt

    echo "Generating␣plots..."
    python3 ./utilScripts/plotConvergence.py \
            $CUR_DIR/convergence.txt \
            $CUR_DIR/convPlot.png
done

echo "Experiment␣deviationTolerance␣complete!"
```

A.5 FFTshifts

Code listing A.12: FFTshift, shifts positive frequencies above or at nyquist frequency to closest negative alias

```c
void fft2shift(double* data, int numRows, int numCols) {
  // first reshuffle rows, then cols.
  // inital implementaion: row by row, col by col, with buffer

  // Swap columns
  double* rowBuffer = (double*) malloc(sizeof(double)*2 * numCols);
  double* colBuffer = (double*) malloc(sizeof(double)*2 * numRows);
  int row, col, rowOffset, colOffset;

  int realFreqsRow = floor(numCols/2) + (numCols%2);
  int negFreqsRow = numCols - realFreqsRow;
  int realFreqsCol = floor(numRows/2) + (numRows%2);
  int negFreqsCol = numRows - realFreqsCol;

  // Swap columns, row by row
  for(row = 0; row < numRows; row++) {
    rowOffset = row*numCols;

    for(col = 0; col < negFreqsRow; col++) { // The negative freqs in front
      REAL(rowBuffer, col) = REAL(data+(rowOffset*2), (col+realFreqsRow));
      IMAG(rowBuffer, col) = IMAG(data+(rowOffset*2), (col+realFreqsRow));
    }
    for(col = negFreqsRow; col < numCols; col++) { // Then constant + positive
      REAL(rowBuffer, col) = REAL(data+(rowOffset*2), (col-negFreqsRow));
      IMAG(rowBuffer, col) = IMAG(data+(rowOffset*2), (col-negFreqsRow));
    }
    for(col = 0; col < numCols; col++) {// Copy back. Inefficient?
      REAL(data+(rowOffset*2), col) = REAL(rowBuffer, col);
      IMAG(data+(rowOffset*2), col) = IMAG(rowBuffer, col);
    }
  }



  // Swap rows, column by colums
  for(col = 0; col < numCols; col++) {
    for(row = 0; row < negFreqsCol; row++) { // The negative freqs in front
      colOffset = col;
      REAL(colBuffer, row) =
        REAL(data+(colOffset*2), ((row + realFreqsCol) * numCols));
      IMAG(colBuffer, row) =
        IMAG(data+(colOffset*2), ((row + realFreqsCol) * numCols));
    }
    for(row = negFreqsCol; row < numRows; row++) { // Constant + positive follows
      REAL(colBuffer, row) =
        REAL(data+(colOffset*2), ((row - negFreqsCol) * numCols));
      IMAG(colBuffer, row) =
        IMAG(data+(colOffset*2), ((row - negFreqsCol) * numCols));
    }
    for(row = 0; row < numRows; row++) {// Copy back. Inefficient?
      REAL(data+(colOffset*2), row*numCols) = REAL(colBuffer, row);
      IMAG(data+(colOffset*2), row*numCols) = IMAG(colBuffer, row);
    }
  }
  free(rowBuffer);
  free(colBuffer);
}
```

Code listing A.13: inverse FFTshift, shifts negative frequencies to closest positive aliases

```
void ifft2shift(double* data, int numRows, int numCols) {
    // first reshuffle rows, then cols.
  // inital implementaion: row by row, col by col, with buffer

  // Swap columns
  double* rowBuffer = (double*) malloc(sizeof(double)*2 * numCols);
  double* colBuffer = (double*) malloc(sizeof(double)*2 * numRows);
  int row, col, rowOffset, colOffset;

  int realFreqsRow = floor(numCols/2) + (numCols%2);
  int negFreqsRow = numCols - realFreqsRow;
  int realFreqsCol = floor(numRows/2) + (numRows%2);
  int negFreqsCol = numRows - realFreqsCol;

  // Swap columns, row by row
  for(row = 0; row < numRows; row++) {
    rowOffset = row*numCols;

    for(col = 0; col < realFreqsRow; col++) { // put the positive to the front
      REAL(rowBuffer, col) = REAL(data+(rowOffset*2), (col+negFreqsRow));
      IMAG(rowBuffer, col) = IMAG(data+(rowOffset*2), (col+negFreqsRow));
    }
    // Then then the negative to the back
    for(col = realFreqsRow; col < numCols; col++) {
      REAL(rowBuffer, col) = REAL(data+(rowOffset*2), (col-realFreqsRow));
      IMAG(rowBuffer, col) = IMAG(data+(rowOffset*2), (col-realFreqsRow));
    }
    for(col = 0; col < numCols; col++) {// Copy back. Inefficient?
      REAL(data+(rowOffset*2), col) = REAL(rowBuffer, col);
      IMAG(data+(rowOffset*2), col) = IMAG(rowBuffer, col);
    }
  }



  // Swap rows, column by colums
  for(col = 0; col < numCols; col++) {
    // Positive back in front
    for(row = 0; row < realFreqsCol; row++) {
      colOffset = col;
      REAL(colBuffer, row) =
        REAL(data+(colOffset*2), ((row + negFreqsCol) * numCols));
      IMAG(colBuffer, row) =
        IMAG(data+(colOffset*2), ((row + negFreqsCol) * numCols));
    }
    // Negative to the back
    for(row = realFreqsCol; row < numRows; row++) {
      REAL(colBuffer, row) =
        REAL(data+(colOffset*2), ((row - realFreqsCol) * numCols));
      IMAG(colBuffer, row) =
        IMAG(data+(colOffset*2), ((row - realFreqsCol) * numCols));
    }
    for(row = 0; row < numRows; row++) {// Copy back. Inefficient?
      REAL(data+(colOffset*2), row*numCols) = REAL(colBuffer, row);
      IMAG(data+(colOffset*2), row*numCols) = IMAG(colBuffer, row);
    }
  }

  free(rowBuffer);
  free(colBuffer);
}
```

Appendix B

Appendix B – Convergence data

This appendix contains the full set of convergence graphs from the "lessStrictQueue" experiments and the "deviation" experiments.

B.1 "lessStrictQueue" experiment suite results

The graphs presented in this sections shows convergence for the relaxedQueue experiments. The colors in the plots are read as follows: green is SSIM of phase image, orange is SSIM of amplitude image, blue is (1 - MSE) of amplitude image. All the image metric plots are relative to the baseline in Section 6.6.1 of the relevant data set. Figure B.1, figure B.2 and Figure B.3 contain data from both USAF and dog stomach data sets. Figure B.4 contains only dog data, Figure B.5 contains only USAF.

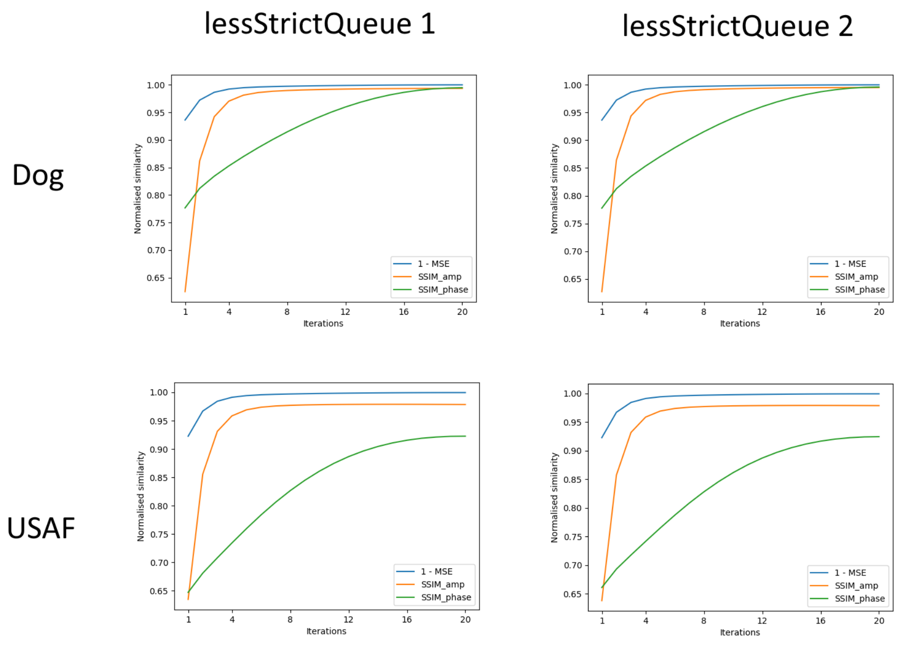


Figure B.1: Convergence of lessStrictQueue 1 and 2 for the two data sets

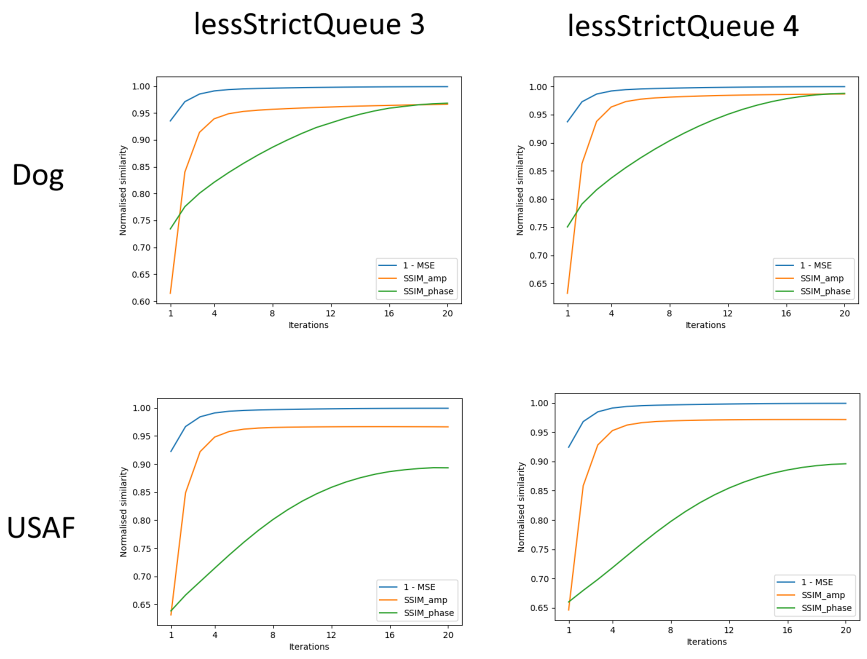


Figure B.2: Convergence of lessStrictQueue 3 and 4 for the two data sets

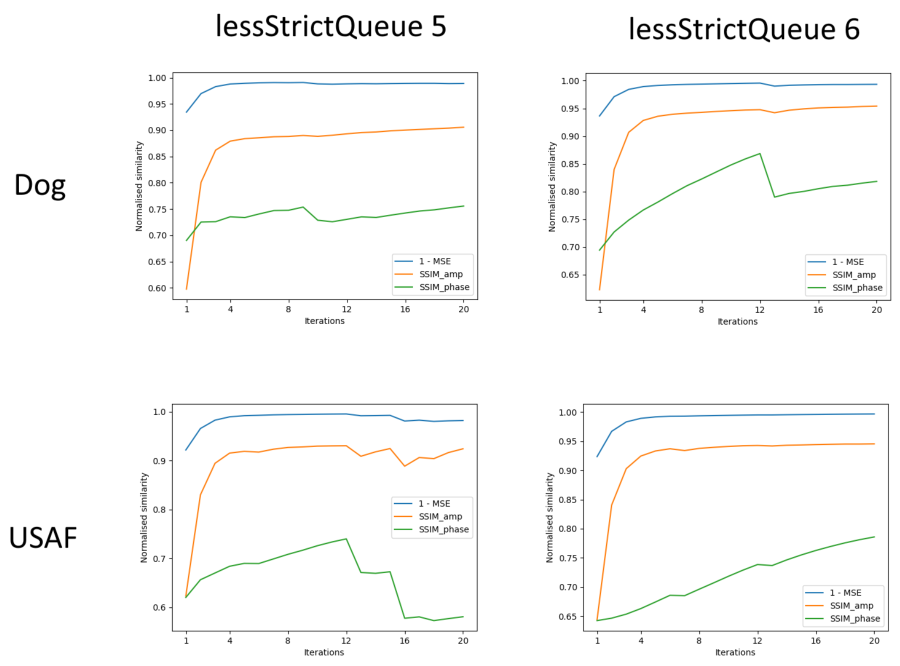


Figure B.3: Convergence of lessStrictQueue 5 and 6 for the two data sets

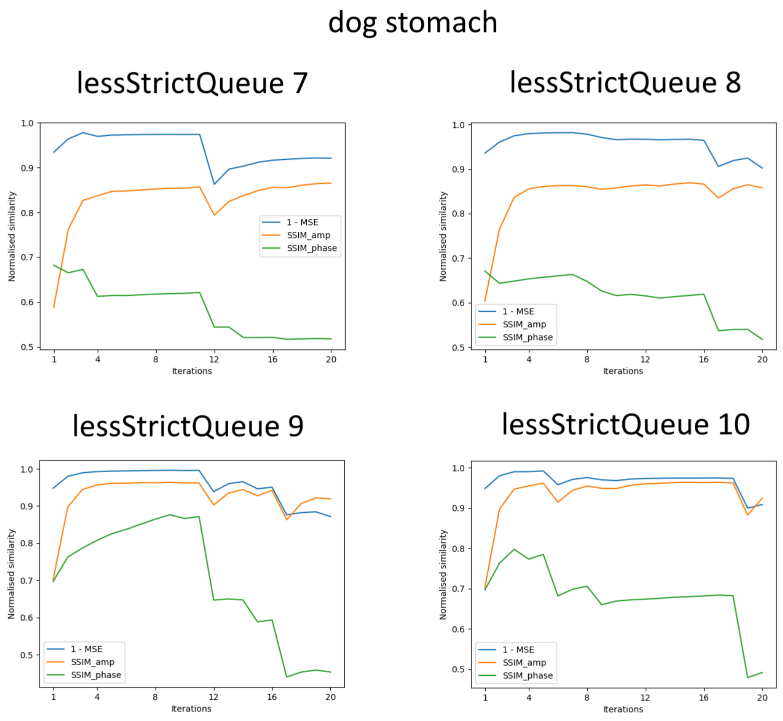


Figure B.4: Convergence of lessStrictQueue 7 through 10 for the dog stomach data

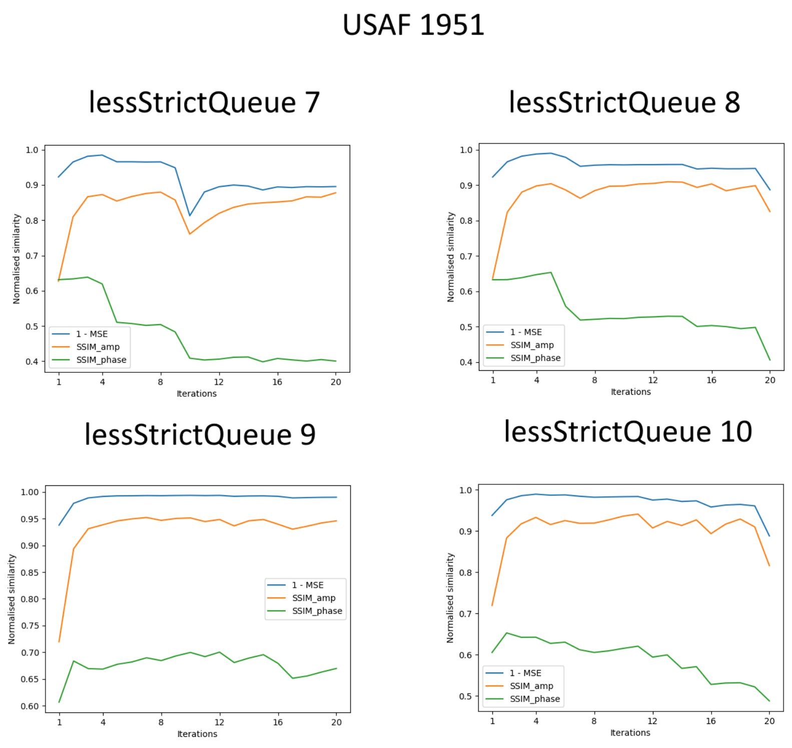
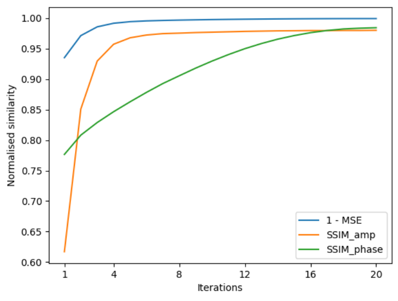


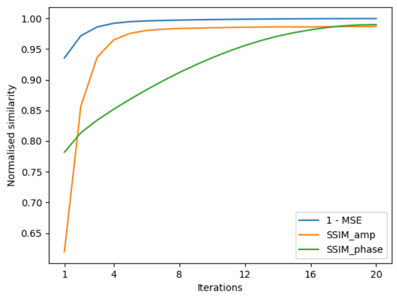
Figure B.5: Convergence of lessStrictQueue 7 through 10 for the Tian USAF target

B.2 "deviation" experiment suite results

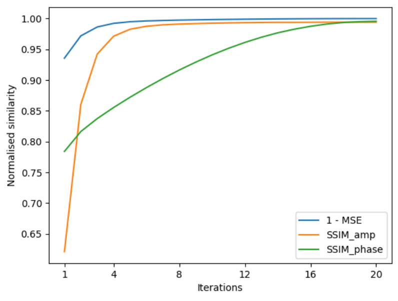
This section contains the convergence graphs of the deviationTolerance experiments. Figures B.6-B.10 contain the convergence plots for the dog stomach data set. Figures B.11-B.15 contain the convergence plots for the USAF data set. The colors in the plots are read as follows: green is SSIM of phase image, orange is SSIM of amplitude image, blue is (1 - MSE) of amplitude image. All the plotted image quality metrics are relative to the relevant baseline in Section 6.6.1.



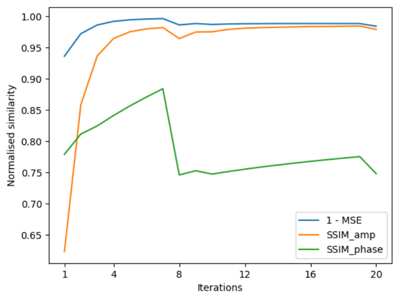
(a) deviation 1



(b) deviation 2

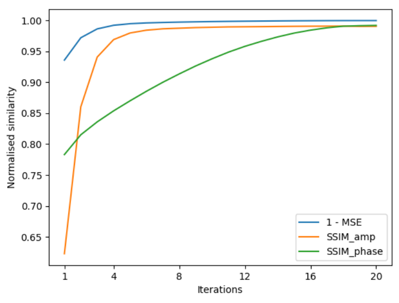


(c) deviation 3

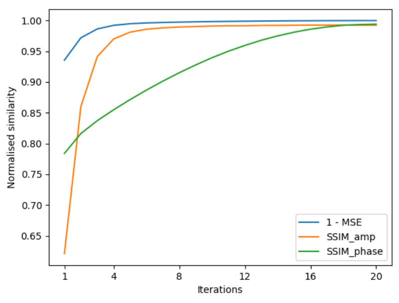


(d) deviation 4

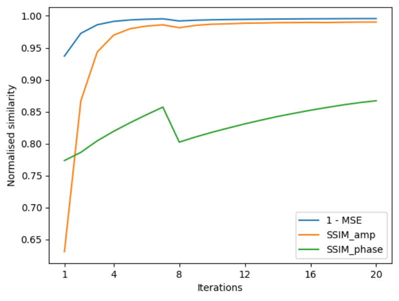
Figure B.6: Convergence of deviation 1-4 on the dog stomach data set



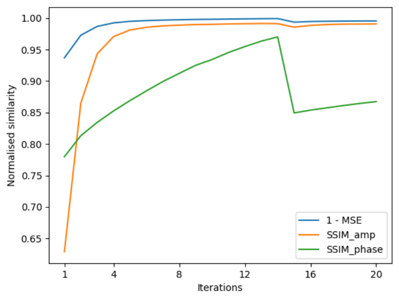
(a) deviation 5



(b) deviation 6

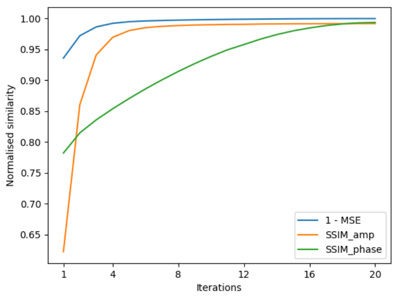


(c) deviation 7

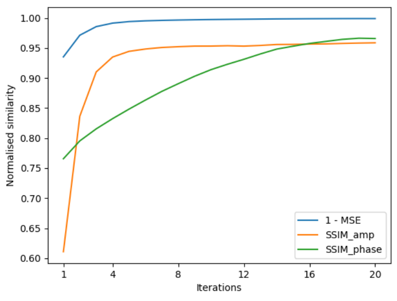


(d) deviation 8

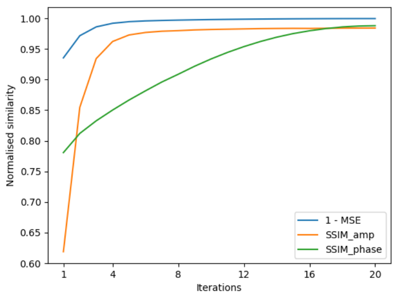
Figure B.7: Convergence of deviation 5-8 on the dog stomach data set



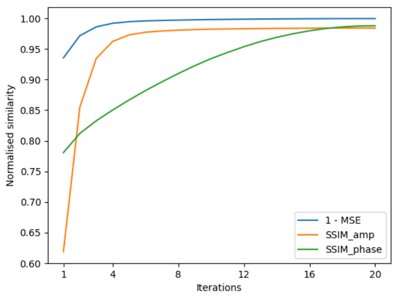
(a) deviation 9



(b) deviation 10

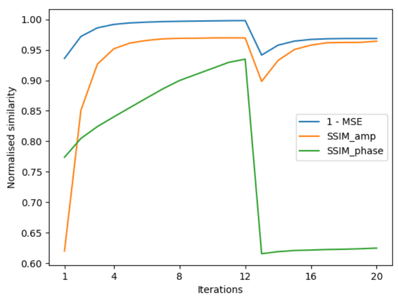


(c) deviation 11

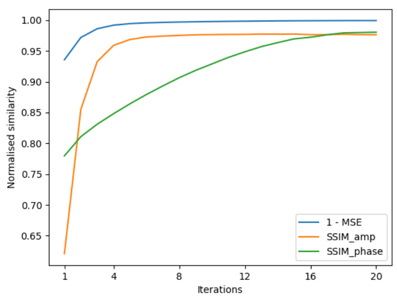


(d) deviation 12

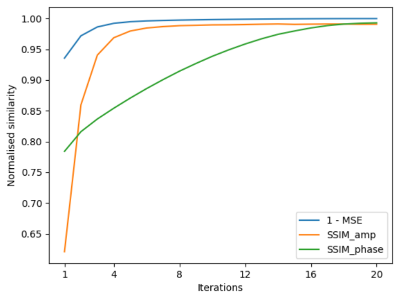
Figure B.8: Convergence of deviation 9-12 on the dog stomach data set



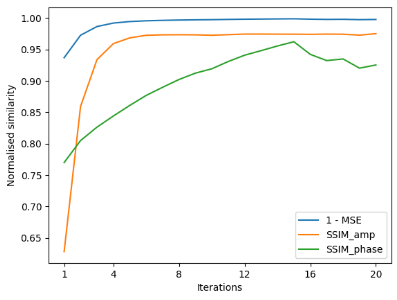
(a) deviation 13



(b) deviation 14

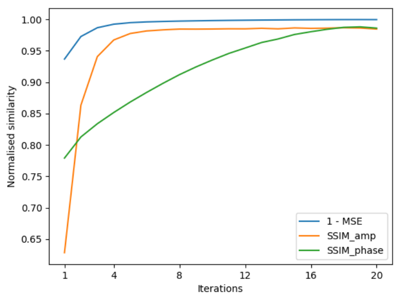


(c) deviation 15

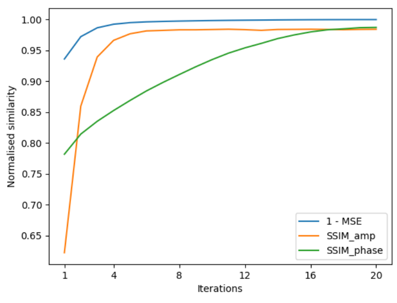


(d) deviation 16

Figure B.9: Convergence of deviation 13-16 on the dog stomach data set

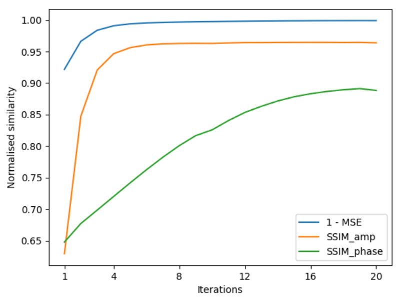


(a) deviation 17

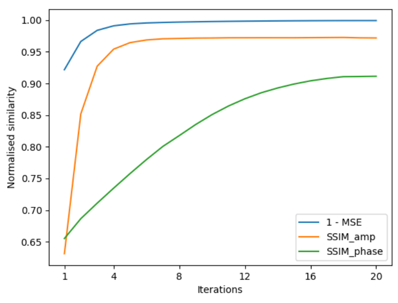


(b) deviation 18

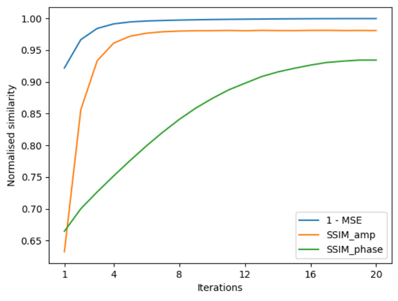
Figure B.10: Convergence of deviation 17-18 on the dog stomach data set



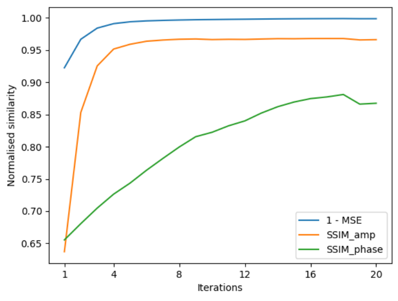
(a) deviation 1



(b) deviation 2

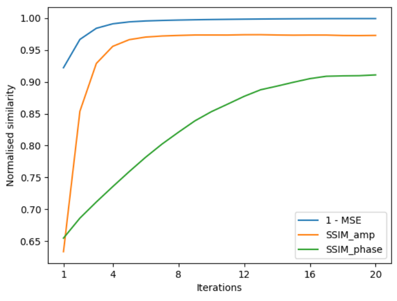


(c) deviation 3

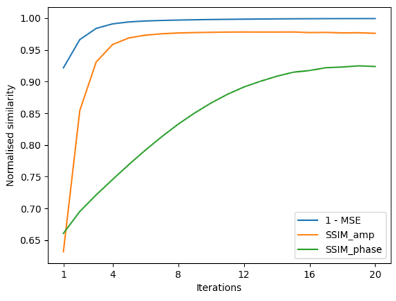


(d) deviation 4

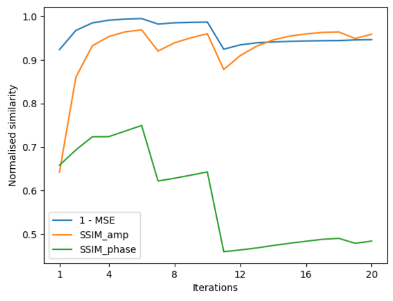
Figure B.11: Convergence of deviation 1-4 on the USAF data set



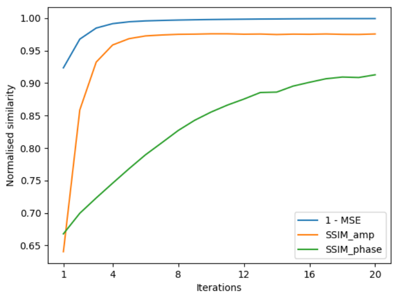
(a) deviation 5



(b) deviation 6

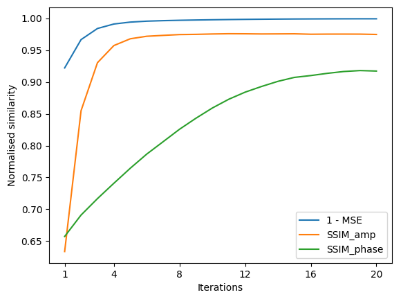


(c) deviation 7

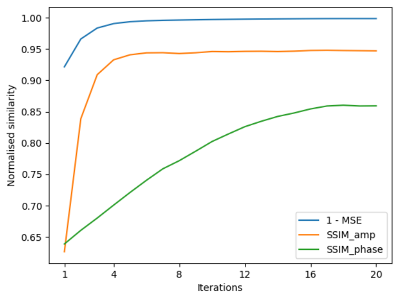


(d) deviation 8

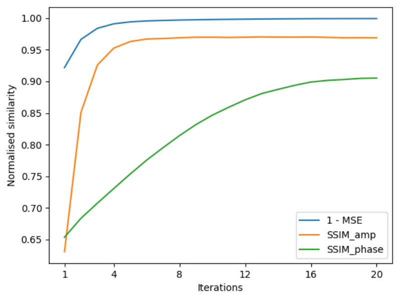
Figure B.12: Convergence of deviation 5-8 on the USAF data set



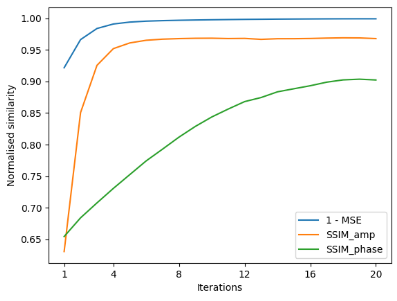
(a) deviation 9



(b) deviation 10

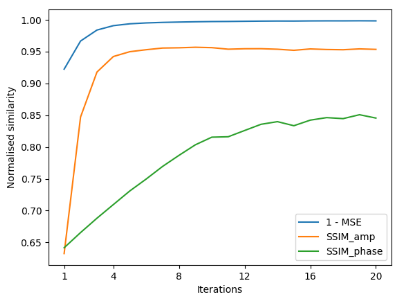


(c) deviation 11

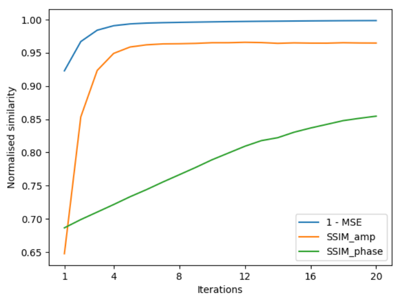


(d) deviation 12

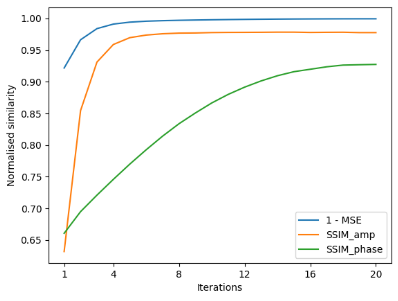
Figure B.13: Convergence of deviation 9-12 on the USAF data set



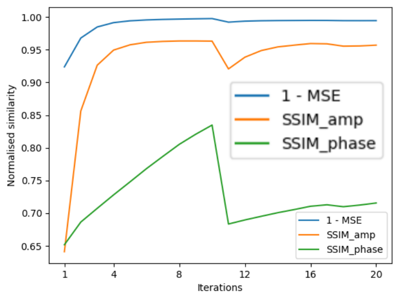
(a) deviation 13



(b) deviation 14

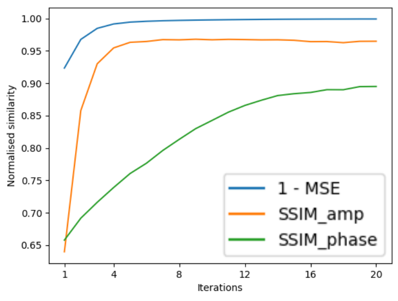


(c) deviation 15



(d) deviation 16

Figure B.14: Convergence of deviation 13-16 on the USAF data set



(a) deviation17

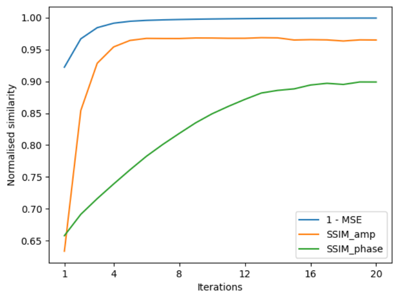


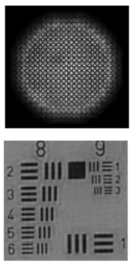
Figure B.15: Convergence of deviation 17-18 on the USAF data set

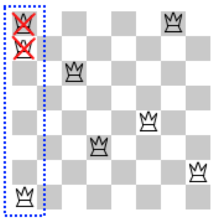
Appendix C

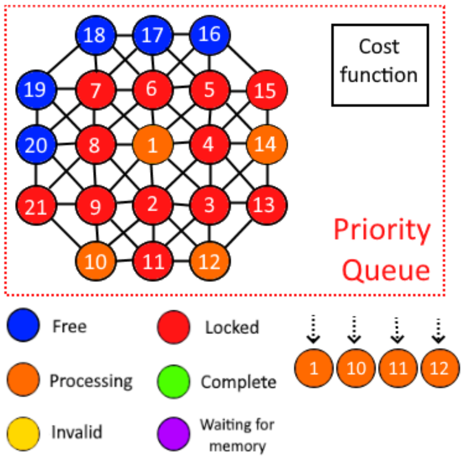
Appendix C – Project poster

Dynamic Task parallelism in FPM and related methods

Many iterative methods works by updating an estimate by fitting it to a collection of constraints in one or more domains. One example is the method of phase retrieval as used in Fourier ptychography (FPM). In FPM, a collection of images each impose a constraint on a known region of the Fourier spectrum of some imaged object. To recover the object, an initial guess is fitted to all the measured constraints. Usually, this is performed iteratively, as the overlap between sub-iterations in Fourier space must be respected to avoid race conditions. Simply partitioning the sub-iterations to parallelise is therefore not ideal, as it is hard to stop overlapping iterations across partitions from executing concurrently.







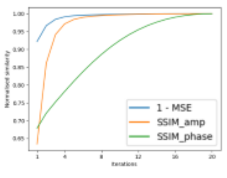
However, by analysing the dependencies between the constraint updates, a dependency graph can be constructed. Combined with a cost function and a basic, preferred execution order, this graph lets us construct a priority queue that can supply threads with thread safe iterations for concurrent execution.

One of the advantages of the central priority queue is that is can monitor the state of the entire algorithm. This permits central control of synchronisation, and a cost function that can use global information to adjust what nodes should be made available for execution. Nodes can also be annotated with more information than just READY, PROCESSING, LOCKED and COMPLETE, if needed for more complicated cost functions

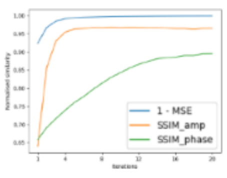
By applying the method outlined above to FPM, the running time was reduced by 5 in the best case, while keeping the reconstruction quality high compared to a single threaded baseline. To achieve this, the cost function had to include a threshold for how out of order an iteration could be performed with respect to the default single-threaded order.

Experiment name	Wall clock time	Processor time	Max threads	Avg. threads	SSIM phase
devitaion 17	30.56	288.21	16	11.27	0.85
devitaion 14	34.04	255.97	16	8.88	0.84
devitaion 18	34.79	277.50	16	6.98	0.85
deviation 15	39.51	230.43	16	6.98	**0.87**
deviation 5	40.81	179.48	8	6.37	0.85
baseline	203.33	203.33	1	1	**0.89**
baseline-8	159.51	159.18	1	1	**0.89**
baseline-16	154.26	153.89	1	1	**0.89**

Moving forward, it would be of interest to apply the priority queue based scheduler to more problems in the same family as FPM, such as Ptychography or other alternating projection based solvers. It is also worth noting that the parallelism exposed by this approach is independent of any inherent data parallelism. Combining the task parallel capabilities of the dynamic priority queue scheduler and the data parallel nature of sub-iterations could yield algorithms faster than either could achieve alone.



Baseline



Deviation 17

Øystein Rognebakke Krogstie

Dynamic task parallel FPM

NTNU
Norwegian University of
Science and Technology