Algorithmic and Infrastructural Software Development for Cryo Electron Tomography



Thomas Frosio

New College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy in *Cellular Structural Biology*

Michaelmas 2023

Abstract

Many Cryo Electron Microscopy (cryoEM) software packages have accumulated significant technical debts over the years, resulting in overcomplicated codebases that are costly to maintain and that slow down development. In this thesis, we advocate for the development of open-source cryoEM core libraries as a solution to this debt and with the ultimate goal of improving the developer and user experience.

First, a brief summary of cryoEM is presented, with an emphasis on projection algorithms and tomography. **Second**, the requirements of modern and future cryoEM image processing are discussed. **Third**, a new experimental cryoEM core library written in modern C++ is introduced. This library prioritises performance and code reusability, and is designed around a few core functions which offers an efficient model to manipulate multidimensional arrays at an index-wise and element-wise level. C++ template metaprogramming allowed us to develop modular and transparent compute backends, that provide great CPU and GPU performance, unified in an easy to use interface. **Fourth**, new projection algorithms will be described, notably a grid-driven approach to accurately insert and sample central slices in 3-dimensional (3d) Fourier space. A Fourier-based fused backward-forward projection, further improving the computational efficiency and accuracy of reprojections, will also be presented. **Fifth**, and as part of our efforts to test and showcase the library, we have started to implement a tilt series alignment package that gathers existing and new techniques into an automated pipeline. The current program first estimates the per-tilt translations and specimen stage rotation using a coarse alignment based on cosine stretching. It then fits the Thon rings of each tilt image as part of a global optimization to estimate the specimen inclination. Finally, we are using our Fourier-based fused reprojection to efficiently refine the per-tilt translations, and are starting to explore ways that would allow us to refine the per-tilt stage rotations.

Acknowledgment

I would like to thank my supervisor Professor Peijun Zhang for her guidance throughout my DPhil. Thank you for believing in me and for creating an environment that allowed me to follow my newly found passion. I would also like to thank my Director of Graduate Studies, Professor Matt Higgins, for his advice and support, particularly during the pandemic. I thank Ben A. Himes, as well as David N. Mastronarde and Daniel Castaño-Díez, for taking the time to answer my many questions about image processing and cryoET.

Thanks to Luiza, Max, Dimple, Anna-Sophia and Alister. I'm truly honoured to be part of our club, I'm glad our paths crossed. Special thanks to Luiza, spending time with you was an enriching experience, you are truly one of the greatest people I have ever met. Another special thanks to Alister, the lad, I'm really grateful to have you as a friend and for our walks in University park.

And of course, to mum, dad and my little sister. I could not have done it without your continued encouragement over all these years and throughout the writing of this thesis. Thank you for your unconditional love and support. I love you.

Contents

1 Cryo Electron Microscopy and Tomography **9**

1.1 Structural biology . . . 11

1.1.1 Structure and function . . . 11

1.1.2 X-ray Crystallography . . . 11

1.1.3 Cryo Electron microscopy . . . 13

1.2 Image formation in cryoEM . . . 13

1.2.1 Electrons are scattered . . . 14

1.2.2 Focusing and magnifying lenses . . . 15

1.2.3 Phase contrast . . . 16

1.2.4 Optical aberrations . . . 17

1.2.5 Thin specimen . . . 18

1.2.6 Not so thin specimen . . . 19

1.3 Single Particle Analysis . . . 22

1.4 Backward and forward projections . . . 24

1.4.1 Backward projection . . . 24

1.4.2 Forward projection . . . 26

1.4.3 Ewald sphere curvature . . . 26

1.5 Tomography . . . 27

2 Computational solutions to cryoEM **29**

2.1 Software and academic research . . . 30

2.1.1 Technical debt . . . 30

2.1.2 The case for core libraries . . . 31

2.2 Manipulating cryoEM data . . . 32

2.2.1 Multidimensional arrays . . . 32
2.2.2 Two different paradigms for accessing data . . . 34
2.2.3 Generating efficient code using an nd-array syntax . . . 36
2.3 What could a cryoEM core library look like? . . . 39
2.3.1 Core functions and domain-specific pointwise operators . . . 39
2.3.2 Vectorisation and Parallelism . . . 40
2.3.3 Mathematical optimisations and artificial neural networks . . . 44

3 An experimental cryoEM core library 47
3.1 Goals and design choices . . . 49
3.1.1 Choosing a programming language . . . 49
3.1.2 Performance and simplicity . . . 50
3.1.3 Device agnostic code . . . 50
3.2 Execution model . . . 51
3.2.1 Devices, memory resources and streams . . . 51
3.2.2 Asynchronous eager execution . . . 54
3.2.3 CPU support . . . 56
3.2.4 GPU support . . . 57
3.3 Multidimensional data . . . 64
3.3.1 BDHW order . . . 64
3.3.2 Strides and memory layout . . . 65
3.3.3 Array and View . . . 66
3.3.4 Accessor and (md)span . . . 67
3.4 Core functions . . . 72
3.4.1 Index-wise . . . 73
3.4.2 Element-wise . . . 74
3.4.3 Reductions . . . 76
3.5 cryoEM specificities . . . 80

3.5.1 Geometric transformations 80
3.5.2 FFT layouts 83

4 Projections in Fourier space 88

4.1 Backward projection 89
4.1.1 Using the data-driven approach 89
4.1.2 Using the grid-driven approach 92
4.1.3 Correcting for the apodisation of the reconstruction 99
4.1.4 Normalising the multiplicity 100
4.2 Forward-projection 102
4.2.1 Fourier extraction 102
4.2.2 On-the-fly windowed sinc convolution 103
4.3 Fused backward-forward projection 107

5 Tilt series alignment 112

5.1 Tilt series alignment and tomogram reconstruction 114
5.1.1 Tilt scheme 114
5.1.2 Tilt series alignment strategies 115
5.1.3 Geometric model and naming conventions 117
5.2 Coarse stage rotation and per-tilt 2d translation alignment 118
5.2.1 Translation alignment using pairwise cross-correlation 118
5.2.2 Stage rotation alignment using cosine stretching 119
5.2.3 Refining the per-tilt translations using cosine stretching 121
5.3 Specimen orientation within the microscope using CTF fitting 122
5.3.1 Initial CTF fitting on average power-spectrum 122
5.3.2 Fitting 3d specimen orientation using global CTF fitting 123
5.3.3 Defocus gradient 125
5.4 Estimating the specimen thickness 126

5.5 Refining the stage rotation and per-tilt translation 127
5.5.1 Projection matching in Fourier space using fused reprojection . . . 127
5.5.2 Per-tilt translation alignment 129
5.5.3 Difficulties with fitting the per-tilt stage rotation 131
5.6 Fourier-based tiled reconstruction 134
5.7 Measuring the quality of the alignment 134

Bibliography **136**

Acronyms **161**

Figures

1.1 Opposite scattered waves interfere at the image plane 20

1.2 EWS curvature and Friedel related beams 21

3.1 Current functions available in the library frontend 87

4.1 Wrap-around effect in Fourier-based reconstructions 95

4.2 Inserting a central slice in Fourier space using a grid-driven approach . . . 98

4.3 Effect of the convolution kernels used in the backward projection 101

4.4 Extraction of a central slice with the on-the-fly w-windowed sinc convolution 106

4.5 Reprojection a tomogram with and without the w-windowed sinc convolution 108

4.6 Sampling of a central slice from a virtual volume made of other central slices 109

4.7 Sampling a discretised and virtual volume 111

5.1 Automated thickness estimate 128

5.2 Per-tilt translations before and after projection matching 132

Snippets

3.1 Devices and Streams 52

3.2 Asynchronous eager execution 54

3.3 User-defined type and operator to the CUDA backend using nvc++. 58

3.4 User-defined type and operator to the CUDA backend using nvrtc. 60

3.5 Layout of `Array` and `View` 67

3.6 Layout of `Accessor` 68

3.7 Examples and usage of `Accessor` 69

3.8 Different ways to access the elements of an `Array` 71

3.9 `noa::iwise` 73

3.10 `noa::ewise` 75

3.11 `noa::reduce_ewise` and `noa::reduce_iwise` 76

3.12 Matrices and simple geometric transformations 80

3.13 `Texture` in device-agnostic code 82

3.14 Examples using rFFTs 85

4.1 Gridding operator 89

4.2 Grid-driven insertion operator 92

4.3 Sampling a central slice from a 3d volume 102

4.4 Sampling a central slice with the on-the-fly convolution along the w-axis . 104

Chapter 1

Cryo Electron Microscopy and Tomography

1.1 Structural biology 11
1.1.1 Structure and function 11
1.1.2 X-ray Crystallography 11
1.1.3 Cryo Electron microscopy 13
1.2 Image formation in cryoEM 13
1.2.1 Electrons are scattered 14
1.2.2 Focusing and magnifying lenses 15
1.2.3 Phase contrast 16
1.2.4 Optical aberrations 17
1.2.5 Thin specimen 18
1.2.6 Not so thin specimen 19
1.3 Single Particle Analysis 22
1.4 Backward and forward projections 24
1.4.1 Backward projection 24
1.4.2 Forward projection 26
1.4.3 Ewald sphere curvature 26

1.5 Tomography 27

In this chapter, we present the essential components of cryoEM in relation to this thesis. As such, we notably introduce the projection transforms and how they relate to the central slices in Fourier space.

1.1 Structural biology

1.1.1 Structure and function

DNA, RNA, proteins, lipids and sugars, all collectively called biomolecules, are the fundamental building blocks of all living organisms on Earth. Among other things, lipids and sugars form complex sheltered and regulated compartments called cells, where DNA stores the organism's genetic information and proteins contribute to its repair and replication [1]. Proteins also facilitate a wide range of other cellular functions, representing enormous scientific interest [2].

Structural biologists work to obtain a structural description of these biomolecules, as well as their location and their biological partners in cells [3], in order to better understand the functioning of living organisms. With a detailed mechanistic understanding of a given biomolecule, it is indeed possible to deduce its functions and roles [4], but also how mutations in its sequence could affect its behaviour and lead to diseasess, or even how to design or improve molecules that target it for therapeutic purposes [5].

It is becoming increasingly clear that nature offers almost infinite variations in the specialisation of proteins and other biomolecules [6]. We are continually discovering new aspects of molecular structure and dynamics that tell us how biological processes work and what we might do to influence or use them. This is why structural biology has developed so much since its invention and why the development of tools to determine biomolecular structures will continue to be important.

1.1.2 X-ray Crystallography

X-ray crystallography was the first technique to solve the structure of a biomolecule, more than a century ago, and has since become the technique that has generated the largest number of high-resolution models of biomolecules reported to date [7].

The main idea of crystallography is to amplify the scattered signal by organising biomolecules in a crystal [8]. Indeed, a single biomolecule would need to interact with an enormous amount of light in order to overcome the noise of a macroscopic measurement and establish a well-resolved image. On the other hand, a crystal lattice allows identical scattering of several millions of biomolecules, thereby multiplying the overall signal while keeping the same nominal noise level, ultimately allowing faithful images to be collected before detrimental radiation damage. Although extremely challenging, when successful, crystallisation combined with a bright and coherent X-ray source can form an excellent imaging system capable of resolving the electron cloud of biomolecules with atomic resolution [9].

Since its original development, considerable efforts were made to increase the accuracy, robustness and throughput of this method. X-ray crystallography beamlines at synchrotron sources are constantly improving [10], facilitating the study of ever smaller crystals, as well as crystals with very large unit cells. The scientific community has gained widespread access to these high-quality X-ray sources, thanks to faster and more user-friendly data collection procedures and highly matured software tools [11]. As a result, X-ray crystallography has come to shape our understanding of how biological processes take place, guided drug development by targeting specific biochemistry, and showed us why some genetic disorders manifest the way they do.

However, while depositions to the Protein Data Bank (PDB) have kept on growing [12], certain targets have remained dramatically underrepresented. Membrane proteins, proteins with flexible domains, large or unstable molecular assemblies, as well as biological polymers, have all posed a particular challenge to the crystallisation procedure. Furthermore, while crystallisation may favour states or assemblies of biomolecules that would not spontaneously persist for long enough to be imaged in the natural cellular environment, it may very well do the opposite and stabilise conformational states that are not favoured in cells [13].

1.1.3 Cryo Electron microscopy

Since as early as the studies of DNA fibres using X-ray crystallography, electron microscopy has been used as an alternative, but it is only recently, thanks to considerable development both in hardware [14] and software [15], that it has become a fully vested structural technique, capable of systematically providing atomic resolution information (or nearly so) of biological molecules, while using an imaging technique that is closer to the natural cellular environment.

The central idea of cryoEM is to cryogenically freeze a biological sample in a thin layer of non-crystalline amorphous ice so that it is preserved in a near-native state and can be imaged in a Transmission Electron Microscope (TEM) [16]. The vitrified ice layer should be thick enough to fully embed the sample within the vitrified ice layer, keeping the macromolecules hydrated and avoiding the destructive air-water interface, while being as thin as possible to minimise the amount of noise in the images [17, 18].

As part of introducing the next chapters of this thesis, the next sections will focus on cryoEM, starting by introducing how images are formed by the electron microscope, and then by presenting how these images are often processed and analysed.

1.2 Image formation in cryoEM

TEMs produce a beam of highly accelerated electrons that illuminates a section of the specimen. In this chapter, we will assume that the microscope produces a beam resulting in parallel illumination, *i.e.* the incident electron wave is a plane wave, that is, both the amplitude and the phase of the wave are constant over any given plane that is normal to the direction of propagation [19, 20]. Small deviations between the direction of propagation and the optical axis, referred to as beam-tilt [21, 22], might persist after the alignment of the microscope. Fortunately, this residual or unnoticed beam-tilt can be accounted for during offline data processing [23]. As such, we will also assume that the direction of propagation

is along the optical axis.

It should be noted that the high velocity of electrons in a typical microscope (between 100 kV and 400 kV for cryoEM) creates a large spatial separation between electrons in the beam, to the point where there's often only one electron passing through the sample at a given time.

1.2.1 Electrons are scattered

As the electron plane-wave passes through the specimen, scattering occurs. Inelastic scattering events, events during which the electrons lose energy, are by far the main outcome from the interaction with the sample [24]. As far as cryoEM is concerned, these interactions do not directly contribute to the useful signal and although these can be mostly eliminated by the use of an energy filter [25, 26], inelastic events remain the main source of noise in cryoEM images.

The main source of signal comes from elastic Coulombic interactions between the incident electron and atoms in the specimen, resulting in phase modulations in the exit wave [27]. Although the Coulomb potentials of the positively charged atomic nuclei of neutral atoms are shielded by their surrounding electron clouds, once the incident electron enters this cloud, it accelerates as it approaches the nucleus, and it decelerates back to its original speed as it exits the electron cloud [28]. During this process, the electron wavelength transiently becomes shorter, before returning to its initial value, resulting in a phase delay relative to other regions of the wave that were either unscattered or scattered differently by other atoms.

Small and localised amplitude modulations also occur, mostly resulting from absorption of electrons in the specimen, or due to the "virtual absorption" from the loss of electrons scattered outside the objective aperture [29]. Energy filters, on top of reducing background noise, can also enhance the amplitude contrast by removing inelastically scattered electrons

[25]. These amplitude modulations are often approximated as being a fixed amount, between 0.07 and 0.1 times the phase-modulation. Although that is expected to be a rather poor approximation, a more accurate approximation is hardly needed in practice, since the effect of amplitude contrast is small to begin with and mainly affects low-frequencies.

Considering that biological macromolecules are made of light elements, the elastic scattering results in only small phase modulations and even smaller amplitude modulations of the scattered electrons [16]. As such, the specimen is said to be a mixed weak-amplitude and weak-phase object. Since only these small modulations are expected and since the phase modulations are much larger than the amplitude modulations, we can accurately assume that the exit wave function, as well as the image intensity, is a linear function of the Coulomb potential of the specimen.

1.2.2 Focusing and magnifying lenses

Interestingly, the scattered electron wave we have just described and the objective lens of the microscope effectively approximates a Fourier transform of the Coulomb potential of the specimen [29, 30, 31]. The intensities of the Fourier components can be directly observed at the back focal plane of the objective lens. However, instead of collecting these intensities, the wave is focused at the back image plane, a process during which the Fourier components are added together, effectively computing the inverse Fourier transform of the exit wave at the back focal plane. Importantly, another set of magnifying lenses are then used to enlarge the resulting image wave function at the back image plane.

Due to the imperfect optical system, mainly due to the optical aberrations of the electromagnetic lenses, this process of focusing and magnifying also convolves the exit wave with a microscope-specific wave aberration function, also referred to as the Point Spread Function (PSF) [32]. At this point, a perfect point at the immediate exit wave may not end up in the final image plane as the same, but magnified, perfect point.

By performing this analysis in Fourier space, we can see that the Fourier transform of the image intensities is the Fourier transform of the Coulomb potential of the specimen, also referred to as the structure factor of the object, multiplied by the Fourier transform of the PSF, often referred to as the Contrast Transfer Function (CTF) [19]. Note that the structure factors and the CTF correctly include the small amplitude modulations in the exit wave and the transfer function for amplitude contrast, respectively.

The CTF therefore acts as a linear filter operation in Fourier space. From this, we directly observe that a binary CTF with a spatial cutoff would act as a lowpass filter and blur any higher resolution feature beyond possible retrieval. Incomplete attenuation of information may however still be recovered through appropriate inversion of the filter (deconvolution of the PSF). As such, knowledge and correction of the CTF can help us, and is necessary to, restore the best possible representation of the scattered object [33, 34].

1.2.3 Phase contrast

Weak-phase objects modulate predominantly the phase and much less the amplitude of the electron wave. If the microscope was an ideal aberration-free optical system (and thus was perfectly in focus), then images of biological samples would not show any contrast. Indeed, the electron detector measures the intensity of the electron wave and cannot directly measure its phase. Therefore, to enable visualisation of the phase modulation, the wave must somehow be optically manipulated in a way that converts the phase information into intensity modulation [35].

Fortunately, optical aberrations, which should be all accounted for in the CTF, can create phase delays between the scattered and unscattered part of the electron wave, resulting in the structure factors of the sample having non-zero phases and amplitudes, ultimately creating (phase-)contrast [19, 20]. The choice of which optical aberrations are best to create contrast is discussed in the next section. Unfortunately, this contrast is not perfect across the frequency spectrum, some frequencies being more contrasted than others, some

displaying a negative contrast or no contrast at all. While knowledge and correction of this oscillatory CTF, as it is now routinely performed [26], can fix this imperfect contrast, frequencies with zero contrast cannot be corrected, since in fact, they were never detected to begin with.

1.2.4 Optical aberrations

As introduced in the previous sections, the CTF is the function containing all of the optical aberrations that the exit wave has accumulated. These aberrations form a diverse set of functions that can be divided into two components, the symmetric component which scales the structure factors of the sample (it is a real value), and the antisymmetric component which shifts their phases (it is a complex value). Because symmetric and antisymmetric aberrations have very different effects on the resulting image, they also need to be estimated and corrected in different ways [36].

Since symmetric aberrations scale the complex magnitudes of structure factors, they can be seen as Thon rings in the power spectra of micrographs [37]. This typically allows us to directly measure the first two orders of symmetric aberrations relatively easily. That includes, the defocus and astigmatism, as well as the constant phase shift resulting from an uniform phase plate, if any. While astigmatism is not desirable, in order to create (phase-) contrast, cryoEM images are intentionally collected with a significant amount of either defocus or phase-shift. Additionally, because microscopes have a considerable amount of the fourth-order spherical aberration, it is also almost always considered [38]. A significant portion of the cryoEM processing pipeline only considers these four aberrations and since all of them are symmetric, the CTF can be simplified to a real-valued function. Unfortunately, the fact that the symmetric aberrations scale the complex magnitudes of structure factors makes them difficult to correct since they cannot be easily cancelled out. Instead, the correct magnitude of the CTF has to be considered throughout the entire data processing pipeline.

Antisymmetric aberrations, e.g. the axial coma resulting from beam-tilt, are way more

difficult to measure. Indeed, the phase shift arising from these aberrations is invisible in the power spectrum of the image, so it cannot be measured without knowing the true phase angles of the structure factors, *i.e.* without prior knowledge about the sample. It was only in 2018, thanks to a refinement procedure using a prior reference [38], that antisymmetric and higher order symmetric components were able to be accurately estimated on cryoEM data. Thankfully, once they are known, these antisymmetric components are trivial to correct and can be easily cancelled out by applying the opposite phase shift to the observed image frequencies.

1.2.5 Thin specimen

As introduced in the previous sections, biological objects are weak-phase objects, so most of the signal coming from the scattering with the electron wave is in the form of small phase modulation. Because these modulations are so small, we can relate the wave function to a linear function of the Coulomb potential of the specimen.

Interestingly, another useful approximation can be made. If the sample is thin enough, we can model the electron wave scanning through and exiting the sample as a single exit wave (as it was implicitly suggested until now). In this case, the phase modulations are the result of the *integration* of the Coulomb potential of the specimen, *i.e.* the exit wave is a *linear projection* of the Coulomb potential [32, 39, 40].

In Fourier space, and according to the central slice theorem, this results in cryoEM images being central slices of the Fourier transform of the 3d Field Of View (FOV) [30, 32]. In other words, the structure factors of the specimen that are sampled when collecting an image all lie on the plane orthogonal to the beam and located at the origin of the Fourier transform of the specimen. Because it is a plane orthogonal to the beam in Fourier space, it is easy to see that when the projection assumption holds, cryoEM images have no depth information.

1.2.6 Not so thin specimen

Central slices are an incredibly useful construct and are often used during data-processing. But what happens when the specimen is not so thin? In this case, we can fall back on a more accurate model in which the defocus gradient within the specimen is accounted for [39, 41]. Indeed, the specimen can be divided into multiple infinitely thin slices that are added up when recorded on the detector. When performing this analysis in Fourier space, we find that cryoEM images are in fact sampling the Fourier components that lie on the Ewald Sphere (EWS) [42] and not on the central slice.

Sampling the EWS instead of the central slice has a huge theoretical consequence [41, 43]. When opposite wavelets at the back focal plane are focused back to the image plane, they interfere to form a single wave (see fig. 1.1). The amplitude and phase of that recombined wave is the linear combination of the amplitude and phase of the two waves. At low resolution, when the projection approximation is still valid, the two waves are Friedel related, *i.e.* they are complex conjugates of each other, so no information is lost during the focusing of the waves. Indeed, the Fourier transform of the real-valued image will compute the two waves back. On the other hand, at high resolution, when the curvature of the EWS becomes significant (see fig. 1.2), the opposite wavelets are not Friedel related [41]. So while they have the same absolute frequency, they can have different amplitudes and phases. When focused, the interference of the opposite waves results in a loss of information since only the amplitude and phase of the resulting wave is measured at the image plane. When computing the Fourier transform of the resulting real-valued image, only this summed amplitude and phase will be available. Interestingly, one consequence of the breakage of the Friedel symmetry is that the CTF itself becomes complex, even when only considering the symmetric aberrations [36]. Indeed, the sum of the symmetric transfer functions of the opposite waves is only real-valued when the waves are complex conjugates of each other.

While cryoEM software could already account for the EWS curvature and complex CTFs

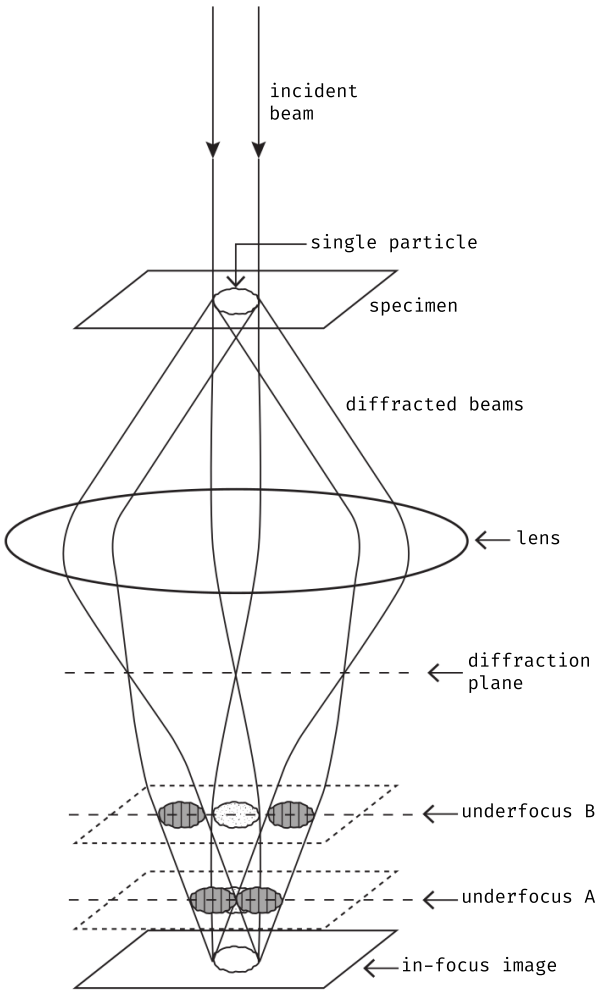


Figure 1.1: Diagram showing how the opposite wavelets are focused back to the image plane, and interfere with each other and the unscattered beam to form the image. Adapted from [44].

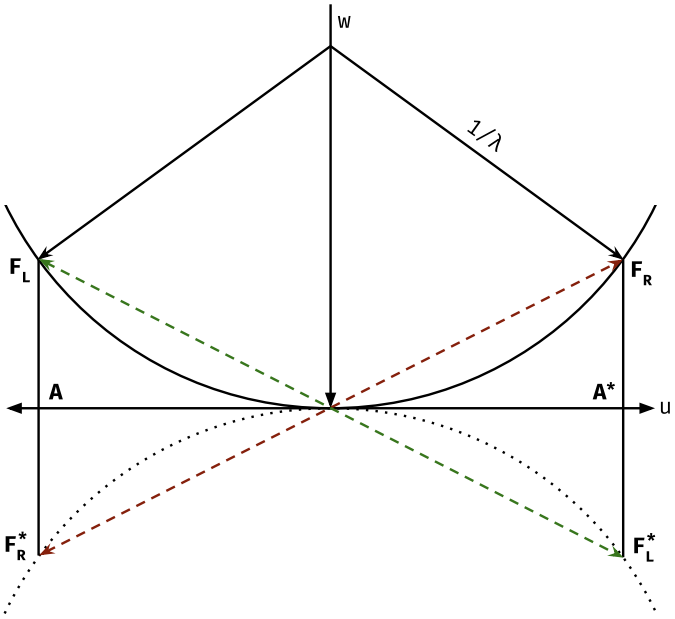


Figure 1.2: Diagram representing a slice through the 3d Fourier volume along the uw-plane. The incident beam is scattered by the object, which emits two symmetrically scattered beams. The EWS surface represents all possible elastically scattered waves. In this diagram, we show two symmetric beams, represented by the Fourier coefficients F_L and F_R. When the EWS curvature is ignored, F_L and F_R lie on A and A^*, respectively. In other words, the two beams are complex conjugate of one another, *i.e.* they are Friedel related. When considering the EWS curvature, however, F_L and F_R are unrelated. Note that this diagrams greatly exaggerates the EWS curvature. In cryoEM, given the wavelength, spacing and size of the reconstructions often used, the EWS at Nyquist is only about a few voxels away from the projection plane.

more than a decade ago [43], a more accurate method allowing to (partially) recover the left and right beams (by an appropriate use of defocus and two real space masks) was found in 2018 [44], and later implemented in reconstruction and refinement algorithms [23, 45].

So which one to choose, the central slice or the EWS? Given the small wavelength of the accelerated electrons used in cryoEM, and the size and sampling of the reconstructions, the section of the EWS that is sampled almost always perfectly overlaps with the central slice, *i.e.* with or without the projection approximation, we end up sampling almost exactly the same Fourier components. In this case, it may be simpler and faster to use the central slice theorem. It is only for large reconstructions and at high frequencies that the two constructs stop overlapping, and that we may see benefits in falling back on the more accurate EWS [46]. Fortunately, in practice, choosing to model the Fourier transform of the images as sections of the EWS or as central slices does not change drastically how the data processing is actually done, and as we will see, most algorithms dealing with central slices can easily be adapted to the EWS. As a result, while referring to EWS sections is more accurate in some situations, cryoEM images are often (rightfully) simplified and referred to as projections.

1.3 Single Particle Analysis

For many species of biological macromolecules, copies of the same molecule exhibit a very high degree of structural similarity. During sample preparation, many copies of these macromolecules end up embedded, in arbitrary positions and orientations (ideally), in vitreous ice. During data collection, many 2-dimensional (2d) projections of these macromolecules, then referred to as particles, are acquired. Despite the very low Signal to Noise Ratio (SNR) of individual particles, the process of accurately filtering and averaging 2d projected copies of the same biological macromolecule can result in atomically accurate reconstructions of the 3d Coulomb potential of the same macromolecule [30, 47]. This process is referred to as Single Particle Analysis (SPA) [48].

In practice, SPA is divided into multiple steps. First, in order to average particles together, it is necessary to locate and find the centre of each particle [47]. Given that images are recorded with a low electron radiation to limit radiation damage and preserve features of interests (low dose mode, introduced by [49]) thus resulting in a low SNR, and due to the possible presence of other molecules and artefacts in the images, this first step is a challenge in itself. Global and local motion happening during exposure, also known as Beam Induced Motion (BIM), should be accurately fitted [50]. Appropriate per-image and per-particle filtering should be done to account for the CTF, as well as electron radiation and imperfections of the instruments [17, 23, 51, 52]. Importantly, the Spectral Signal to Noise Ratio (SSNR) should also be estimated and added to the filtering function to give less weight to the noisier parts of the signal.

Then, these initial measurements are refined using reference-based optimisations [47]. Indeed, once a reference is computed from the initial particle average, optimising the similarity between individual particles and projections of that reference can be used to greatly improve the accuracy of many parameters, or even allow new ones to be fitted. Reference-based optimisation can, for instance, improve the fitting and correction of existing CTF components. Some aberrations, *i.e.* asymmetric and high-order symmetric aberrations, can only be fitted using a reference [38]. The BIM can also be improved by tracking the temporal and spatial motion of the particles against a high SNR reference, in a step often referred to as particle polishing [53]. Moreover, the determination of the 3d position and orientation of the particles themselves is significantly improved using reference-based optimisations. In case of a heterogeneous sample, this process is more complicated because each particle must then be compared to multiple references to find the group it most likely belongs to [45, 54, 55]. Noticeably, the newly produced reconstruction(s) can then be used as new reference(s) for a new cycle of optimisation, revealing the inherent iterative property of SPA.

In summary, many parameters should be considered to obtain reconstructions with atomic resolutions. Most of these parameters aim to accurately model the noise in the data and

to correct it as far as possible so that the resulting normalised noise left in the data can be averaged out during reconstruction. During recent years, reference-based optimisations have constantly been improved and new ideas and algorithms have emerged, making cryoEM and SPA reliable and powerful tools for structural biology.

1.4 Backward and forward projections

Computing and reprojecting 3d reconstructions from and to 2d projections is central to the cryoEM workflow. These two steps, referred to as the backward and forward projection, are extensively discussed and used in this thesis, so they deserve a more detailed introduction.

Backward projection, often abbreviated to backprojection, is the transformation that projects a n-dimensional reference frame into a $(n+1)$-dimensional reference frame. In cryoEM, n is usually 2. The forward projection is simply the reverse operation.

Both backward and forward projections can be efficiently computed using the Fourier slice theorem [56, 57]. This is the approach that most cryoEM software packages have taken, both for its computational efficiency and simplicity, but also for its ability to accurately keep track of what happens to each Fourier frequency during the projection. The theorem states that the Fourier transform of a projection represents the values of a central section through the Fourier transform of the higher dimensional object. In our case, this means that the 2d Fourier transform of a projection of a 3d object corresponds to a 2d central slice of the Fourier transform of that 3d object.

1.4.1 Backward projection

Let's start by defining an empty 3d Fourier volume. Central slices, *i.e.* the Fourier transforms of the projections, can be inserted into this volume [58, 56]. The insertion method and how the central slices are modelled will be extensively discussed in chapter 4 and are intentionally omitted in this introduction. The transformation parameters of the slice relate the 2d

reference frame of the projection to the 3d reference frame of the reconstruction. In other words, each central slice has a 3d rotation, which is equal to the rotation of the real space object relative to the beam. Note that an additional 2d scaling matrix is often added to the central slice geometry to account for (anisotropic) magnification differences between projections. While these could be applied before the backward projection (or after the forward projection), most implementations prefer to include the magnification correction directly into the Fourier insertion [15]. Importantly, the rotation centre of the projections should be shifted to a common point. This operation is often done by phase shifting the central slices. At this point, the slices can be rotated in 3d space and inserted into the Fourier volume, effectively (backward) projecting the 2d projections to a common 3d reference frame.

When inserting multiple central slices into the same volume, the 3d frequencies always end up with an uneven sampling, e.g. some frequencies are sampled more than once [56]. Fortunately, the multiplicity, *i.e.* how many times a given frequency is sampled, can be easily tracked and corrected during the Fourier insertion. Moreover, each central slice is often attached to a 2d sampling function, often simply referred to as weight, containing the spectral parameters of the projection. These notably include the (complex) CTF, exposure weighting, and other envelopes. As such, instead of simply inserting the values of the central slices, both the values and corresponding weights are commonly inserted at the same time.

Once the insertion of all of the central slices is done, there's usually two possibilities: forward projection or real space reconstruction. The former case is presented in the next section. In the latter case, the weights are applied and the inverse Fourier transform of the 3d Fourier volume is computed, outputting the real space 3d reconstruction [56]. Note that the way the weights are applied depends on the insertion method (see section 4.1.4), as well as the additional filtering that is often done before or after the inverse Fourier transform.

1.4.2 Forward projection

As mentioned previously, reference-based optimisations are central to cryoEM image processing [47]. These optimisations require the ability to forward project a 3d reference, *i.e.* a real space reconstruction, into a 2d projection. While this can be done by naively integrating the reference along the projection line, the Fourier slice theorem still offers a much more efficient and accurate option.

So far we have seen that we can insert central slices into a Fourier volume to compute the backward projection. However, and still according to the Fourier slice theorem, the inverse operation of extracting, *i.e.* sampling, a central slice from the Fourier volume results in the forward projection of that reconstruction. Similarly to the insertion, the projection axis defines the central slice that should be extracted. Importantly, both the values and the weights can be accurately extracted too. This is one advantage of these Fourier-based methods compared to the real space ones [58], and is of great importance when the resulting projections are to be used for reference-based alignments.

1.4.3 Ewald sphere curvature

As we now know from section 1.2.6, cryoEM images are not exactly projections and therefore do not perfectly sample central sections. Instead, they sample the EWS [42]. One advantage of the Fourier slice theorem is that while it is formulated around projections and central sections, it can be relatively easily adapted to the EWS: instead of inserting or extracting central slices, we can insert or extract central sections of the sphere.

Importantly, because the projected output has to be real, the Hermitian symmetry of the Fourier volume must not be broken. In other words, when inserting a value at a frequency, the complex-conjugate of that value should also be inserted at the Hermitian-related frequency [41, 43]. When dealing with central slices, this happens naturally because the left and right beams are the Hermitian pair, *i.e.* they are related by a central symmetry and are complex

conjugates of each other. In other words, the left beam is equal to the Hermitian mate of the right beam, and vice-versa. However, when dealing with the EWS, the left and right beams are not Hermitian pair, so the Hermitian mates of both beams should be explicitly sampled (see fig. 1.2). This effectively results in having to write to (during Fourier insertion), or to read from (during Fourier extraction), twice as many frequencies compared to the central slice case.

1.5 Tomography

In some specimens, the biological macromolecules of interest cannot be isolated from their environment. These cases are often not suitable for the classic SPA because densities in individual images often overlap with the objects of interest, thereby limiting the accuracy or even preventing their alignments [59, 60]. In tomography, a series of images from the same FOV is recorded as the sample is tilted through a set of defined tilts relative to the incident electron beam. This series of tilted images, referred to as tilt series, is then aligned to a common 3d reference frame [61]. The details of this alignment will be further discussed in this thesis, but the ultimate goal is to produce, from the backward projection of the tilt series, a 3d reconstruction of the FOV, referred to as tomogram [62].

Importantly, these tomographic reconstructions have significant distortions in every direction except in the planes (one for each tilt) orthogonal to the incident beam, so while unique objects and environments can be 3d reconstructed, thus allowing direct 3d analysis of crowded environments, their structural analysis is currently limited to low resolution observations. On the other hand, multiple copies of the objects of interest can be extracted from tomograms. Therefore, SPA-like techniques can then be used to process these subtomograms as "3d single particles". This is referred to as Subtomogram Averaging (STA) [63, 64].

While SPA algorithms can often be directly applied to STA, care should be taken when importing SPA implementations to STA. For instance, while in theory no distinctions should

be made between the Fourier insertion of central slices from a single particle dataset and a tilt series, in practice, things can be more complicated. Following this example, in SPA, the value and weights of most 3d Fourier components of a reconstruction is the result of possibly tens or even hundreds of thousands of central slices. On the other hand, most 3d Fourier components of a single (sub)tomogram are either only sampled by a few central slices or not sampled at all [52]. In this case, sampling errors are not averaged out and become much more significant in the final result. The importance of the insertion method for tomographic reconstructions will be discussed in chapter 4.

From a mathematical point of view, it seems to be better to fractionate the exposure into a tilt series and recover even partial 3d information about the FOV, than to record a single projection [65]. This 3d information can for instance greatly facilitate the alignment of objects in crowded and native environments, reduce the problematic effect of preferential orientation of the sample in vitreous ice, and generally improve classification methods by providing a lot more structural information per particle. However, from a practical point of view, things are different. As the tilt increases, the specimen thickness relative to the beam increases, lowering the already low SNR, complicating every other processing step in the workflow [49]. While modern software packages can still reach high resolution reconstructions [15, 45, 66, 67], these high resolution reconstructions are still, for the vast majority, only possible with well behaved, purified and thin samples. Moving beyond these samples will be very hard technically and is likely to require massive changes from the community, both in terms of data availability and software practices.

Chapter 2

Computational solutions to cryoEM

2.1 Software and academic research 30
2.1.1 Technical debt 30
2.1.2 The case for core libraries 31
2.2 Manipulating cryoEM data 32
2.2.1 Multidimensional arrays 32
2.2.2 Two different paradigms for accessing data 34
2.2.3 Generating efficient code using an nd-array syntax 36
2.3 What could a cryoEM core library look like? 39
2.3.1 Core functions and domain-specific pointwise operators 39
2.3.2 Vectorisation and Parallelism 40
2.3.3 Mathematical optimisations and artificial neural networks 44

In this chapter, we discuss the core traits and requirements of modern and future cryoEM image processing. We notably introduce how the elements of multidimensional arrays can be accessed and discuss how something that can appear trivial can in fact determine the core features of a library, from Graphical Processing Unit (GPU) acceleration to automatic differentiation, and how this can ultimately affect the entire technology stack of a project or a field.

2.1 Software and academic research

2.1.1 Technical debt

Testing new ideas and developing new algorithms require different skills and tools than writing production-ready software. For development, programming languages and libraries that facilitate fast development pace are usually chosen, e.g. Python [68] and its large set of scientific libraries [69, 70, 71]. On the other hand, codebases designed to create production-ready software usually prioritise different goals, e.g. scalability, stability or performance. These goals almost always impose design choices that constrain developers in what technology to use and what code to write, which eventually tend to increase complexity and slow down development.

As a result, one could think that having two codebases, one for development and testing, and one for production, is the best choice: developers can quickly test ideas thanks to the limited amount of constraints, and users can still rely on efficient and reliable implementations packaged in a polished interface.

In practice, however, as projects and workflows grow larger, maintaining two different codebases quickly becomes impractical. Instead, projects usually aim to conciliate the best of these two worlds in a single codebase. Designing such codebases, and achieving good balance between user and developer experience, requires careful thinking and, especially in the long term, good software practices [72, 73, 74].

In cryoEM, as in many other academic fields, these efforts are rarely encouraged and rewarded. One reason could be that developing field-specific algorithms and proof-of-concept implementations is rarely distinguished from the act of developing and maintaining the software infrastructure to efficiently share these algorithms to users or to allow developers to test their ideas in the first place.

As a result, many cryoEM projects have accumulated significant technical debt [75] over the

years. Technical debt defines the implied cost of future refactoring required when choosing an easy but limited solution instead of a better approach that could take more time to implement in the first place. As this debt makes future development harder, it tends to accumulate. In order to survive, projects have to invest some of their development efforts in repaying that debt or keeping it at a manageable level. As we have said, cryoEM projects can rarely afford to invest such efforts, and end up accumulating unnecessary complexity, to the point where the cost of maintenance and development can become unmanageable, even for the original authors.

2.1.2 The case for core libraries

In order to reduce the cost of maintenance and development, it is often advisable to rely on generic and popular libraries. When it comes to field-specific tools and algorithms, it is the responsibility of the field to create such well packaged open source libraries [76, 77, 78, 79]. While it may require a significant community effort at first, these libraries are of great value for their field, especially in the long term.

Unfortunately, many cryoEM packages (understandably) choose the easy but limited solution of rewriting their own version of the cryoEM core algorithms, by designing (over-)specialised implementations, perfectly suited for their need. Today, this is one of the main reasons for technical debt in cryoEM software: many projects have to maintain their own copy of the cryoEM core features, which can represent significant or even unrealistic efforts for some teams.

Libraries are an opportunity to share efforts and development time, and improve the overall code quality and feature set [80, 81, 82]. Projects have different requirements and expect different features from such core libraries, but with carefully chosen community design guidelines and technologies, as well as good software practices, we believe that the cryoEM field could develop and rely on these core libraries. This thesis aims to contribute to the existing discussions and standardisation efforts towards the making of such cryoEM core libraries.

Note that while we believe that good software practices, such as standardising research efforts in shared field-specific libraries, could greatly improve the developer and user experience, and already constitute great scientific value, we would also like to emphasise that if the field wants to go beyond classic image processing techniques, further significant changes will be required, especially in terms of how software development is perceived and what it entails in terms of resources and community efforts [83, 84, 85].

2.2 Manipulating cryoEM data

2.2.1 Multidimensional arrays

Multidimensional arrays [69, 86], also referred to as nd-arrays or tensors, are one of the core abstraction cryoEM packages use to store and manipulate data. Nowadays, these abstractions are available in most ecosystems, either as a language or library feature. However, and like a lot of other scientific fields, the foundations of cryoEM software were built when these abstractions were not widely available and when their adoption was not without cost. In this section, we will introduce how modern nd-arrays are *usually* implemented (at a high level), which will lead us to the next section, discussing how multidimensional data is manipulated through these arrays and what it entails in terms of technology stack, memory usage, runtime performance and overall feature-set.

An nd-array consists of a contiguous one-dimensional segment of computer memory, combined with an indexing scheme that maps an nd-sequence of integers, aka indices, into the location of an item in the memory segment. The ranges in which the indices can vary is specified by the shape of the array. How many bytes each item takes and how the bytes are accessed and interpreted is defined by the item-type, also known as data-type, associated with the array.

The data-type is often restricted to one of the primitive types, e.g. 32-bit integer or floating-

point [69], but it can technically be any (user-defined) type. While the type itself defines how bytes should be interpreted and manipulated, when it comes to memory, the type is reduced to only two simple properties: a size and an alignment requirement. The size is simply the number of bytes necessary to store a value of that type. The alignment requirement is the non-negative integer value representing the number of bytes between successive memory addresses at which objects of this type can be allocated. While it is easy to understand that the size of the data-type directly affects the amount of memory necessary to store the nd-array, the consequences of the data-type's alignment requirements are more difficult to grasp without some understanding of memory systems. A summarised introduction of the subject will be provided in section 2.3.2 in the context of vectorised instruction sets [87], but for now it is sufficient to know that the size and alignment requirement of a type both define how, and more specifically how efficiently, the underlying bytes of a type can be loaded from, and stored to, Random-Access Memory (RAM).

The indexing scheme relates the logical nd-indices to a single integer value, often called memory offset. This offset is relative to the beginning of the 1-dimensional (1d) memory segment where the data is stored and thus directly points to the item corresponding to these indices. The implementation of the indexing scheme varies from one library to another, but the most flexible approach for dense (as opposed to sparse) nd-arrays is to attach each dimension to a stride [69, 86]. A stride is a single integer value that sets the (constant) memory offset between items along a particular dimension of the array. To know the memory offset of an item at a particular index along a dimension, we simply need to multiply the stride of that dimension with the index. In the multidimensional case, we simply need to add the memory offset of each dimension. Strides provide an efficient way to access items in the nd-array, but their main advantage is that they allow complex shape manipulations of the array without requiring to modify or copy the underlying data. For instance, arrays can be easily reshaped, subregions can be selected and extracted into separate arrays, dimensions can be accessed in reverse order by simply negating their stride, and dimensions can be

broadcasted, an operation during which the stride of a dimension is set to 0, resulting in the dimension (of arbitrary size) being represented by a single item in memory.

The 1d contiguous buffer, the data-type, the shape, the strides and the number of dimensions are the core parameters that are often chosen to encode a dense nd-array. Libraries differ in which parameters they decide to expose and the degree of freedom they give developers to manipulate these parameters [69, 88, 89]. Fixing (one of) these parameters at compile-time may allow more optimisations resulting in more efficient code being produced, but this is generally at the cost of flexibility and is ultimately a design choice the library has to make.

2.2.2 Two different paradigms for accessing data

Once the data is stored, we need to be able to access, *i.e.* read from and write to, the elements of the array very efficiently. Most programming languages provide efficient for-loops that can be used to iterate through every element of an array. While these loops are powerful constructs, there are many advantages in trying to abstract them away from the user interface, including benefits in safety, maintenance, performance, scalability and hardware support. These abstractions can be classified into two main paradigms, the pointwise paradigm and array paradigm. Languages and libraries capable of manipulating nd-arrays usually provide at least one of them.

The pointwise paradigm requires developers to think "per element" [90, 91, 92]. Pointwise Application Programming Interfaces (APIs) are the closest abstractions to the loop construct and let developers define the "body" of the abstracted loop, which is thus called for every element in the nd-array. This "body" is called *operator* and it can be any function or callable type, so it is not surprising that these APIs are the most flexible from a user perspective. Languages and libraries have different naming conventions for such APIs, but one of the most well known pointwise functions is the "map" function (C++ calls it `std::transform`). Interestingly, GPU programming is greatly invested in this paradigm [93, 94], and GPU kernels and shaders all operate on a per-index basis (this is further discussed in section 2.3.2).

The array paradigm [69] requires developers to think at the level of the nd-array, and in this case, the API and the operators are designed to be applied to the nd-arrays directly (as opposed to their elements). Syntactically speaking, this paradigm is often the simplest way to manipulate multidimensional data, and if programming languages and/or libraries come with robust array syntaxes and APIs, it can be as flexible as pointwise syntaxes and APIs. Interestingly, library implementations of array approaches almost always rely internally on pointwise abstractions, and can thus be thought of as a higher level abstraction built on top of the pointwise abstraction.

With a pointwise approach, developers implement algorithms by directly manipulating the values of the array. If the data-type of these values is a primitive type, this ultimately relies on the core API and operators provided by the programming language and standard library (e.g. `f32 + f32`). This sequence of operation is then encapsulated in a new operator that can be passed to a pointwise API, like `map()`. In the case of compiled languages, compilers can optimise these operators as any other piece of code.

With array approaches, developers implement algorithms by directly manipulating the nd-arrays, as if the nd-array was a primitive type. These array-libraries effectively create a Domain-Specific Language (DSL) made of core functions/operators (usually a few hundreds or thousands) which are designed to operate on the arrays (e.g. `Array + Array`). The fact that a predefined set of operators is now enough to express a wide range of algorithms, and given that this predefined set of operators is fully implemented and controlled by the nd-array libraries, is fundamental to understand the key feature differences between pointwise and array approaches. First, having to only support a limited set of predefined operators allows array approaches to separate, across languages, the user interface from their implementations/backends. These libraries can indeed use programming languages like Python [68], to implement concise, easy to use and safe interfaces, while still relying on a more verbose statically-typed compiled language to generate efficient code. Second, it becomes much easier to pack additional functionalities into these operators. For instance,

the derivative of every operator in the DSL can be predefined by the library. Then, as long as the data is only manipulated through the DSL, libraries can track what happens to the data to implement Automatic (or Algorithmic) Differentiation (AD) [95], allowing them to accurately and efficiently differentiate user-defined functions automatically.

2.2.3 Generating efficient code using an nd-array syntax

Generating performant code from the array syntax is much more challenging compared to programs written using the pointwise syntax. The array syntax most notably leads to more loop constructs (which is particularly relevant for GPUs), as well as more and larger temporaries (which are now multidimensional arrays), resulting in worse performance and an increased memory footprint.

Fortunately, on top of supporting the efficient and practical pointwise approach, statically-typed compiled languages can still generate performant code from an array syntax. Indeed, the syntax abstractions are fully transparent to the compiler, so operators can be directly inlined, and thanks to lazy-evaluation [96, 97, 98], some loops can be fused together and temporaries can be almost entirely eliminated. Lazy-evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed, effectively allowing programmers to nest operators together (in the functional programming sense) resulting in an Abstract Syntax Tree (AST), or graph, which encodes what will happen to the data when the nested expression is evaluated. In C++, these expressions are called expression templates [99]. Note that the evaluation of the graph is entirely controlled by the type system (and therefore by the programmer) and is not the result of a compiler optimisation. Lazy-evaluation is an incredibly powerful technique, both in the context of array and pointwise approaches, and can in theory lead to overall better performance due to compilers and libraries having more optimisation opportunities due to the delayed execution. xtensor [100] and Eigen [89] are two good C++ examples demonstrating the benefit of lazy-evaluation for the array syntax. The C++26 execution proposal [98] does the same thing but for pointwise

approaches.

In cases where the user interface is done in a dynamically-typed interpreted programming language like Python, things are different. Indeed, while these hybrid approaches are an effective way to build high performance libraries, building them is much more complicated. In order to generate better code from the (only available) array syntax, these libraries are using a similar graph-based approach that the one described above for statically-typed compiled languages. In this case, however, it is the responsibility of these hybrid libraries to construct, compile and optimise the graph [101, 102]. In the case of TensorFlow [103] and Pytorch [104], this requires low-level understanding of the internals of CPython [105], requires knowledge of C++ (thereby undermining one of the original goals of using Python in the first place), makes it difficult and expensive to evolve large frameworks, results in more complex ecosystems, and pushes the world towards graph-based approaches, which are more complex. Significant standardisation efforts are currently invested to simplify this process (e.g. Medium-Level Intermediary Representation (MLIR) [106]) and the future will tell if the industry will succeed to migrate to these new frameworks. In the meantime, with the current established technologies, only a few libraries (and companies) can afford having to implement entire compilers and optimisers for their operators. Importantly, in order to preserve the (already so costly) usability of these dynamically evaluated graph-based approaches, lazy-evaluation is mostly reduced to an optimisation step, and it is unclear how far these optimisations, like the promising `torch.compile()` feature of PyTorch 2.0, will be able to go. At the time of writing, these optimisations are mostly reduced to the (kernel) fusion of some operators and even simple temporaries, e.g. mesh-grids of coordinates, cannot be reliably eluded. While we expect that these optimisations will improve as the technology matures, given that these temporaries (which only exist because of the array syntax) can reach gigabytes in size, we may end up in a situation where some algorithms may or may not be able to run on a particular device, purely based on some optimisations that are (by design) almost completely opaque to most developers.

Recognising that it is not always easy or possible to efficiently express some operations as a composition of existing array operators, libraries like NumPy [69], TensorFlow [103] or Pytorch [104], offer ways for developers to create their own domain-specific pointwise operators (NumPy calls them `ufuncs`). These operators are usually written in statically-typed compiled languages, which includes the language in which these libraries are implemented, *i.e.* C for NumPy (which also supports Fortran) or C++ for TensorFlow and PyTorch. DSL, like Numba [107] or Triton [91], were also developed to this effect and offer a way to write efficient pointwise code directly from Python. Unfortunately, while writing pointwise operators can offer significant performance improvements, the functionalities only implemented through the array syntax, like automatic differentiation, are lost. Moreover, it is unclear how these custom operators will be parsed by future compilers and whether they could lead to worse performance in the future, as a result of being more opaque to the graph optimiser compared to the native operators.

Evidently, the field is evolving at an unprecedented pace, and it is to our opinion that the current technologies do not provide an efficient and elegant solution when it comes to manipulating multidimensional data, especially when considering the need for parallel computing and automatic differentiation, which are both more and more relevant for image processing and cryoEM. This is particularly true for the "hybrid" Python deep-learning libraries, which despite their incredible machinery (whose complexity is admittedly spiralling out of control), offer limited options and paradigms. Many projects are trying to solve these issues or offer better alternatives [108, 109, 110, 111, 112, 113], but maybe central to those is the apparition of new MLIRs [106, 114]. These are sitting between the unoptimised AST emitted by compilers or libraries (e.g. TensorFlow or PyTorch) and the final code representation (e.g. LLVM-IR [115]), and are designed to express modern data-flows and perform complex program analysis. TensorFlow and PyTorch have already started working towards supporting these, but note that MLIR reaches beyond deep-learning frameworks and offers a novel approach for building extensible and reusable compiler architectures. These

standardisation efforts were long-awaited by the community and as these projects mature, we hope that new efficient and elegant technologies will rise and lead the way toward more reasonable solutions for developers and end-users.

2.3 What could a cryoEM core library look like?

2.3.1 Core functions and domain-specific pointwise operators

For performance critical applications, like most cryoEM applications, the ability to write pointwise operators is often a requirement. The central role of geometric transformations in cryoEM is a good example to illustrate this point.

One of the most essential pieces of cryoEM software is indeed the ability to perform geometric transformations, e.g. rotation/translation/scale, wrappings or projections. These transformations are performed so often that very little concessions can be made before significantly degrading the performance of the application. Ideally, cryoEM software should have full control over the coordinate system and geometric transformations, *i.e.* rotation or affine matrices and quaternions. Interpolation and border/addressing methods should be defined by the cryoEM core libraries to ensure predictable performance and accuracy, but also to support a wide range of interpolation methods that are useful in cryoEM, e.g. cubic B-spline interpolation, windowed-sinc interpolation or hardware interpolation using GPU textures, which are all rarely supported in generic scientific libraries. Data with arbitrary strides should also be supported to handle a wider range of scenarios and no (big) temporaries should be dynamically allocated. As with most of the cryoEM operations, these transformations should support a wide range of hardware accelerations (see next section).

This level of flexibility and performance is very difficult to implement (or maybe impossible) using only the array syntax, and while libraries like PyTorch [104] provide array APIs for geometric transformations, their lack of flexibility quickly makes them unsuitable for most

performance critical applications. On the other hand, the pointwise syntax is much more flexible and allows developers to easily create their own transformations kernels (that includes coordinate transformation, interpolation and other per-element logics) as a single operation that only needs to loop through the input/output arrays once and does not require any temporary array. Moreover, the pointwise syntax preserves the ability to take different code branches based on a per-element logic (logic which can be predicted by CPUs [116]) thereby allowing to skip unnecessary and expensive operations. This return-early pattern is extensively used in cryoEM. For instance, when manipulating discrete Fourier spectrums (as it is routinely done in cryoEM), frequencies above Nyquist are often skipped. These high frequencies represent about 20% and 50% of the total amount of frequencies for 2d and 3d spectrums, respectively. For large spectrums, this can mean that millions of elements can be saved from expensive transformations.

Fortunately, when it comes to cryoEM, there's only 4 pointwise core functionalities we mostly care about: map, sort, reduce and scan. Interestingly, map, also known as "for-each", can be used to implement the vast majority of cryoEM, including the aforementioned geometric transformations, but also most of its image and signal processing algorithms. Given the importance of these core functions for cryoEM, we believe that cryoEM core libraries should not only provide an interface to these functions, but also fully control and maintain their implementations. While this does not exclude relying on generic performant libraries like Eigen [89] or Kokkos [88], it seems that to use certain functionalities (e.g. hardware interpolation from GPU textures), providing our own implementations could also be valuable.

2.3.2 Vectorisation and Parallelism

In order to achieve good performance, the pointwise core functions should make good use of the available hardware, which often involves good use of vertorised instructions and threads. The goal of this section is not to discuss these admittedly complex subjects, so instead,

we will focus on the practicalities from the developer's perspective. Note that we only consider the type of hardware used for general and high-performance computing (HPC) [117]. Moreover, we focus on applications that only rely on the (offline) compute pipelines of GPUs, which is sometimes referred to as General-Purpose Computing on GPU (GPGPU) [118].

Processors, e.g. Core Processing Units (CPUs) or GPUs, constantly need to access data stored in RAM to put them in registers [119]. Registers are the places in the chip where data can be manipulated according to various instructions (e.g. add the two single-precision floating-points stored in the 32-bit register A and B, and store the value in register C). Accessing RAM is expensive, and data caches [120], *i.e.* the places where data can be more efficiently transferred from and to registers, are used to reduce the number of necessary transactions to RAM. Accessing data in a local and predictable manner, so that processors can prefetch and efficiently store data in caches, is of the utmost importance to achieve good performance on modern hardware. While these caches complexify the way memory addresses are encoded, programming languages and processors mostly hide this complexity away so that developers can reason in terms of a unique address space, where each addressable byte in this space has a unique memory address. Memory addresses are essentially defined by a single integer value m, and are said to be n-byte aligned when m is a multiple of n (where n is a power of 2). The alignment requirement r (see section 2.2.1) of a type says that objects of this type are to be stored at addresses that are (at least) r-byte aligned.

Modern processors are designed in such a way [121] that their memory subsystem access memory at a certain alignment granularity. This has many consequences, one of which being that memory addresses are accessed differently depending on their alignments. More specifically, data located at addresses aligned to a lower value than the system granularity, *i.e.* misaligned data, are more complex and slower to load. To prevent this, programming languages and compilers often ensure that the alignment requirement of types is large enough to be efficiently accessed while keeping the type size to a minimum. The alignment

requirement can also be set programmatically, usually to a larger value than this default value (giving rise to "over-aligned types"), to specifically target vertorised instructions and registers so that contiguous data can be processed more efficiently. Interestingly, and at the cost of increasing the type size, languages and/or compilers often enforce that the type size is larger than, and a multiple of, the alignment requirement. This is to ensure that items stored next to each other in an array all satisfy their alignment requirement.

From the programmer's perspective, vertorised instructions operate on (virtual) aggregate-types [23, 122, 123]. These types are made of multiple items of the same primitive type aggregated in a contiguous array, have an alignment equal to the aggregate size, *i.e.* they are "over-aligned", and perfectly fit in the vertorised registers. Once the data is loaded in these vertorised registers, it can be manipulated using vectorised instructions (e.g. add the two vectors, element-wise, each containing 4 single-precision floating-points stored in the 128-bit register A and B, and store the resulting vector in the 128-bit register C). This allows multiple items to be manipulated simultaneously, within the same instruction: this is referred to as Single Instruction Multiple Data (SIMD). While these vertorised operations are mostly known for CPUs, GPUs also support some of them, usually for loading and storing data from and to memory. Explicit vertorisation can be achieved by abstracting away primitive types into these larger aggregate types, which are often made available by SIMD or standard libraries, or by compilers. Thanks to metaprogramming, programmers can also write pointwise operators that are generic enough for them to be compatible with both primitive and aggregate types. Note that while writing explicit SIMD code can be a good solution and generic enough to support different instruction sets [122], compilers have become extremely good at automatically vectorising code [124, 125]. Developers can enhance the chance of automatic vertorisation by accessing data in a predictable manner (e.g. fixed strides known at compile time) and by using well-aligned types.

On top of having instructions that can process multiple data at once, processors also have multiple threads (of execution), which offer a scalable way to process a large amount of

data in parallel [126]. In the context of CPUs, threads are essentially a programming construct; they are created, managed and scheduled by the operating system, and offer a way for programmers to better saturate the CPU resources. One important property is that threads share memory resources, including some memory caches. As a result, when writing multithreaded code for CPUs, developers often try to make threads don't access neighbouring data, but instead work on their own (large) chunk of data. This leads to a much more performant and scalable paradigm, where threads don't collide as much for accessing data in cache (a well-known phenomenon called false-sharing) and where SIMD instructions can still be used in multithreaded contexts.

GPUs use a different architecture, called Single Instruction Multiple Threads (SIMT), to process multiple elements in a single instruction. In this case, the processor directly creates, manages, schedules, and executes a group of parallel threads. In CUDA [127], this group is called a warp and contains 32 threads. As opposed to SIMD instructions which operate on the SIMD aggregate-type (thereby exposing the architecture to the software), SIMT instructions specify the operations of a single thread (and not to the group of threads). This allows programmers to mostly forget that the threads are grouped together and effectively write "single-threaded code" (e.g. pointwise operators working on primitive types) that can run in massively parallel contexts. Note that in order to make good use of the SIMT instructions (and similar to single-threaded non-SIMD code), programmers should make sure that threads within the same group access neighbouring data.

CUDA is the most popular GPGPU platform in cryoEM, by far. Projects access CUDA either from CUDA-C++, MATLAB (and its MEX files [128]), or TensorFlow [103] / Pytorch [104] (directly from Python or more rarely from C++). While we do expect other frameworks to gain popularity as GPUs from various manufacturers become more and more performant, and as parallel programming is progressively being standardised into programming languages and compilers [98, 91], CUDA is still likely going to remain leaps ahead of the competition for many years. It may also be worth emphasising that CUDA is designed around the C++

ecosystem, and that its compilers can directly ingest most C++ code and do not rely on a shading language (at least in the context of offline computing). Other frameworks, like AMD's ROCm [129], oneAPI [130], or the higher-level SYCL model [131], have followed the same strategy.

For maintainability and scalability reasons, these different paradigms and architectures should be kept hidden from user space as much as possible and be confined to the implementation of the aforementioned core functions. For our purposes, this mostly means having specialised ways to traverse nd-arrays, e.g. a single-threaded way and a multithreaded way, all accessible from a simple and implementation-agnostic API.

2.3.3 Mathematical optimisations and artificial neural networks

Very quickly in cryoEM pipelines, the optimisation problem, *i.e.* mapping inputs to outputs, becomes apparent. Derivative-less optimisations offer a good enough approach for cases with a small number of inputs and for black-box functions. However, as the number of inputs increases (sometimes as little as two scalar inputs), these optimisations end up being impractical and derivatives become a requirement. Computing derivatives by hand, or using numerical differentiation, e.g. central finite difference, can be good enough for some scenarios, but they tend to be error-prone and quickly become unmanageable, unreliable and extremely expensive to compute. As the functions to optimise become more complex, fast and accurate derivatives become more of a necessity.

Automatic (or Algorithmic) Differentiation (AD) techniques [95] provide effective solutions to compute derivatives, which can then be ingested by optimisation algorithms (e.g. L-BFGS [132], gradient descent [133], etc.). AD refers to the set of techniques that compute derivatives through accumulation of values during code execution, with the goal to generate numerical derivative evaluations rather than derivative expressions (see symbolic differentiation [134]).

AD can be implemented through a DSL, where every single operation inside the DSL is explicitly defined to be differentiable. This is the approach taken by TensorFlow [103] or PyTorch [104]. This approach requires developers to rewrite their implementations using exclusively the operations defined by the DSL, which therefore has to support as much operations from the original code as possible. If these DSLs don't exactly match the original code or use different paradigms (e.g. pointwise *vs* array), rewriting algorithms can be an expensive operation and does not always lead to the same performance.

Operator overloading is another approach to implement AD, and aims to differentiate more arbitrary code. This is the approach taken by autodiff [135], Adept [136] or JAX [137]. Instead of implementing entirely new languages, operator overloading replaces core language constructs like primitive data-types or functions (e.g. in Adept, `double` becomes `adouble`). While it tends to be easier to add support for these library types and functions, some code changes are still usually required, and as with the DSL approach, all operations have to be performed inside the library's API, which drastically limits the use of external libraries. Maybe more importantly, operator overloading often leads to worse performance compared to the DSL approach.

The third approach is source-rewriting, which aims to make the original language differentiable. Tools like Tapenade [138] can statically analyse programs and produce new source code with the corresponding gradients, which is then compiled/interpreted by the original language's framework. This approach can lead to well optimised code and often doesn't require any rewriting. However, maintaining these tools is very expensive since they have to be able to parse entire languages and keep up with new releases. In practice, tools like Tapenade only end up supporting a very specific subset of C-code. Moreover, they are also assuming that source code is available, which is not always possible when relying on external libraries.

However, a fourth more recent and more powerful approach was recently developed, and

could progressively take over to become the new standard for AD. Instead of trying to differentiate unoptimised source code (which all three aforementioned approaches do), this new approach differentiates already optimised code. Enzyme [139] has shown that doing so exposes new optimisation opportunities and leads to more performant code, or similar code at worst. Moreover, since the AD is only performed after initial optimisation passes, it only needs to ingest some kind of Intermediary Representation (IR), like LLVM-IR [115], which greatly facilitates maintenance and allows tools like Enzyme to automatically support a wide range of programming languages, compilers, deep-learning frameworks and hardware.

In conclusion, and as of time of writing, supporting AD through something other than Python's deep-learning libraries seems impractical. As we have already discussed in section 2.2.3, the Python interfaces of these libraries are currently limited to an array syntax, which is far from ideal performance-wise. We hope that new technologies will be able to lift this restriction in the future, but in the meantime, it seems that the safest recommendation for cryoEM is to use these libraries, and add pointwise operators (and their derivatives) when/if performance becomes an issue. Writing such operators is often done directly in C++ and C++-based DSLs like CUDA [127], HIP [129], OpenACC [140], or C++ programming models like SYCL [131]. Note that some C++ libraries, like Kokkos [88], are already providing a unified infrastructure to all of these technologies and C++26 execution contexts should further simplify the user experience. Python-based DSLs, like Numba [107] or Triton [91], are also an effective solution and may offer less fragmented approaches if the entire codebase is already in Python. Note, however, that these are not as established and as supported, and it may be wiser for projects that value stability, longtime support and flexibility to choose C++.

Chapter 3

An experimental cryoEM core library

3.1 Goals and design choices 49
3.1.1 Choosing a programming language 49
3.1.2 Performance and simplicity 50
3.1.3 Device agnostic code 50
3.2 Execution model 51
3.2.1 Devices, memory resources and streams 51
3.2.2 Asynchronous eager execution 54
3.2.3 CPU support 56
3.2.4 GPU support 57
3.3 Multidimensional data 64
3.3.1 BDHW order 64
3.3.2 Strides and memory layout 65
3.3.3 Array and View 66
3.3.4 Accessor and (md)span 67
3.4 Core functions 72
3.4.1 Index-wise 73
3.4.2 Element-wise 74
3.4.3 Reductions 76
3.5 cryoEM specificities 80

3.5.1 Geometric transformations 80
3.5.2 FFT layouts 83

In this chapter we present the core library that was developed during this project and discuss the design choices we have made. This library is still under heavy development and we would not recommend using it in production at this stage. The source code is available at https://github.com/thomasfrosio/noa.

3.1 Goals and design choices

One of the original goals when this project started was to study and collect the cryoEM core algorithms and gather them in a documented and tested library. One of our hopes was to then use this library to facilitate future software developments in our team.

3.1.1 Choosing a programming language

In the previous chapter, we recommended new projects to use Python and its scientific libraries (notably PyTorch [104]) as the main development ecosystem. However, we have not chosen this path, for three main reasons.

First, this recommendation is the result of almost 4 years of work in the field, and when we started, Python was not an obvious choice to us. PyTorch was actually one of the first libraries we experimented with to implement a cryoEM core library. Unfortunately, our benchmarks were not as good as we had hoped, and since we were working with large Cryo Electron Tomography (cryoET) datasets at the time, performance was one of our main criterions, so we quickly tried other options. Second, when we started, most if not all of the algorithms in cryoEM were written in a pointwise manner (C-style for-loops, pointer indexing and pointwise logic), so it seemed more logical to us to use a programming language that could naturally allow this style of programming. Third, learning and becoming familiar with the C++ ecosystem appeared useful, especially given that the existing Python or MATLAB projects were (and still are) relying on C++ code to accelerate certain sections of their algorithms.

When we started this project, having a fractured development ecosystem between C++ and Python did not seem to be a great idea, especially given that we had little to no experience in software development. Since using C++ was inevitable, we decided to fully embrace it and use it as our main programming language for this project. The project first started using C++17, but we are progressively migrating to C++20.

3.1.2 Performance and simplicity

One of the early goals of this project was for existing codebases to be able to use this library. As such, we had made performance and GPU support (which was lacking at the time) one of our main goals. While this is still true today, as the project grew, code simplicity became more and more important. Writing simple code is indeed of the utmost importance to us, since we are well aware that we will not afford to maintain a complex codebase for long.

While this is not an exact science, we followed two principles that have helped us to write both performant and simple code. First, when (and only when) obvious patterns appear, abstract them away to reuse as much code as possible. Second, always prefer less code to perfectly optimised code. This last rule should only be broken if benchmarks show that significant performance improvements could be achieved with more complex code, but even then, the price of this complexity and the inevitable technical debt it will create should be kept in mind.

3.1.3 Device agnostic code

Device-agnostic code is a term often used by PyTorch developers, but it is found in other areas of programming. Code that is generic enough to run on different devices, e.g. CPU and GPU, without the need for special adaptations is said to be device-agnostic. This model is directly aligned with the idea of keeping code simple and from the early stages of the project we wanted to keep the user space device-agnostic.

In practice, this means that the library API should allow developers to define, at runtime, the device on which to execute the code. This also means that the user-space, which includes user-defined operators, should not directly interact with the library internals and that these internals should instead be kept hidden into backends to separate user and library space as much as possible.

3.2 Execution model

3.2.1 Devices, memory resources and streams

`Device` is the main object to refer to and to manipulate compute-devices. The library currently supports two types of devices, CPUs and GPUs, and each device type is internally handled by a backend. `Device` contains a unique identifier, which is simply an integer used to uniquely refer to a compute device. While the `Device` type can be used to reset internal resources (via `Device::reset()`) and enforce global synchronisation barriers (via `Device::synchronize()`), it is mostly used as a "pointer" to a particular compute device, e.g. to allocate an array on that specific device, and to query simple information about the compute context, e.g. to list of GPUs available in the system.

`Devices` only refer to the high level hardware that can execute code, e.g. the CPU device or the GPU devices; it does not refer to the different memory resources available on the system. `Allocator` is the type used to describe different memory resources, e.g. "private", "unified" or "pinned" [141]. Each memory resource has its own set of advantages and disadvantages and users should be aware of them to ensure correctness and better performance.

`Streams` are similar to CUDA streams [141], *i.e.* they are (asynchronous) dispatch queues and are attached to a device. For the CPU, `Stream` defaults to the current thread, thus all execution is by default synchronous. An asynchronous stream can be created too, which redirects works to a thread managed by the stream. Importantly, the library defines a per-thread and per-device stream, called the "current stream". While users rarely need to pay attention to it, the current streams play a major role in how the library works, notably how operators are sent to the backends (see code snippet 3.1).

`Device` and `Stream` are two fundamental types of the unified API of the library and both interfaces with the corresponding backend types, e.g. `noa::cuda::Device` and `noa::cuda::Stream`, and uses some kind of type-erasure technique (e.g. `std::variant`) [142] to stay

backend- and device-agnostic.

```
using namespace ::noa::types; // import Device and Stream

// By default:
// - the CPU current stream is the current thread.
// - the GPU current stream is the NULL stream
//   (at least in CUDA).
auto device = Device("gpu:0");
auto my_stream = Stream(device);

// (Re)set the current stream for "gpu:0".
// This can be changed at any time.
Stream::set_current(my_stream);

// After this point, the library will use "my_stream"
// as the asynchronous dispatch queue for arrays using
// the "gpu:0" device. Note that Stream is reference
// counted and Stream::set_current() keeps a copy of
// the stream.

// The library can indeed retrieve the current stream of
// any device at any point and dispatch work on that
// stream:
{
    // Library frontend (only a few functions actually
    // dispatch work to backends):
    Stream& current_stream = Stream::current(device);
    if (device.is_cpu()) {
        // This can expose backend specific code
```

```
29          current_stream.cpu().enqueue(...);
30      } else if (device.is_gpu()) {
31          // This can expose backend specific code
32          current_stream.gpu().enqueue(...);
33      }
34  }
35
36  // Users can also temporarily change the current stream
37  // using StreamGuard. StreamGuard derives from Stream,
38  // but:
39  // 1. Its constructor captures a reference of the current
40  //    stream.
41  // 2. It then sets itself as the current stream.
42  // 3. Its destructor synchronises the queue making sure
43  //    work is completed (like Stream), and then resets
44  //    the captured stream back to the current stream.
45  {
46      StreamGuard new_stream(device, Stream::ASYNC);
47      // ... following work will on "new_stream" ...
48  }
49  // "new_stream" is destroyed, "my_stream" is back to
50  // being the current stream
```

Snippet 3.1: Devices and Streams

As we will see, nd-arrays are not attached to a stream, only to a device (`Device`) and a memory resource (`Allocator`). This is important because arrays should be shareable between streams of the same device. For instance, an image could be copied by one stream and transformed (out-of-place) by another stream. Since they don't depend on each other, users can then use two different streams to schedule these two operations to run in parallel.

3.2.2 Asynchronous eager execution

The library currently only supports an (asynchronous) eager execution model, that is, functions are dispatched to the underlying hardware as soon as possible. Depending on the device and the stream, these functions are executed asynchronously on the underlying hardware, meaning that the program execution is not blocked and that functions may return before completion. One exception is for functions that return scalars, for instance, `noa::sum(const Array<f64>&)->f64`. In this case, the current stream is synchronised before the function returns. Note that alternative functions that don't synchronise by returning array(s) instead of scalars are often available.

```
// import i64, f64, Shape, Array, ArrayOption
using namespace ::noa::types;

// Allocate an array of 256x256x256 double-precision
// floating-points. Then enqueue the Randomize<Uniform>
// operator to initialise the values with random numbers
// with an uniform distribution, between -1 and 1.
const auto options = ArrayOption{}.set_device("cuda:0");
const auto shape = Shape4<i64>{1, 256, 256, 256};
const Array array = noa::random<f64>(
    noa::Uniform{-1., 1.}, shape, options);
// Note the use of class-template argument deduction
// (CTAD) to simplify Array<f64> to Array.

// At this point, "array" should be seen as a promise:
// its elements may not be initialised or even allocated
// yet. The library simply enqueues operators to the
// current stream of the device cuda:0 and the device
```

```
// executes these operators eagerly.

// Then compute the sum of the elements of that
// randomised array. Internally, the library does
// Stream::current(array.device()) to query the relevant
// stream and then enqueues the ReduceAccurateSum
// operator. Note that no synchronisations have been
// performed so far. However, this function returns a
// scalar, so it must wait that all work is completed
// before returning. When this function returns, the
// current stream is synchronised.
const f64 sum_0 = noa::sum(array);

// Then compute the sum again, but this time using the
// overload that returns an array. (Note that it also
// exists an overload that takes an existing output array).
// Since this function returns an array, as opposed to a
// scalar, it doesn't need to synchronise the stream
// before returning.
const Array sum_1 = noa::sum(array, ReduceAxes::all());

// All axes are reduced, so sum_1 is an array with a
// single element. To access this element, we should
// first synchronise the stream to make sure the
// device is done computing the sum. Array::eval()
// does just that and is equivalent to
// Stream::current(sum_1.device()).synchronize().
const f64 sum_2 = sum_1.eval().span()[0];
// .span() is one of the many ways to efficiently
// access values from Arrays.
```

```
49
50 assert(noa::allclose(sum_0, sum_2));
```

Snippet 3.2: Asynchronous eager execution

With this API, it is possible to write programs that rarely need to synchronise execution. This approach scales extremely well with GPUs since the CPU time spent launching GPU kernels can now be overlapped with GPU compute.

In the previous chapter, we have introduced lazy-evaluation as another mode of execution. This mode is currently not implemented in this library since we originally preferred the simplicity of the "eager-mode". However, nothing would technically prevent us from implementing a lazy API in the future.

3.2.3 CPU support

The library currently has one CPU backend, which is simply the "native" architecture (e.g. x86-64 or arm64). The entire backend is templated and almost entirely exposed in public headers, so that the library internals can be inlined with the user code, and the entire program can get optimised nicely without any link-time optimisation (aka interprocedural optimisation [143]) involved. The CPU stream is synchronous by default, pushing all work to the current thread without any runtime overhead. An asynchronous mode is also provided, in which case tasks are wrapped in a `std::function` [144] and sent to the thread managed by the stream.

As of time of writing, no explicit SIMD vectorisation was implemented. Instead, code is written to help the optimizer automatically generate these vectorisations. The element-wise and reduction core functions are, for instance, checking for the 1d and contiguous cases and explicitly generating code for these cases. This results in slightly longer compile time and binary sizes since both the generic case and the 1d contiguous case have to be generated,

but given that library template functions are only instantiated on demand, this has not appeared to be an issue so far. Multithreading is done using OpenMP [145] and each core function has a multithreaded implementation. The backend implementations allow to turn multithreading off directly at the call site, to keep the amount of generated code to a minimum. The current strategy is to only run the multithreaded implementation for very large arrays, but we have not really explored this area in depth and things are likely to change in the future.

3.2.4 GPU support

As of time of writing this thesis, the library has a single GPU backend, the CUDA backend. CUDA-C++ [127] is a DSL so similar to C++ that developers can easily write code that can be compiled to both C++ and CUDA-C++. We extensively use this feature and most library operators have a single implementation which is used by both the CPU and the CUDA backends. Unfortunately, while most code can be abstracted to native C++, CUDA kernels cannot be compiled and launched from C++ source files. We currently see three different strategies to deal with this problem.

The most popular strategy is to compile the CUDA backend (and its `.cu` files) when building the library. As a result, the applications only need to link the library as part of their build, and can still benefit from the GPU acceleration without having to compile any CUDA code directly. This approach is one of the most common ways to use CUDA, especially for projects that only use GPUs transitively in their programs. Unfortunately, this approach has a big disadvantage: it does not scale well. Indeed, while CUDA kernels can be templated, in this scenario the library needs to explicitly instantiate these templates with all of the types and type combinations it aims to support, as opposed to compiling the template kernels on a per-application basis. For libraries that intend to support a wide range of types and functions, this often means having to compile thousands of kernels, most of which will not be used by the application. Compiling this much code quickly becomes impractical and leads

to unreasonable compile times (minutes to hours), large binary sizes (hundreds of Mbytes to Gbytes) and slow loading times (more than 30 seconds). Maybe more importantly, this early compilation stage prevents us from easily integrating user-defined types and operators.

We originally chose this option almost by default, and it took us a long time to realise that this was not sustainable. Fortunately, there are two other ways to use CUDA from C++.

Indeed, the library and its CUDA backend could be fully templated and header-only, thereby leaving the burden of compilation to the applications. By including the library, users would thus also include the CUDA kernel definitions of the library. However, as we have previously said, CUDA kernels cannot be compiled in C++ translation units, meaning that in order to compile and launch CUDA kernels, the application source files need to be compiled as CUDA (`.cu`) files using a CUDA-capable compiler (usually nvcc [146]). This option has the advantage that only the kernels necessary to the application would get compiled and allow users to easily pass their own types and operators to the GPU. nvc++ [147], the new CUDA High-Performance Computing (HPC) compiler, probably offers the seamless integration between C++ and CUDA-C++ code, allowing for instance to pass lambdas defined in normal C++ directly to the device (see code snippet 3.3), whereas nvcc need the additional `__device__` attribute. Libraries like Kokkos [88] have chosen this approach, resulting in every C++ file from the user being compiled as a CUDA file. The disadvantage of this approach is that all C++ files calling the library need to be compiled by the CUDA compiler, which is a big requirement for some projects. While nvcc and nvc++ are great compilers, we may still prefer to compile our code (that includes user and library code) with more widely used C++ compilers, like g++ [148] and clang++ [149]. Since we didn't want to impose our compiler on the users, we did not choose this option at first. However, recognising its practicality and ability to write elegant C++ code that runs on GPUs, we are migrating to including this option to the library.

```
// -- my_application.cpp --

#include <noa/Array.hpp>
#include <noa/Ewise.hpp>
#include "my_application_header.hpp"

struct my_type { double value; };

noa::Array array = noa::random<double>(
    noa::Uniform{-1., 1.}, 1024, {.device="cuda"});
noa::Array output = noa::like<my_type>(array);
double random = noa::random_value<double>(noa::Uniform{0., 1.});

// Lambdas can be passed directly and transparently
// to the GPU kernel! This is because this file is treated,
// despite its .cpp extension, as a CUDA file (.cu) and
// compiled by a CUDA-capable compiler (nvc++). As a result,
// the library, in this case the noa/Ewise.hpp header, can
// include the templated kernels in this translation unit.
noa::ewise(array, output,
           [random] (double value, my_type& out) {
               out.value = value > 0 ?
                   std::sqrt(value) : random;
           });
```

Snippet 3.3: User-defined type and operator to the CUDA backend using nvc++.

A Just-In-Time (JIT) compilation mode of the CUDA backend was also recently added to the library. This mode consists in passing source code to a compiler at runtime, so that the library can compile and launch a GPU kernel on-the-fly when the application needs it.

CUDA provides nvrtc [150], a device compiler (packaged as a library) specially designed for such runtime compilation. We are currently using the jitify [151] library from NVIDIA to preprocess, compile, link and load the kernels at runtime. Thanks to memory and file caches, this approach is effectively as efficient as the "offline modes" we have just described. In JIT mode, the library and the application are pure C++ projects and are compiled as any other C++ projects, *i.e.* CUDA is not involved in the build system anymore. Indeed, in this mode, CUDA is just a library, and as a library, it can be linked statically, or loaded dynamically (and lazily) if the application decides at runtime that the GPU is needed. Runtime compilation also creates new optimisation opportunities by baking in runtime constants and autotuning to compile time, and it allows to easily and specifically target the architecture of the GPU(s) on the system where the application is running. While JIT compilation is less invasive and can potentially generate better code than the offline mode we previously described, it requires the user-defined types and operators to be in separate headers that we can pass to nvrtc [150]. As shown in the code snippet 3.4, this can be done easily by simply adding the headers where these types are defined to the list of headers that the library sends to nvrtc. This can be done when setting up the project, directly from CMake [152]. The only real limitation we have found with this approach is that lambdas (which have an unspecified type) cannot be passed as operators.

```
// -- CMakeLists.txt --
project(my_application LANGUAGES C++) // CUDA is not required

// Fetch noa from github
include(FetchContent)
FetchContent_Declare(noa
        GIT_REPOSITORY ${noa_github_repository}
        GIT_TAG ${noa_github_tag})
FetchContent_MakeAvailable(noa)
```

```
10
11 // noa comes with its targets and one function called
12 // "noa_add_headers_to_cuda_jit", which takes a list of headers
13 // to preprocess and preinclude to the CUDA runtime compiler.
14 // Note that these headers are embedded inside the application,
15 // so they don't need to be available at runtime.
16 noa_add_headers_to_cuda_jit(my_application_header.hpp)
17
18 // Then continue as normal...
19 add_executable(my_application
20     my_application_header.hpp
21     my_application.cpp
22     )
23 target_link_library(my_application PRIVATE noa::noa)
24
25
26 // -- my_application_header.hpp --
27
28 // Include or define types and operators that should be
29 // supported by the CUDA backend. Because CUDA-C++ is very
30 // similar to C++, we can easily write code that can be
31 // compiled by any C++ compiler and also compiled by the
32 // CUDA runtime compiler, nvrtc. Note that the library
33 // calls nvrtc at compile time (in a step called preprocess)
34 // to parse this file, 1) to make sure the syntax is valid,
35 // 2) to fetch included headers and 3) to embed the
36 // header(s) in the application binary for better portability.
37
38 // All of noa's headers are fully compatible with nvrtc
39 #include <noa/Types.hpp>
```

```
40
41 // My special data-type.
42 struct my_type { double value; };
43
44 // My special operator. The operator needs to be copyable.
45 // Interestingly, STL functions like std::cos are supported,
46 // as long as there exists a corresponding implementation
47 // in the libcu++ standard library. Alternatively, noa
48 // comes with replacements for these, like noa::cos, which
49 // are compatible with GPU device code.
50 template<typename T>
51 struct my_special_operator {
52     T random{};
53     constexpr void operator()(T value, my_type& out) const {
54         out.value = value > 0 ? std::sqrt(value) : random;
55     }
56 };
57
58
59 // -- my_application.cpp --
60
61 #include <noa/Array.hpp>
62 #include <noa/Ewise.hpp>
63 #include "my_application_header.hpp"
64
65 noa::Array array = noa::random<double>(
66     noa::Uniform{-1., 1.}, 1024, {.device="cuda"});
67 noa::Array output = noa::like<my_type>(array);
68 double random = noa::random_value<double>(
69     noa::Uniform{0., 1.});
```

```
70
71 // If this is the first time the application is run on
72 // the system (no file cache) and if it is the first time
73 // this function is called in the program (no memory cache),
74 // nvrtc is called to compile this version of noa::ewise.
75 // Thanks to the CMake setup, it has at its deposition all
76 // of the library headers and pre-includes the user-provided
77 // header(s), in this case "my_application_header.hpp.
78 noa::ewise(array, output, my_special_operator{random});
79 // Note that a lambda can still be passed, but an exception
80 // will be thrown if the device is a GPU.
```

Snippet 3.4: User-defined type and operator to the CUDA backend using nvrtc.

The library aims to support both the offline approach, where all files are compiled as CUDA (`.cu`) files by nvc++, and the JIT approach using nvrtc. The offline approach is more invasive, as it forces the library and the application to be compiled by nvc++ in order to provide GPU support. The JIT approach, on the other hand, completely separates the CPU and GPU code generation. However, while it has its performance advantages, it forces user-defined operators to be in separate headers.

In the future, other GPU backends could easily be added. As we will see, the library is designed to be backend- and device-agnostic and most of its code resides in the core and library frontend. As a result, each backend only has to implement a few types and the core functions. AMD's ROCm-HIP [129] platform is likely to be the next GPU backend we add, due to its similarities with CUDA. SYCL [131] could also be explored, and in the long term future, C++26 execution [98] is likely to become the primary backend.

3.3 Multidimensional data

3.3.1 BDHW order

The library uses a fixed 4-dimensional (4d) model to manipulate multidimensional data. The dimension order is always following the Batch-Depth-Height-Width (BDHW) order. This model was first chosen for its simplicity and performance.

First, having a fixed number of dimensions allows us to encode the shape and strides in static arrays, instead of dynamic arrays saved on the heap. These arrays can be easily created and efficiently passed around, without involving dynamic allocations. The optimisations of having a fixed sized array are also welcomed.

Second, having to deal with a fixed number of dimensions was originally simpler to code on the library side, but is also simpler to reason about on the user side. Since the BDHW convention is used throughout the library, users know what to expect and have to follow a strict but simple convention. This is particularly relevant when comparing with NumPy-like arrays [69] and their dynamic number of dimensions, where a significant part of the code is just dedicated to rearranging dimensions [153].

Third, because the BDHW convention is enforced throughout the library, dimensions have a logical meaning. For instance, we can naturally differentiate between a stack of 2d images (`shape=[n,1,h,w]`) and a 3d volume (`shape=[1,d,h,w]`). Functions can have strict checks regarding the shape of the input/output arrays and can quickly detect logical incompatibilities. For instance, `noa::geometry::transform_2d` expects 2d arrays and will throw an error if 3d arrays are passed.

While we only allow nd-arrays up to 4 dimensions, we do support more complex data-types compared to NumPy (or PyTorch). For instance, to represent an array of `4x4` matrices in NumPy, one would need a floating-point array with 3 dimensions (e.g. `shape=[n,4,4]`) or maybe more to follow broadcasting rules. In our case, the `4x4` matrix is the data-type

(e.g. `noa::Mat44<f64>`) resulting in having to use one single dimension to store an array of matrices (e.g. `shape=[n,1,1,1]`). Interestingly, both approaches often result in the same memory layout.

As a result, and in our experience, 4 dimensions seem to be enough. We don't exclude in the future to revisit this model, for instance by adding the number of dimensions in the type of the array (`Array<f64,5>`), or by supporting nd-arrays with a dynamic number of dimensions.

3.3.2 Strides and memory layout

Each dimension of our nd-arrays is encoded by a stride, which is defined at runtime. If the array is contiguous, the innermost dimension, *i.e.* the dimension with the smallest stride, has a stride of 1 element. Strides of 0 denote "broadcasted dimensions", where a single element in memory is used to represent an entire dimension. Note that while the library technically supports negative strides, they are not tested yet and some things are likely to break.

A memory layout is said to follow the rightmost order when the innermost stride is at the right (the "last" dimension) and strides increase right to left. Similarly to NumPy, this is what we default to. In the BDHW convention, the rightmost order is synonymous to row-major and a rightmost contiguous array is said to be C-contiguous (as opposed to F-contiguous). For instance, a C-contiguous 3d array of shape `[2,3,4,5]` has its strides equal to `[60,20,5,1]`.

The library tries to make as few assumptions on the memory layout as possible. As a result, to ensure good performance, it tries (whenever possible) to find the fastest way to iterate through arrays by looking at the shape and strides of the inputs and/or outputs. However, doing so is not always possible. For instance, `noa::iwise`, a function that abstracts a nd-loop, does not take arrays as inputs, thus cannot reorder dimensions. In these cases,

the rightmost order is always assumed. Moreover, when input and/or output arrays have different layouts, it is not always easy or possible to find the overall best order, so we do recommend to keep with the default and use rightmost arrays.

3.3.3 Array and View

As introduced in chapter 2, the multidimensional array is one of the fundamental types for manipulating multidimensional dense data. This library provides a owning 4d-array called `Array` and a non-owning 4d-array called `View`.

The data-type of the array is defined at compile time. Having the data-type exposed in the type of the arrays, e.g. `Array<T>`, gives more information to the type system to catch misuses and helps to enforce certain things at compile time. For instance, some (member) functions are only available for certain data-types, like `noa::signal::bandpass(...)` which only accepts complex arrays.

Data-type reinterpretations are of course allowed, as long as the C++ strict type-aliasing rule [154] is not broken. An array of any type can be reinterpreted to an array of bytes, thereby allowing to manipulate "type-erased" arrays. Reinterpreting between the complex data-type and the underlying floating-point type is also allowed. Reinterpretation is done via the `Array<T>::as<U>()` member function.

`Array` allocates and owns its data, and uses `std::shared_ptr` [155] to reference-count resources. As a result, the data-type of `Array` cannot be const-qualified. On the other hand, `View` simply keeps track of a raw-pointer (which can point to const data) and assumes the resource will stay valid throughout its lifetime. `View` is a lighter version of `Array`: it is 80 bytes in size, and satisfies the "trivial class" requirement [156], making it cheap to construct and move around. `Array` and `View` can be used interchangeably in the library API, but in practice, we recommend to use `Array` to allocate and manage memory and use `View` to manipulate and pass data around.

```
template<typename T>
class Array {
    Shape<i64, 4> shape; // a static array of four i64 values
    Stride<i64, 4> strides; // same as above
    std::shared_ptr<T[]> buffer; // the 1d dynamic array
    ArrayOption option; // contains Device and Allocator
};

template<typename T>
class View {
    Shape<i64, 4> shape; // same as Array
    Stride<i64, 4> strides; // same as Array
    T* pointer; // points to an existing array
    ArrayOption option; // same as Array
};
```

Snippet 3.5: Layout of `Array` and `View`

3.3.4 Accessor and (md)span

`Array` and `View` provide an `operator()` taking 4d-indices, which can be used to access the underlying values of the nd-arrays. However, this is not the preferred way to access data from these arrays. First, this may not be the most efficient way to access the data, for instance, 1d contiguous arrays can be accessed using a single index and uses a fixed stride of 1 (known at compile time). Second, depending on the device and memory resource, direct access may not be even allowed. In fact, `Array` and `View` are defined in the unified interface of the library (the frontend) and don't belong to the core or the backends. They are simply here to offer a cleaner and simpler way to manipulate multidimensional data.

`Accessor` is a core-type used to abstract multidimensional element-accesses into a common interface. This type is intended to be used by the backends, and as such, it prioritises performance. An accessor contains the strict necessity: the raw-pointer of the memory region and the strides.

```
template<typename T, size_t N, typename I,
         PointerTraits PointerTrait = PointerTraits::DEFAULT,
         StridesTraits StridesTrait = StridesTraits::STRIDED>
class Accessor {
    T* pointer;
    Stride<I, N> strides;
    // StridesTraits::CONTIGUOUS stores N-1 elements
};
```

Snippet 3.6: Layout of `Accessor`

As opposed to `Array` and `View`, the number of dimensions (`N`) is in the type definition, as well as the integer type of the strides (`I`). Two additional template parameters are also available: `PointerTraits` and `StridesTraits`. By default, the pointer is not marked with any attributes, but the "restrict" traits can be added with `PointerTraits::RESTRICT`. This is useful to indicate that pointers don't alias, which may help to generate better code. By default, strides are fully dynamic (one dynamic stride per dimension). However, the rightmost dimension can be marked contiguous using `StridesTraits::CONTIGUOUS`. `Accessors` (and the library internals) use the rightmost convention, so that the innermost dimension is the rightmost dimension. As such, contiguity here implies C-contiguous. F-contiguous layouts are not supported by the accessors, as these layouts are expected to be already reordered to C-contiguous before creating the contiguous accessor. With `StridesTraits::CONTIGUOUS`, the innermost/rightmost stride is fixed to 1 and is not stored, resulting in the strides being truncated by one dimension (`Strides<I,N-1>`). In

case of a 1d contiguous accessor, this means that the strides are empty (`Strides<I,0>`) and the indexing is equivalent to pointer/array indexing.

```
// Import i64, f64, Shape and Accessor
using namespace ::noa::types;
auto shape = Shape4<i64>{2, 3, 4, 5};
std::unique_ptr buffer = std::make_unique<f64[]>(
    shape.elements());
// Note that Shape4<i64> is a simple alias to Shape<i64, 4>

// Create a 4d accessor. Shape::strides() computes the rightmost
// strides of the shape. The C-contiguous order is the default,
// shape.strides() is equivalent to shape.strides<'C'>(), but
// F-contiguous order could also be created, using
// shape.strides<'F'>().
auto accesor_4d = Accessor<f64, 4, i64>(
    buffer.get(), shape.strides());

for (i64 i{0}; i < shape[0]; ++i) {
    // Reduce dimension using multidimensional C-style indexing.
    // Note that accessor_3d is an AccessorReference, which
    // simply points to an existing Accessor.
    auto accessor_3d = accessor_4d[i];
    for (i64 j{0}; j < shape[1]; ++j)
        for (i64 k{0}; k < shape[2]; ++k)
            for (i64 l{0}; l < shape[3]; ++l)
                accessor_3d(j, k, l) =
                    static_cast<f64>(j + k + l);
}
```

```
27
28 // Various type definitions are provided to simplify code.
29 // AccessorContiguousI64 defaults to I=i64 and StridesTrait=
30 // StridesTraits::CONTIGUOUS. Note here that the stride is
31 // not needed in the constructor, because the accessor is
32 // 1d and strides are contiguous, i.e. the strides are the
33 // Strides<I,0> empty type.
34 auto accessor_1d_contiguous =
35     AccessorContiguousI64<const f64, 1>(buffer.get());
36
37 f64 sum{0};
38 for (i64 i{0}; i < shape.elements(); ++i) {
39     sum += accessor_1d[i];
40     // or equivalently accessor_1d(i) or buffer[i];
41 }
```

Snippet 3.7: Examples and usage of `Accessor`

While accessors are a good abstraction for backends, they are not the safest, nor they are the most practical. This mostly results from the fact that the size of the dimensions are not stored, so accessors cannot bound-check the indexes against their dimension size. Sizes are indeed not stored because they are not necessary to compute the memory offsets, and in a lot of cases, the input and output arrays have the same shape, which would lead to storing useless or redundant data in the operators.

In user space, (md)spans should be used instead. These are similar to `Accessor`, but keep track of the size of each dimension. The library comes with its own `Span`, which is equivalent to C++20 `std::span` [157] (note that we do plan to use `std::span` instead of `noa::Span` as part of our C++20 upgrade). We plan to add multidimensional spans in the future, which would be equivalent to C++23 `std::mdspan` [86].

```
using namespace ::noa::types; // f64, i64, i32, View, Array
using namespace ::noa::indexing; // Ellipsis and Slice

// This is similar to noa::filled<f64>({4, 1, 64, 64}, 3.);
const Array array = noa::empty<f64>({4, 1, 64, 64});
for (auto& e: array.span())
    e = 3.;

// Extract the 32x32 centre.
// This is be equivalent to array[..., 16:48, 16:48] in NumPy.
Array centre = array.subregion(
    Ellipsis{}, Slice{16, 48}, Slice{16, 48});

// Array::span() throws an error here, because "centre"
// cannot be reshaped to a 1d array anymore.
// for (auto& e: centre.span())
//     e = 2.;

// With mdspan (which we don't support, yet) we could
// use nested loops. At the time of writing, we could
// instead use an Accessor.
// Array::accessor_and_shape() stacks the outer-dimensions
// together, so shape_3d is {4, 32, 32}
const auto [accessor_3d, shape_3d] =
    centre.accessor_and_shape<f64, 3>();
for (i64 batch{0}; i < shape_3d[0]; ++batch)
    for (i64 y{0}; y < shape_3d[1]; ++y)
        for (i64 x{0}; x < shape_3d[2]; ++x)
            accessor_3d(batch, y, x) = 2.;
```

```
30
31 // We could also use the index-wise function (see the
32 // next section for more details). This should generate
33 // the same code as the above example.
34 noa::iwise(
35     centre.shape().filter(0, 2, 3), array.device(),
36     [accessor_3d = centre.accessor<f32, 3>()]
37     (i64 batch, i64 y, i64 x) {
38         accessor_3d(batch, y, x) = 2.;
39     });
40
41 // But since this example doesn't require indices,
42 // we could also use the element-wise function
43 // (see the next section for more details).
44 // Something like:
45 noa::ewise(centre, {}, [](f64& e) { e = 2.; });
46 // Which is equivalent to:
47 noa::fill(centre, 2.);
```

Snippet 3.8: Different ways to access the elements of an `Array`

3.4 Core functions

The library heavily relies on pointwise operations and provides core functions for generic algorithms, including the `noa::iwise` function, the `noa::ewise` function, the `noa::reduce_ewise` and `noa::reduce_iwise` functions.

3.4.1 Index-wise

The index-wise core function, `noa::iwise`, provides a unified interface to nd-dimensional for-loops, where n is between 1 and 4. This function takes a shape, a device and an index-wise operator. The operator will be called once for each nd-indices in the nd-shape, in no particular order. As one of the core function, `noa::iwise` actually interacts with the backends and dispatches work to the current-stream. Most cryoEM-specific functions in the library rely on `noa::iwise` and never have to directly interact with the backends. For instance, `noa::signal::bandpass(...)` is creating the `noa::signal::Bandpass<...>` pointwise operator and passes it to the core functions.

```
// The template-function declaration looks like:
// template<typename T, size_t N, typename F>
// void noa::iwise(
//     const noa::Shape<T, N>&,
//     const noa::Device&,
//     F&&
// );

// When passing the nd-indices, the backends have to follow the
// following logic:
// - First, try to pass the nd-indices as a packed
//   noa::Vec<T, N>. See (1).
// - If the operator doesn't support this, try to
//   pass the indices one by one. See (2).
noa::iwise(
    centre.shape(), array.device(),
    [accessor = center.accessor()](Vec4<i64> indices) { // (1)
        accessor(indices) = 2.;
```

```
19      });
20
21  // Here's a more complex example:
22  // Shape::filter(0, 2, 3) returns a new shape, such as
23  // Shape<i64, 3>{shape[0], shape[2], shape[3]}.
24  // Shape::as<i32>() is here to emphasise that the
25  // indices have the integer type of the input shape.
26  noa::iwise(
27      centre.shape().filter(0, 2, 3).as<i32>(), array.device(),
28      [accessor_3d = center.accessor<f64, 3, i32>()]
29      (i32 batch, i32 k, i32 l) { // (2)
30          accessor_3d(batch, k, l) = 2.;
31      });
```

Snippet 3.9: `noa::iwise`

3.4.2 Element-wise

The element-wise core function, `noa::ewise`, can be seen as a wrapper over the index-wise core function. This function is useful when the position of the element in the array is not required, e.g. copy, fill, for arithmetics or other maths functions like `sin()`. These take the input and output arrays, and an element-wise operator. One advantage of this function over the `noa::iwise` function is that it can analyse the input and output array(s) to deduce the most efficient way to traverse them, by reordering the dimensions, collapsing contiguous dimensions together (up to 1d), and can trigger vectorisations by checking for data contiguity and aliasing. For instance, `noa::ewise(input, output, noa::Copy{})` can be optimised up to a `memcpy` CPU instruction, and similarly, `noa::ewise(input, output, noa::Fill{0})` can be optimised up to a `memset` CPU instruction. On the GPU, vectorised load/store instructions will be used. Originally, the API supported a fixed

number of inputs (up to 3) and a single output. More recently, significant efforts were put into making this API more flexible and it can now take a variable number of inputs and outputs.

```
// The template-function declaration looks like:
// template<typename Inputs, typename Outputs, typename F>
// void noa::ewise(const Inputs&, const Outputs&, F&&);

using namespace ::noa::types; // to imports i32, f32 and Tuple

const auto buffer = noa::arange<f32>(300).reshape({3,1,10,10});
const auto lhs = buffer.view().subregion(0); // shape=(1,1,10,10)
const auto mhs = buffer.view().subregion(1); // same shape
const auto rhs = buffer.view().subregion(2); // same shape

const auto output_0 = noa::empty<f32>({1, 1, 10, 10});
const auto output_1 = noa::like(output_1);

// All inputs and outputs should be on the same device and have
// compatible shapes. wrap(...) is a template function similar
// to std::forward_as_tuple but does instead return a special
// tuple-like type that will get unpacked in the operator.
// Also note that Array and View can be used interchangeably,
// as always.
noa::ewise(
    noa::wrap(lhs, mhs, rhs), noa::wrap(output_0, output_1),
    [](i32 l, i32 m, i32 r, f32& o0, f32& o1) {
        o0 = l + m * 2;
        o1 = std::cos(m / l + r);
```

```
    });

// The (sort of) equivalent in NumPy would be:
// output_0 = lhs + mhs * 2
// output_1 = np.cos(mhs / lhs + rhs)
```

Snippet 3.10: `noa::ewise`

3.4.3 Reductions

The pointwise-reduction core functions are the final core functions that backends need to implement. These aim to provide whole-array reductions or per-axis reductions. Each reduction provides an element-wise or index-wise interface, with the same advantages and disadvantages as the `noa::ewise` and `noa::iwise` functions. For brevity, the example below presents the whole-array reduction API, but note that the per-axis reductions have a similar API.

```
// The template-function declarations look like:
// template<typename Inputs, typename Reduced,
//          typename Outputs, typename F>
// void noa::reduce_ewise(
//     const Inputs&, const Reduced&,
//     const Outputs&, F&&);
//
// template<typename T, size_t N, typename Reduced,
//          typename Outputs, typename F>
// void noa::reduce_iwise(
//     const noa::Shape<T, N>&, const Device&,
//     const Reduced&, const Outputs&, F&&);
```

```
13
14 // Reduction operators should define (at most)
15 // 3 member functions:
16 // .init(inputs..., reduced...)
17 //  -> Main reduction step, to convert the input types to the
18 //     reduced types. The iwise reduction has init(Vec<T, N>,
19 //     reduced...) or init(indices..., reduced...). If not
20 //     defined, this defaults to:
21 //     (*this)(inputs..., reduced...);
22 // .join(reduced..., reduced...)
23 //  -> To join the reduced values together. This is necessary
24 //     for multithreaded/GPU reductions. If not defined, this
25 //     defaults to (*this)(reduced..., reduced...);
26 // .final(reduced..., outputs...)
27 //  -> Called as a postprocessing step to convert the
28 //     reduced types to the output types. If not defined,
29 //     this defaults to:
30 //     ((outputs = static_cast<Outputs>(reduced)), ...);
31
32 // These defaults greatly simplify the operators.
33 // For instance, computing the sum of an array can be done
34 // such as:
35 Array<f64> src;
36 f64 sum{0};
37 noa::reduce_ewise(
38     src, f64{}, sum,
39     [](auto value, auto& sum) { sum += value; });
40
41 // Here's a more complex example.
42 const Array<f32> array; // an array we want to reduce
```

```
const Array<i32> mask; // a binary mask
f64 sum_inside_mask; // the wanted the sum inside the mask...
f32 max_inside_mask; // and the maximum value inside the mask

// Using the element-wise approach:
struct ReduceMaskEwise {
    void init(f32 value, i32 mask, f64& sum, f32& max) {
        if (mask > 0) {
            sum += static_cast<f64>(value);
            max = std::max(max, value);
        }
    }
    void join(f64 isum, f32 imax, f64& osum, f32& omax) {
        osum += isum;
        omax = std::max(omax, imax);
    }
    // default .final()
};
noa::reduce_ewise(
    noa::wrap(array, mask),
    noa::wrap(f64{}, std::numeric_limits<f32>::lowest()),
    noa::wrap(sum_inside_mask, max_inside_mask),
    ReduceMaskEwise{});

// Or using the index-wise API:
struct ReduceMaskIwise {
    Accessor<const f32, 4> a_array;
    Accessor<const i32, 4> a_mask;

    void init(const Vec4<i32>& indices, f64& sum, f32& max) {
```

```
73          if (a_mask(indices) > 0) {
74              const auto& value = a_array(indices);
75              sum += static_cast<f64>(value);
76              max = std::max(max, value);
77          }
78      }
79      // Same as ReduceMaskEwise.
80      void join(f64 isum, f32 imax, f64& osum, f32& omax) {
81          osum += isum;
82          omax = std::max(omax, imax);
83      }
84      // default .final()
85  };
86  noa::reduce_iwise(
87      array.shape(), array.device(),
88      noa::wrap(f64{}, std::numeric_limits<f32>::lowest()),
89      noa::wrap(sum_inside_mask, max_inside_mask),
90      ReduceMaskIwise{array.accessor(), mask.accessor()});
91
92  // Note that the library provides more specialised
93  // reduce functions. For instance, the above example
94  // could be simplified (maybe at the cost of slightly
95  // worse performance) to:
96  sum_inside_mask = noa::sum(array, mask);
97  max_inside_mask = noa::max(array, mask);
```

Snippet 3.11: noa::reduce_ewise and noa::reduce_iwise

3.5 cryoEM specificities

On top of the generic nd-arrays utilities and core functions, the library comes with a set of operators and functions (see fig. 3.1). This library is still experimental, so these are subject to change and updates adding new operators and functions are frequently pushed.

3.5.1 Geometric transformations

As introduced in previous chapters, geometric transformations are central to cryoEM data processing, so evidently these transformations need to be perfectly integrated into cryoEM core libraries.

We have previously discussed in section 3.3. that the rightmost BDHW dimension order is used throughout the library. That includes multidimensional arrays, shape and strides, vectors (e.g. translation vectors), but it also includes rotation and affine matrices, thus offering a consistent ecosystem for geometric transformations. Leftmost matrices are admittedly more frequent in the literature, but note that the library provides tools to construct matrices, so users rarely have to construct matrices themselves, and in the worst case scenario, leftmost matrices can easily be converted to rightmost matrices as shown in the following code snippet.

```
// Import f64, f32, i64, Vec, Mat33, Shape and Array.
using namespace ::noa::types;
namespace ng = ::noa::geometry;
namespace ni = ::noa::indexing;

// Example of a 4x4 affine matrix, in the leftmost
// and rightmost order:
// leftmost = [[Xx, Xy, Xz, Tx],
//             [Yx, Yy, Yz, Ty],
```

```
//              [Zx, Zy, Zz, Tz],
//              [ 0,  0,  0,  1]]
//
// rightmost = [[Zz, Zy, Zx, Tz],
//              [Yz, Yy, Yx, Ty],
//              [Xz, Xy, Xx, Tx],
//              [ 0,  0,  0,  1]]

// The library is consistent with its dimension order, and since
// matrices are no exceptions, we use the rightmost dimension
// for matrices too. If the user comes with their own leftmost
// matrices, the library provides utilities to convert them to
// rightmost matrices:
Mat44 rightmost = ng::translate(Vec3{0., 0., 1.}); // zyx1
Mat44 leftmost = ni::reorder(
    rightmost_matrix, Vec4{2, 1, 0, 3});

// Transform a vector.
const auto vector = Vec3<f64>{0, 0, 1}; // unit vector x
const auto rotation_z = ng::rotate_z(noa::deg2rad(45.));
const auto rotated_vector = rotation_z * vector;

// Here's a more complex example: rotating a 3d volume
// around z by 45 degrees.
i64 z = 64, y = 64, x = 64;
auto shape = Shape4<i64>{1, z, y, x};
auto rotation_center = (shape.pop_front().vec / 2).as<f64>{};
// ... which is equivalent to Vec3<f64>{z/2, y/2, x/2}
auto rotation_z = ng::rotate_z(noa::deg2rad(45.)); // Mat33<f64>
auto inverse_transform = ( // Mat44<f32>
```

```
    ng::translate(rotation_center) *
    ng::linear2affine(rotation_z) *
    ng::translate(-rotation_center)
).inverse().as<f32>();

Array<f32> input, output; // ...initialise the arrays...

// Compute the affine transformation.
ng::transform_3d(input, output, inverse_transform);

// Internally, the index-wise operator looks something like:
// operator()(i64 batch, i64 z, i64 y, i64 x) {
//     auto coordinates = Vec3<f32>{z, y, x};
//     coordinate = inverse_transform * coordinates;
//     auto interpolated_value = interp_input_at(coordinate);
//     output_accessor(batch, z, y, x) = interpolated_value;
// }
```

Snippet 3.12: Matrices and simple geometric transformations

Another example of the deep integration of geometric transformations in the library is the `Texture` type. `Texture` is provided by the library frontend and aims to allow device-agnostic code while still using GPU textures. Here's an example of the usage of textures.

```
// Import f32, Mat33, Interp, Array and Texture.
using namespace ::noa::types;
Array<f32> input, output; // to initialise...
Array<Mat33<f32>> inverse_transforms; // to initialise...

// If output is on the GPU, a GPU texture is created from input.
```

```
// If output is on the CPU, Texture is a simple wrapper of the
// existing array.
auto input_textures = Texture<f32>(
    inputs, outputs.device(), Interp::CUBIC_BSPLINE_FAST);

// Make sure the matrices are on the compute-device.
if (inverse_transforms.device() != outputs.device())
    inverse_transforms = inverse_transforms.to(output.device());

// Compute the affine transformation. On the GPU, this uses
// hardware interpolation.
noa::geometry::transform_2d(
    input_textures, outputs, inverse_transforms);
```

Snippet 3.13: `Texture` in device-agnostic code

3.5.2 FFT layouts

As it is often done in signal and image processing, the library computes Discrete Fourier Transforms (DFTs) using a family of algorithms called Fast Fourier Transforms (FFTs) algorithms [158]. There are two important aspects we would like to emphasise in this section.

First, the DFT of real-valued inputs are Hermitian, *i.e.* the component at frequency x is the complex conjugate of the component at frequency $-x$, which means that for real-valued inputs there is no information in the negative frequency components that is not already available from the positive frequency components. FFT libraries often have special implementations designed to operate on real inputs, and exploits this symmetry by computing only the positive frequency components, up to and including the Nyquist frequency. The resulting "half-transforms" are often called Real Fast Fourier Transforms (rFFTs) or non-redundant

transforms (as opposed to the full-transforms which store both x and $-x$). Furthermore, because complex elements are twice as large as the real-valued ones, the half-transforms are still large enough to store the real-valued inputs. FFT libraries have used this property to develop in-place rFFTs.

Second, FFT algorithms ingest and output spectrums with frequencies organised in a certain order. Given the 1d spectrum `A` of size `n` and using NumPy's array notation, `A[0]` contains the zero-frequency term, then `A[1:n/2]` contains the positive-frequency terms, and `A[n/2+1:]` contains the negative- frequency terms, in order of decreasingly negative frequency. For instance, if `n`=6, the layout is `[0,1,2,-3,-2,-1]`, and if `n`=7, the layout is `[0,1,2,3,-3,-2,-1]`. For half-transforms, if `n`=6 or `n`=7, the layout is `[0,1,2,3]`. To facilitate certain operations or to visualise the transform, a circular-shift can be applied to shift the zero-frequency to the centre of the spectrum. We refer to this operation as "centering", and libraries often provide the `fftshift`/`ifftshift` functions to perform these centerings.

We provide `noa::fft::remap`, a function used to remap FFTs to half/full and centred/non-centred layouts. It supports all of the possible combinations of layouts and can operate in-place for some layouts and shapes. Importantly, functions operating on FFTs always take a remap operation to keep track of what the input layout currently is and what the output layout should be. Note that in this case only a subset of layouts are usually supported. This remap operation is specified as the first template parameter of these functions, so everything is encoded at compile time. Note that since these remappings are merged into the other pointwise operators, they usually have a zero cost. The library denotes half-transforms with a `H`, full-transforms with a `F` and the centred layout with a `C`. The layout of the input and outputs, *i.e.* the remap operation, is encoded such as `H|F(C)2H|F(C)`, where the left side (relative to the `2`) refers to the input layout and the right side refers to the output layout. For instance, `H2HC` refers to a remap going from a non-redundant non-centred transform to a non-redundant centred transform (this is sometimes called a `rfftshift`).

```
using namespace ::noa::types; // import f32
namespace nf = ::noa::fft;
namespace ns = ::noa::signal;

// This is a special factory function to generate arrays for
// in-place rffts. "images" is real, "images_rfft" is complex,
// both point to the same memory.
auto [images, images_rfft] = nf::empty<f32>({1, 1, 64, 64});

 // ...initialise images...

// In-place bandpass filtering.
// - The Fourier transform functions always take the non-
//   centred transforms.
// - Other functions operating in Fourier space need the
//   Remap argument, which can be entered as a string or
//   using the noa::fft:Remap enum.
// - Note that bandpass, as many other functions capable
//   of operating on rffts, needs the logical-shape of
//   the transform, i.e. the shape of the real array.
nf::r2c(images, images_rfft);
ns::bandpass<"h2h">( // h2h -> no remap
    images_rfft, images_rfft, images.shape(),
    {.highpass_cut=0.05, .highpass_width=0.04,
     .lowpass_cut=0.4, .lowpass_width=0.1}
);
nf::c2r(images_rfft, images);

// Another example, with out-of-place filtering
```

```
// and FFT remapping:

// In-place rfft, then in-place centering, i.e. fftshift.
nf::r2c(images, images_rfft);
nf::remap("h2hc", images_rfft, images_rfft, images.shape());
// ...do something that needs the centered images_rfft...

// Then do the out-of-place filtering.
// Notice the on-the-fly remapping hc2h, aka irfftshift,
// anticipating the c2r transform.
const auto filtered_rfft = noa::like(images_rfft);
ns::bandpass<"hc2h">(
    images_rfft, filtered_rfft, images.shape(),
    {.lowpass_cut=0.5});
nf::c2r(filtered_rfft, images);
```

Snippet 3.14: Examples using rFFTs

Generic and core functions (in noa)				
dot matmul lstsq randomize rmsd normalize normalize_per_batch sort	split_complex fuse_complex real imag ewise iwise reduce_ewise reduce_iwise	min, max min_max median sum, mean var, std mean_var mean_std l2_norm	cast copy copy_batches fill zeros ones arange linspace iota	extract insert_sequence permute permute_copy resize insert_subregions extract_subregions argmin argmax
Factory functions (in noa)				
random	fill ones	zeros empty	arange linspace	iota like
Signal (in noa::signal)				
convolve, convolve_separable median_filter cross_correlate find_peak cross_correlation_score		bandpass lowpass highpass phase_shift_2d phase_shift_3d	ctf_anisotropic ctf_isotropic fsc_anisotropic fsc_isotropic	standardize_ifft window_blackman window_gaussian window_sinc
Geometry (in noa::geometry)				
fourier_insert_rastesize_3d fourier_insert_interpolate_3d fourier_extract_3d fourier_insert_interpolate_and_extract_3d gridding_correction rotational_average rotational_average_anisotropic cartesian2polar polar2cartesian			euler2matrix, matrix2euler ellipse, sphere, rectangle, cylinder transform_2d transform_3d symmetrize_2d symmetrize_3d transform_and_symmetrize_3d transform_and_symmetrize_2d cubic_bspline_prefilter	
FFT (in noa::fft)				
next_fast_shape alias_to_real	zeros ones	empty	remap resize	r2c, c2r c2c

Figure 3.1: Current functions available in the library frontend. As this is intended to be a brief overview of the user interface, this only includes the frontend, and ignores most utility functions, as well as the core and the backends

Chapter 4

Projections in Fourier space

4.1 Backward projection 89
4.1.1 Using the data-driven approach 89
4.1.2 Using the grid-driven approach 92
4.1.3 Correcting for the apodisation of the reconstruction 99
4.1.4 Normalising the multiplicity 100
4.2 Forward-projection 102
4.2.1 Fourier extraction 102
4.2.2 On-the-fly windowed sinc convolution 103
4.3 Fused backward-forward projection 107

As introduced in section 1.4, backward and forward projections are two critical steps in cryoEM. In this chapter, we will look at how these operations are usually implemented, and the modifications and improvements we have made when adding these transformations to our cryoEM core library.

4.1 Backward projection

Reconstructing a 3d volume from 2d projections can be accurately and efficiently implemented in Fourier space by inserting the Fourier transform of these projections, aka central slices, into the Fourier transform of the volume [31, 56, 159]. This operation requires resampling the central slices into a 3d cartesian grid. To do so, virtually all major current cryoEM software packages rely on very fast direct Fourier inversion algorithms [15, 66, 160, 161, 162], which are often implemented using one of the two different approaches.

4.1.1 Using the data-driven approach

Data-driven interpolation [163], aka gridding, spreads each data-point out over neighbouring grid-points, in what is essentially a convolution of the data-points with a kernel of a certain size and shape [164, 165, 166, 167]. Note that this convolution is implemented from the point of view of the input, which is the inverse operation of what is usually done for convolution. Indeed, convolutions are usually implemented by looping through the output, and each output value is the result of the weighted sum between the weights of the convolution kernel and the corresponding input values. In this case, however, implementations loop through the input, and the convolution kernel is applied to a single input data-point at a time and computes how much of this data-point will go into each neighbouring voxel. This operation is sometimes referred to as rasterisation. The code snippet 4.1 shows a simplified version of a data-driven approach, as usually implemented in cryoEM.

```
1 // Called for each pixel of every slice to insert:
2 void operator()(i64 slice_index, i64 u, i64 v) {
3     // 1. Get the fftfreq at the (u, v) coordinate.
4     auto fftfreq_2d = index_to_fftfreq(u, v);
5
6     // 2. Rotate the central slice to place it in the grid
```

```
    // reference-frame. This effectively does:
    // rotation_matrix * {0, fftfreq_2d[0], fftfreq_2d[1]}
    fftfreq_3d = transform_slice_to_grid(
        fftfreq_2d, slice_forward_rotation[slice_index]);

    // 3. Compute the convolution/rasterisation weights.
    // The kernel needs to be positioned exactly where the
    // input value is (at fftfreq_3d). Each neighbouring voxel
    // will be associated with a weight, the weight says how
    // much of the input value will go into the voxel. Most
    // cryoEM implementations use a trilinear pulse as the
    // kernel shape.
    auto weights = rasterisation_weights(fftfreq_3d); // (n,n,n)

    // 4. Convolution between the rasterisation kernel
    // and the input value.
    auto input_value = read_slice_at(slice_index, u, v); // (1,)
    auto rasterised_values = weights * input_value; // (n,n,n)

    // Note that step 3 and 4 effectively constitute a (forward)
    // 3d interpolation of the input data onto the grid.

    // 5. Add the contribution of this input pixel to the
    // 3d grid. This function will add the rasterised values
    // to the corresponding neighbouring voxels. Most cryoEM
    // implementations use a 2x2x2 kernel, so each input
    // value affects about 8 voxels. Multithreaded
    // implementations need this step to be thread-safe
    // because different input values contribute to the same
    // output voxels; this is usually done using an atomic add.
```

```
37      add_to_grid(fftfreq_3d, rasterised_values);
38  }
```

Snippet 4.1: Gridding operator

While this method can in theory be as accurate as the grid-driven approach (which is presented in the next section), there are some practical limitations that may limit its usefulness. First, and as we have said, while this approach requires to only loop through every Fourier component of the 2d central slices, as opposed to every voxel of the 3d grid, each input data-point needs to update n^3 voxels, where n is the size of the 3d convolution kernel. As such, the data-driven approach is impractical and inefficient for cases with large kernels ($n > 4$). Moreover, multithreaded implementations (including GPU implementations) that process the input data-points in parallel need to ensure that no data races occur when writing to the grid. In practice, this is done using an atomic-add, which further increases the cost of the rasterisation. Second, and as introduced in chapter 1, the multiplicity of the grid, *i.e.* how much data was inserted into each voxel, often needs to be tracked and normalised (see section 4.1.4). For small datasets that can be backward projected in a single operator call, e.g. a tilt series, grid-driven methods can easily normalise the multiplicity on-the-fly, while inserting the data in the cartesian grid. Unfortunately, this is not possible for data-driven approaches, which requires the insertion weights to be saved to another grid for post-processing. Third, while technically a 3d interpolation, instead of querying an interpolation of the input data-points at a given voxel coordinate, the rasterisation applies the interpolation weights at the given grid-points to the input data-point. This is a rather unique way to interpolate data, which makes existing implementations and methods (e.g. GPU texture interpolation) unfitted for the data-driven interpolation, thereby increasing the complexity of this approach.

Despite these limitations, data-driven interpolation is the most popular approach for SPA, mostly due to its good performance when using small convolution kernels. Indeed, SPA

often inserts many thousands of central slices into the same Fourier volume. It is thus safe to assume that interpolation and wrap-around errors (see next section) resulting from these small kernels are likely to be entirely averaged out. In cryoET, things are not so simple, and larger kernels can significantly improve the quality of reconstructions.

4.1.2 Using the grid-driven approach

The grid-driven approach tries to find how much each input central slice contributes to each voxel in the 3d Fourier volume [163]. In practice, this means that the entire grid needs to be traversed to insert a single or multiple central slices. While this may initially result in more computations, we will show that this approach may still be the preferred method for reconstructions that are made of only tens or a few hundreds of central slices. The code snippet 4.2 shows a simplified version of a grid-driven approach, as implemented in our library.

```
// Called for each voxel of the grid.
void operator()(i64 w, i64 v, i64 u) {
    // 1. Get the fftfreq of the voxel at coordinate (w, v, u).
    auto fftfreq_3d = index_to_fftfreq(w, v, u, size);

    c64 value{0};
    f64 multiplicity{0};

    // For every slice to insert:
    for (i64 i{0}: irange(n_slices)) {
        // 2. Compute the inverse transformation, going from
        // the 3d grid to the slice. This is effectively
        // projecting the voxel along the normal of the
        // central slice. "slice_rotation" is an accessor of
```

```
        // 3x3 rotation-matrices or quaternions.
        auto [fftfreq_z, fftfreq_2d] = transform_grid_to_slice(
            fftfreq_3d, slice_inverse_rotation[i]);

        // Only continue if the voxel is affected by this slice.
        if (std::abs(fftfreq_z) <= fftfreq_blackman) {
            // 3. Compute the windowed sinc.
            // This only depends on "fftfreq_z".
            f64 window = compute_windowed_sinc(
                fftfreq_z); // between 0 and 1

            // 4. Compute the value.
            // This only depends on "fftfreq_2d". This is a
            // normal 2d interpolation and can be any 2d
            // interpolation method.
            c64 i_value = interpolate_2d(i, fftfreq_2d);

            // 5. Apply the window to the value.
            // This effectively convolves the perfectly thin
            // central slice with the window function.
            i_value *= window;

            // 5. Add the contribution of the ith slice to
            // the current voxel. In this example, we assume
            // that all of the slices are inserted at once,
            // so we can record the multiplicity on-the-fly.
            value += i_value;
            multiplicity += window;
        }
    }
```

```
45
46      // 5. Correct for the multiplicity.
47      // In this example, we assume all of the slices were
48      // inserted, so "multiplicity" is the total sampling
49      // weight for the current voxel. If it is greater than
50      // one, it means multiple input slices contributed to
51      // this voxel, so we can take the weighted-mean of these
52      // contributions, effectively computing a weighted
53      // backward projection.
54      value /= std::max(1., multiplicity);
55
56      // We have collected the contribution of every slice for
57      // that voxel, we can now save it. Note that we also support
58      // adding to the volume, for cases where not all central
59      // slices are inserted at once.
60      write_at(w, u, v) = value;
61  }
```

Snippet 4.2: Grid-driven insertion operator

In the contiguous world, central slices are perfectly thin, *i.e.* there's no information along the normal of the central slices. In the discrete world, inserting an (even almost) perfectly thin central slice into the Fourier volume is actually not desirable because the projected rays would wrap-around at the edges of the reconstruction due to the periodic nature of the DFTs (see fig. 4.1). One of the goals when inserting central slices into the 3d grid is to somehow limit, or completely remove, these wrap-around effects. To do so, the perfectly thin central slices can be convolved with a window function, thereby multiplying projected rays with the inverse Fourier transform of that window function, which effectively results in applying a real-space mask onto the reconstruction.

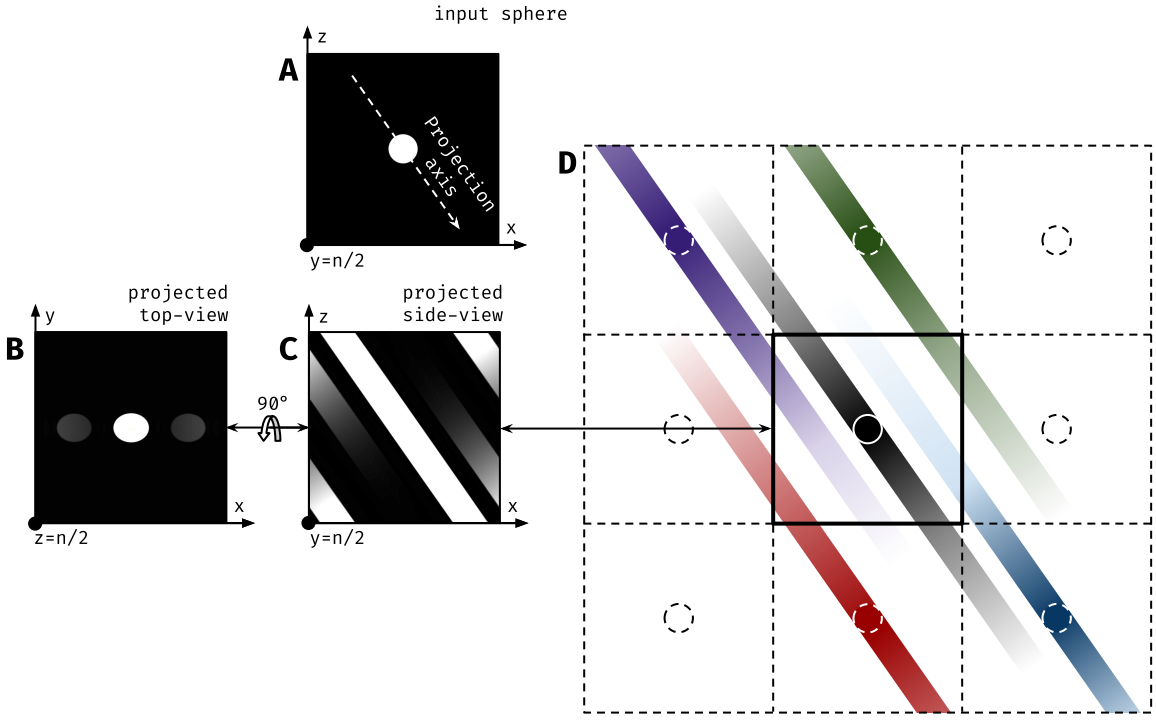


Figure 4.1: Inserting a (nearly) perfectly thin central slice illustrates the periodic property of DFT resulting in wrap-around effects in the reconstruction. In this example, a sphere (**A**) is backward projected at a 35° rotation angle around the y-axis. The central slice is inserted using a windowed sinc function of 2-pixel wide. The projected top-view (**B**) and side-view (**C**) are shown. The wrapping of the projected rays at the edges of the reconstruction are result of the periodicity of the DFT (**D**).

The window function currently implemented in our core library is a windowed sinc function and is applied along the normal of the central slice. Visually, this operation can be seen as "spreading" the slice values along its normal (see fig. 4.2). The window function is the product of two functions: the sinc function [168] and the blackman window function [169].

The inverse Fourier transform of the sinc function is the rectangular function. As a result, convolving the Fourier volume with the sinc function effectively applies a real-space rectangular mask to the reconstruction. Importantly, the orientation of the 1d sinc is equal to the direction along which the binary mask is applied. As such, applying the 1d convolution along the normal of the central slice results in a 1d rectangular function following the axis of projection of that slice. Moreover, the size of the rectangular function can be controlled by the frequency of the sinc, notably, a frequency of $1/n$, where n is the size of the reconstruction, results in a rectangular function of size n. Note that we define the sinc function as $sin(\pi x)/(\pi x)$.

The sinc function has an infinite impulse response in both positive and negative directions, which makes it impractical to use; convolving the 3d Fourier volume with the sinc function is extremely expensive, as inserting a single central slice affects every voxel in the volume. As shown in fig. 4.3, windowing the ideal sinc is an effective strategy to truncate the size of the sinc and removes most of the undesirable oscillations from the sinc truncation (known as Gibbs phenomenon [170, 171]). We are currently only supporting the blackman apodisation function [169, 172], which is simple to parameterize and has a good performance-to-cost ratio, but note that we do plan to support more window functions in the future.

Both the frequency of the sinc and the size of the blackman window are parameters of our window function. The sinc frequency defaults to $1/n$, but a larger frequency can be used to further limit the range of the projected rays in the reconstruction. The size of the blackman window, which we also express as a (normalised) frequency, is mostly used to control the performance-to-cost ratio of the insertion, *i.e.* large windows are more expensive

to compute, but have a better frequency response (that is assuming that the rectangular function is the ideal response).

The grid-driven approach allows one to easily design and apply large window functions that can be convolved with the (perfectly thin) central slices. In practice, each voxel coordinate is projected along the normal of the central slice about to be inserted, effectively decomposing the 3d coordinates into a 2d coordinate and a distance along the normal (see fig. 4.2). Note that to account for the Ewald Sphere curvature, the projection can be done along the normal of the sphere [41, 43]. The distance along the normal is used to compute the window function at that voxel, and, independently, the input data-points are interpolated at the projected 2d coordinate. In the data-driven approach, the convolution kernel effectively computes a 3d interpolation of the data, which can be difficult to scale. In the grid-driven approach, any kind of 2d interpolation method can be used. Note that we currently only support bilinear interpolation for this transformation, but we do plan to add more complex methods, including a 2d (windowed-)sinc interpolation for better accuracy [168]. Interestingly, while these more complex methods are more expensive to compute, they might result in better overall performance due to the fact that they allow the use of smaller oversampling factors.

As shown in the above code snippet 4.2, each voxel is assigned to one thread, which needs to go through and collect the contribution of every input central slice. While this is computationally expensive, we have implemented two optimisations that greatly improve the runtime performance of this approach. First, the 2d interpolation used to get the value of the central slice is only performed if the voxel is in the range of the central slice, *i.e.* if it is not zeroed-out by the window function of the slice. Thankfully, knowing whether a voxel is affected by a central slice can be as simple as computing a 3x3 matrix-vector product. This return-early optimisation further increases the importance of the size of the window function: the smaller the window function, the less voxels end up being affected by the central slices, resulting in less 2d interpolations having to be performed. Second, loading 3x3 rotation matrices requires 9 load-instructions (interestingly, the other transformation

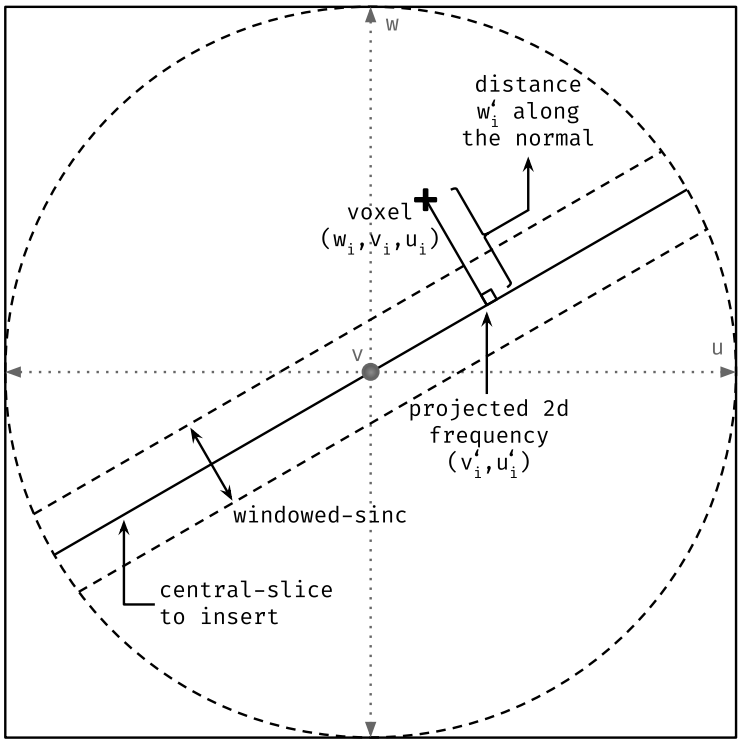


Figure 4.2: The insertion of a central slice in Fourier space using a grid-driven approach. This shows how a voxel is computed: 1) The voxel coordinate is projected onto the central slice. 2) The value of the central slice at the projected 2d frequency is queried using a 2d interpolation method. 3) The distance along the normal is used to compute the windowed sinc value for that voxel. In this example, the voxel is out of the windowed sinc of the central slice, so the window is 0, *i.e.* the central slice does not affect this voxel. 4) The interpolated value is multiplied with the windowed sinc value, and the result is added to the voxel. 5) These operations are repeated for each central slice.

matrices have an even number of elements, and can thus be loaded more efficiently). For cases that insert many slices at once, we have found that loading the matrices into registers ends up being particularly expensive for GPUs, even if the matrices are stored in constant or shared memory. As a result, we have added the option to encode the 2d rotation of the central slices into quaternions instead of matrices. Single-precision quaternions can be loaded to registers using a single 16-bytes vectorised load instruction (`load.vec4.f32` in CUDA's instruction set), which can result in GPU kernels running more than 30% faster than kernels using 3x3 rotation matrices.

In conclusion, the grid-driven approach scales very well with large window functions that allow to accurately control the extent of the projected rays. It uses 2d interpolation instead of the 3d interpolation used by the data-driven approach, and can be implemented efficiently on GPUs. When inserting only a few central slices, this approach is about three to four times slower than the data-driven approach, mostly due to the fact that the entire volume needs to be traversed. However, when inserting a large number (multiple hundreds) of central slices at once, the grid-driven approach runs as fast or faster than the data-driven approach..

4.1.3 Correcting for the apodisation of the reconstruction

Since a convolution in Fourier space corresponds to multiplication in real space, the resulting reconstruction from the Fourier inversion is the product of the final wanted reconstruction and the inverse Fourier transform of the convolution kernel. In practice, this results in an apodisation of the output reconstruction. In other words, this means that the reconstruction will be bright at the centre and gradually fade to be darker near the edges.

In the data-driven approach, the kernel is the 3d interpolation function. In this case, we can obtain the final wanted reconstruction by dividing the (real-space) output volume with the inverse Fourier transform of the 3d interpolation function (see fig. 4.3) [173, 174]. In the grid-driven approach, the convolution kernel is the product of the 2d interpolation function in

the projected plane, and the 1d windowed sinc along the normal of the projection axis. The former is best corrected as a pre-processing step by dividing the 2d (real-space) projections by the inverse Fourier transform of the 2d interpolation function. This effectively brightens the edges of the projections to anticipate for the apodisation effect resulting from the interpolation in Fourier space. The latter can be corrected by dividing the reconstruction, similarly to the data-driven approach, but note that this is often not required if a large enough windowed sinc is applied (see fig. 4.3) [168]. Similarly, the apodisation can be mostly ignored when large oversampling factors are used.

4.1.4 Normalising the multiplicity

When inserting central slices into the 3d grid, we often want to keep track of the backward projected weight, *i.e.* how much data was inserted into each voxel. We refer to this projected weight as the multiplicity. One important step of the backward projection consists in normalising this multiplicity to correct for inhomogeneities in the sampling density, or in other words, to even out the Fourier volume by downweighting frequencies that would otherwise dominate in the reconstruction.

By default, every Fourier component of the central slices is attached to a unit weight. However, most implementations [175, 176], including ours, also accept sampling functions, aka weights, describing the fraction of signal available in each input Fourier component about to be backward projected. These weights are projected alongside the central slices and can thus be used for the normalisation. In practice, if a voxel has a multiplicity greater than one after the backward projection, it means that multiple central slices contributed (valuable signal) to that 3d frequency, and we can therefore divide the value at that voxel by the multiplicity, effectively taking the weighted mean of all of the contributions at that voxel. On the contrary, if a voxel has a multiplicity that is less than one, we should not divide by the multiplicity, but instead keep it as is.

In SPA, we often assume that the multiplicity of voxels is greater than one or is zero (in which

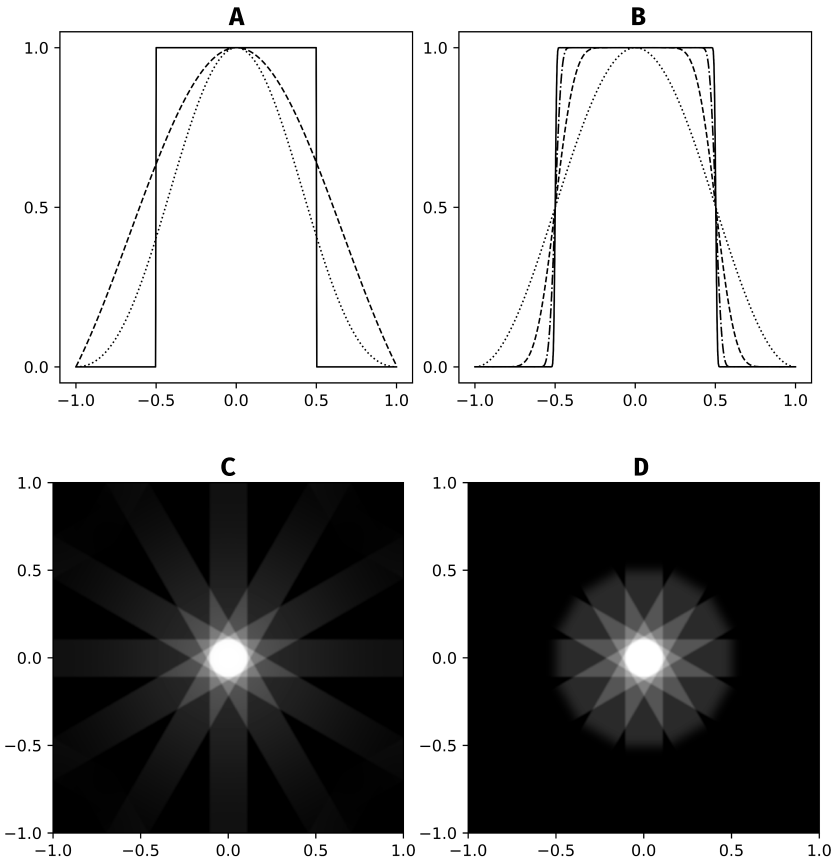


Figure 4.3: Effect on the reconstruction of the convolution kernel used in the backward projection. (**A**) The nearest interpolation kernel (*dashed*), linear interpolation kernel (*dotted*) and ideal sinc interpolation kernel (*solid*) are shown. The horizontal axis is the normalised size of the reconstruction; `[-0.5,0.5]` is the range of the reconstruction. (**B**) Response of windowed sinc interpolation kernels, with a sinc frequency of $1/n$, where n is the size of the reconstruction, and with blackman windows of size: $1/n$ (*dotted*), $5/n$ (*dashed*), $16/n$ (*dashdot*) and $64/n$ (*solid*). (**C** and **D**) Backward projection of a circle at angles `[0,30,60,90,120,160]` with a windowed sinc of size $1/n$ (**C**) and $16/n$ (**D**). The reconstruction was done using a twice oversampled grid, by zero-padding the input images by a factor of 2.

case the frequency was not sampled). This assumption holds 1) because central slices are discretised by small convolution kernels, where most of the weight is placed into only a few voxels, and 2) because thousands of slices are inserted into the same volume, resulting in the vast majority of voxels being sampled many times, leading to their multiplicity being larger than one. As a result, implementations may simply divide the volume with the multiplicity (while making sure to not divide by zero). In tomography, this assumption does not hold and care should be taken in handling voxels with a multiplicity that is less than one (e.g. [175].

In the code snippet 4.2, we are showing how the multiplicity can be normalised on-the-fly using the grid-driven approach. Note that this is only possible if all of the central slices are inserted at once. If multiple batches are to be inserted (which is often necessary for cases dealing with multiple hundreds or thousands of central slices), the multiplicity will have to be recorded separately.

4.2 Forward-projection

4.2.1 Fourier extraction

The forward projection of a volume along a given projection axis, *i.e.* integrating the real-space volume along an axis, can be efficiently implemented in Fourier space by simply "extracting", or sampling, from the Fourier volume, the central slice that lies on the normal of the projection axis. Sampling the central slice requires a 3d interpolation of the volume grid-points at the 3d frequencies corresponding to the 2d Fourier components of the central slice (see fig. 4.4 and code snippet 4.3 below). This interpolation can be any 3d interpolation method, but most cryoEM implementations use the trilinear interpolation method.

```
1 // For each pixel of every slice to extract:
```

```
void operator()(i64 slice_index, i64 v, i64 u) {
    // 1. Get the fftfreq at the (v, u) coordinate.
    auto fftfreq_2d = index_to_fftfreq(v, u);

    // 2. Rotate the central slice to place it in the grid
    // reference frame. This effectively does:
    // rotation_matrix * Vec{0, fftfreq_2d[0], fftfreq_2d[1]}
    auto fftfreq_3d = transform_slice_to_grid(
        fftfreq_2d, slice_forward_rotation[slice_index]);

    // 3. 3d interpolation of the grid-points at this frequency.
    auto interpolated_value = interpolate_at(fftfreq_3d);

    // 4. Save the value in the central slice.
    write_at(slice_index, v, u) = interpolated_value;
}
```

Snippet 4.3: Sampling a central slice from a 3d volume

4.2.2 On-the-fly windowed sinc convolution

In order to remove the contribution from outside the region of interest, the forward projection is often done on a masked (real-space) volume. For instance, it is best to mask out the regions above and below the sample before forward-projecting full tomograms.

Computing and applying this mask in real-space is a simple operation, but note that this mask can also be applied directly in Fourier-space by convolving the 3d Fourier volume with the Fourier transform of the mask. If the mask is a simple rectangular function along one dimension, which covers a surprisingly good amount of cases, the Fourier transform of the mask is a 1d sinc function, *i.e.* the convolution kernel is one-dimensional. Similarly

to the window function used in section 4.1.2, this sinc can be windowed to speed up the computation while still preserving most of its frequency response by resulting in a smooth rectangular function in real-space. More importantly, this convolution doesn't need to be done on the entire volume, but can instead be done on-the-fly, when sampling the 3d frequencies from the volume, *i.e.* only at the frequencies that need to be sampled for the extraction of the central slice(s). As a result, while this convolution does significantly increase the cost of the Fourier extraction, 1) it can still be efficiently implemented by a simple 1d weighted mean (see fig. 4.4) and 2) it can remove the need to compute (expensive) 3d (inverse) DFTs to generate the reconstruction and apply the real-space mask. This will become particularly relevant in the next section.

The current implementation in our library (see code snippet 4.4) implements such on-the-fly convolution, using the same windowed sinc function as described in section 4.1.2. At the time of writing, this convolution is limited to the w-axis (the depth) of the Fourier volume, and can thus only apply a real-space mask along a z-axis of the reconstruction (section 5.5 shows how such a mask can be useful).

```
// For every (v, u) pixel of every slice to sample,
// plus the extra dimension w to loop through the
// convolution w-window.
void operator()(i64 slice_index, i64 w, i64 v, i64 u) {
    // 1. Get the fftfreq at the (v, u) coordinate.
    auto fftfreq_2d = index_to_fftfreq(v, u);

    // 2. Rotate the central slice to place it in the grid
    // reference frame. This effectively does:
    // rotation_matrix * {0, fftfreq_2d[0], fftfreq_2d[1]}
    auto fftfreq_3d = transform_slice_to_grid(
        fftfreq_2d, slice_forward_rotation[slice_index]);
```

```
13
14      // 3. Add the w-offset. This is the position in the
15      // convolution w-kernel. Given the index w and the
16      // blackman window size, return the fftfreq offset, such as:
17      // fftfreq_w_offset = (w - (window_size / 2)) / recons_size
18      // For instance, recons_size=200, window_size=9:
19      // w=[0..8] -> fftfreq_w_offset=
20      // [-0.02,-0.015,-0.01,-0.005,0.,0.005,0.01,0.015,0.02]
21      auto fftfreq_w_offset =
22          w_index_to_fftfreq_offset(w, blackman_size);
23      fftfreq_3d[0] += fftfreq_w_offset; // offset the frequency
24
25      // 4. Compute the windowed sinc.
26      f64 convolution_weight = compute_windowed_sinc(
27          fftfreq_w_offset); // between 0 and 1
28
29      // 5. 3d interpolation of the grid-points at this frequency.
30      auto interpolated_value = interpolate_at(fftfreq_3d);
31
32      // 6. Convolve (weighted-sum).
33      // This +=" is atomic for multithreaded implementations.
34      slice_accessor(slice_index, v, u) +=
35          interpolated_value * convolution_weight;
36  }
```

Snippet 4.4: Sampling a central slice with the on-the-fly convolution along the w-axis

Interestingly, the convolution has the effect of "bringing" signal from other areas of the volume, more specifically, from the Fourier components that are above and below the extracted central slice(s). This is particularly significant for sparse Fourier volumes made of only a

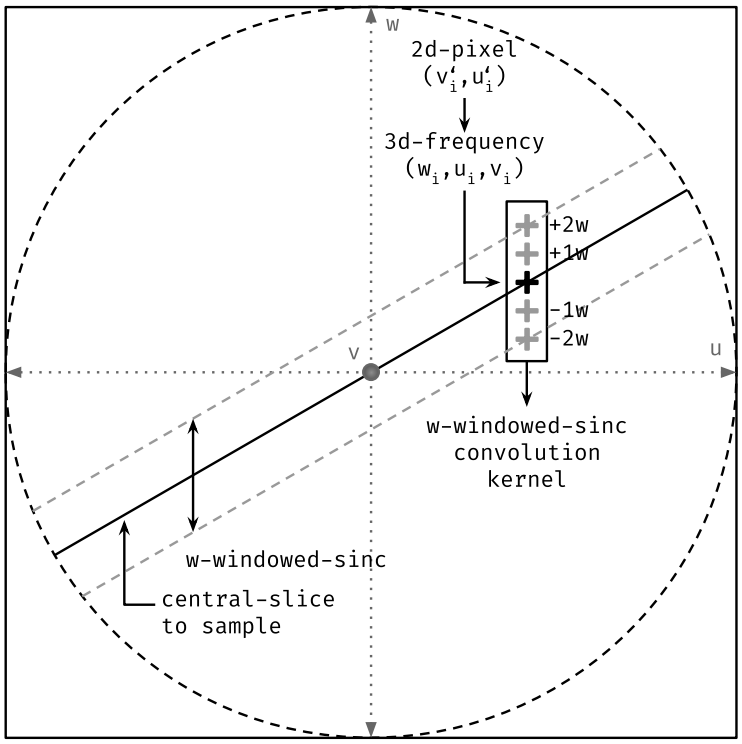


Figure 4.4: The extraction, *i.e.* sampling, of a central slice, with the optional on-the-fly w-windowed sinc convolution. This shows how a pixel of the central slice is computed: 1) The slice is rotated and placed into the grid reference frame; the pixel in the slice is transformed to a 3d frequency. 2) The grid is 3d interpolated to get the value at the given 3d frequency. 3) This value is convolved with the w-windowed sinc kernel, which requires to repeat step 2 for the 3d frequencies in the w-kernel.

few central slices. fig. 4.5 shows such an example, emphasising on how the w-windowed sinc convolution (or the real-space (smooth) z-rectangular mask) can result in having more signal in the forward projections.

4.3 Fused backward-forward projection

So far, we have discussed the backward and forward projections as two separate transformations. The backward projection projects 2d projections into a discrete 3d reconstruction, and the forward projection reprojects that 3d grid along some projection axes. In the previous sections, we have discussed how these transformations can be done in Fourier space, and how the central slices can be convolved with windowed sinc functions to accurately mask the real-space reconstruction directly from Fourier-space. As a result of computing everything from Fourier-space, new scenarios start to appear where the reconstruction does not need to be computed in order to get reprojections of that said reconstruction, *i.e.* the Fourier 3d volume (the inverse DFT of the reconstruction) is all we need.

Some algorithms, like projection matching, need to compute forward projections of the reconstruction, but don't actually need the reconstruction for anything else, nor do they need the 3d Fourier volume. In this case, the Fourier volume can be "virtual", *i.e.* only defined by the central slices that compose this volume. When the forward projection is needed and central slices need to be extracted, we can sample, on-the-fly, the virtual 3d Fourier volume as a continuous 3d space by simply collecting the contribution of the input slices at the sampled frequencies (fig. 4.6).

We have developed an operator that supports such fused backward-forward projections. This operator never needs to iterate through the 3d grid, nor does it need to allocate it, thereby potentially saving gigabytes of memory and allowing programs to work with large reconstructions that would not fit in memory. Interestingly, using a virtual volume is also more accurate, since the volume, and thus the signal of the input slices, is never actually

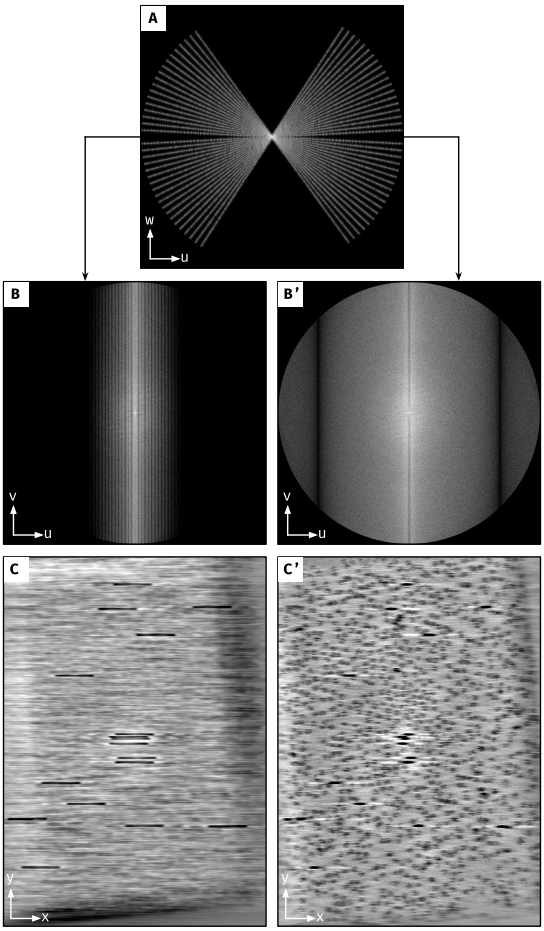


Figure 4.5: Forward projection a tomogram with and without the w-windowed sinc convolution. The tilt series, excluding the 0° tilt, is inserted into the 3d Fourier volume (**A**) with a windowed sinc of {`sinc=1/n, blackman=0.01`}, where `n` is the size of the twice oversampled Fourier volume. Then, the central slice at 0° tilt is sampled from that volume, without (**B**) or with (**B'**) the w-windowed sinc convolution parameterised with {`sinc=1/60, blackman=4/60`} (the sample thickness is equal to 60 pixels). The log-scale power-spectrums of **A**, **B** and **B'** are shown. Finally, the inverse DFT of this slice is computed (**C** and **C'**). Without the w-convolution, the signal is distorted perpendicular to the tilt axis (`v`|`y`-axis) due to the lack of projected signal perpendicular to the tilt. With the w-convolution, there is less distortion, due to the w-convolution pulling more signal from the input central slices, or equivalently, from the real-space (smooth) rectangular mask removing the noise from above and below the sample. `Dataset=EMPIAR-10304-tilt1`.

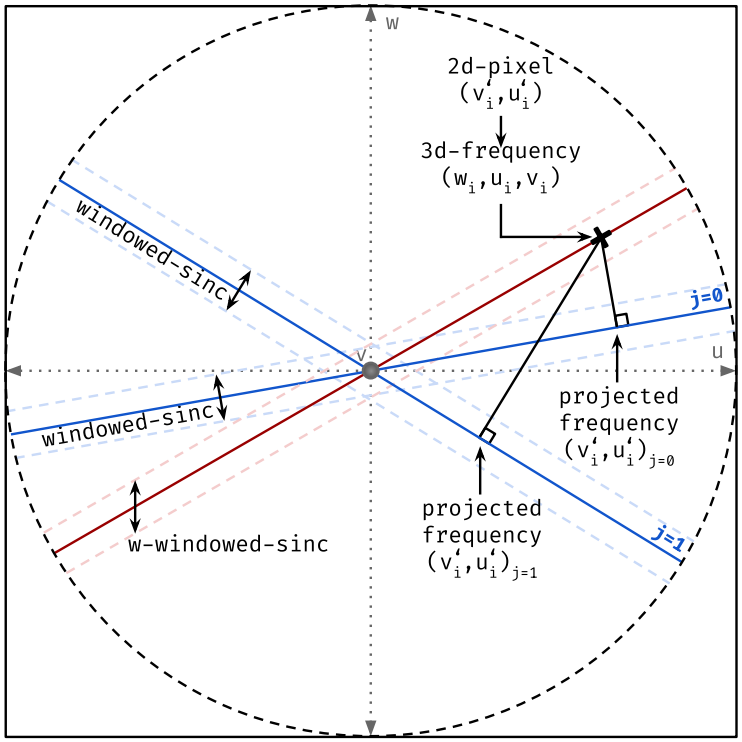


Figure 4.6: The extraction, *i.e.* sampling, of a central slice from a virtual volume made of other central slices. This effectively combines the insertion of central slices shown in fig. 4.2 and the extraction of central slices shown in fig. 4.4 into a single operation. Importantly, since these two steps are now a single operation, the "grid" can be encoded as a continuous (virtual) volume, where 3d frequencies can be queried on-the-fly without any kind of 3d interpolation involved.

discretised (see fig. 4.7) and because the 3d interpolation of the forward projection can be entirely skipped.

This fused operator is designed for cases that need to extract only a few central slices. It is for instance perfectly suited for aligning a tilt series with reprojections of the corresponding tomogram (see section 5.5). Indeed, the runtime performance of this approach gets worse than the non fused approach if thousands (or possibly tens or hundreds, depending on the size of the w-windowed sinc) of slices are to be sampled from the virtual volume. Indeed, for every output central slice, the operator needs to sample the virtual volume by collecting the signal of every input slice, using 2d interpolation. The w-windowed sinc convolution is particularly expensive given that an even larger region of the volume needs to be sampled to perform the convolution. This is as opposed to the usual forward projection method presented in section 4.2, where the contribution of the input central slices is already collected in the 3d grid, resulting in a much cheaper extraction step.

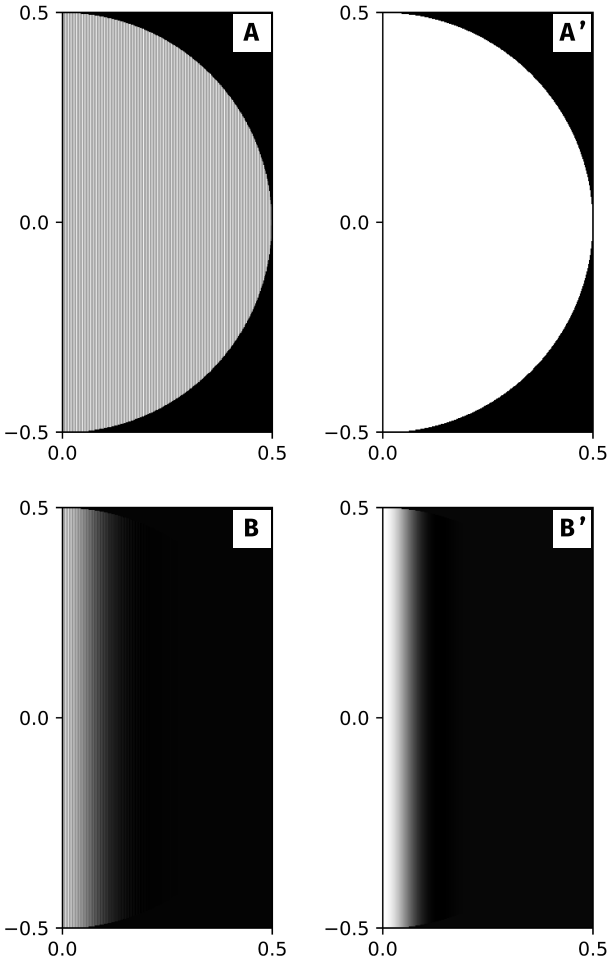


Figure 4.7: Sampling a discretised and virtual volume. A single slice of ones is inserted into the discrete volume (**A**-**B**) and virtual volume (**A'**-**B'**) at an angle of 45° tilt. The tilt is defined as a rotation around the vertical y-axis. The same central slice is sampled from the volumes (**A** and **A'**) and a central slice at 46° tilt is also sampled (**B** and **B'**). Frequencies above Nyquist are set to zero. This shows how the discretisation affects the sampled central slices.

Chapter 5

Tilt series alignment

5.1 Tilt series alignment and tomogram reconstruction 114

5.1.1 Tilt scheme 114

5.1.2 Tilt series alignment strategies 115

5.1.3 Geometric model and naming conventions 117

5.2 Coarse stage rotation and per-tilt 2d translation alignment 118

5.2.1 Translation alignment using pairwise cross-correlation 118

5.2.2 Stage rotation alignment using cosine stretching 119

5.2.3 Refining the per-tilt translations using cosine stretching 121

5.3 Specimen orientation within the microscope using CTF fitting 122

5.3.1 Initial CTF fitting on average power-spectrum 122

5.3.2 Fitting 3d specimen orientation using global CTF fitting 123

5.3.3 Defocus gradient 125

5.4 Estimating the specimen thickness 126

5.5 Refining the stage rotation and per-tilt translation 127

5.5.1 Projection matching in Fourier space using fused reprojection . . . 127

5.5.2 Per-tilt translation alignment 129

5.5.3 Difficulties with fitting the per-tilt stage rotation 131

5.6 Fourier-based tiled reconstruction 134

5.7 Measuring the quality of the alignment 134

In cryoET, the vitrified specimen is physically tilted respectively to the electron beam and 2d images of the same field-of-view are collected at different tilts. The relative orientations of the images within a tilt series is only approximately known from the data collection. The tilt series alignment refers to the process of determining these orientations, and is still subject to significant development efforts in the community. As part of testing and showcasing our core library, we started to implement a tilt series alignment program hoping to improve the accuracy, robustness and efficiency of existing approaches. This chapter only reports the initial stage of development and exploration we have done as of time of writing. The source code is available at https://github.com/thomasfrosio/quinoa.

5.1 Tilt series alignment and tomogram reconstruction

5.1.1 Tilt scheme

The distribution of the collected information within a 3d tomogram is the result of the so-called tilt scheme used during the tilt series acquisition [177]. This scheme essentially breaks down to: the total angular range covered by the tilt series, the angular increments between tilt images, the order in which each tilt image is collected and the electron exposure used to acquire each image.

To select a suitable tilt scheme, one must consider three key factors. The first is that at high tilts, the sample is thicker in the direction of the electron beam, leading to an increase of noise due to multiple scattering [29, 178]. The second is that beam-induced damage of biological samples is cumulative, and that high-resolution information is degraded first [51]. The third is whether the tomogram is intended for subtomogram averaging. Because of these two first factors, state-of-the-art tilt schemes share one main idea, which is that the low-tilt images, where the sample is at its thinnest relative to the beam, should be collected early in the tilt series, before radiation damage has accumulated. In parallel, if the tomogram is to be used for subtomogram averaging and if the biomacromolecules in the sample are organised in a wide and homogeneous range of orientations, it is usually possible to tolerate a smaller tilt range and/or a larger tilt increment [179, 180]. This allows a higher electron exposure per tilt, which results in a higher SNR in the images, thereby helping data-processing algorithms down the line. A larger fraction of the total exposure can also be dedicated to the low tilts [181], further increasing their SNR. In this case, the role of the high tilts is mostly reduced to providing the low and middle out-of-plane frequencies that are valuable for the subtomogram 3d alignment and classification.

5.1.2 Tilt series alignment strategies

In order to remove the large shifts that remain from the data acquisition, almost all tilt series alignments start with an initial coarse alignment by cross-correlating adjacent tilt images [182, 183]. Such simple pairwise Cross-Correlation (CC) can only provide a rough estimate of the translation between the two projected images because projections of the 3d sample along different projection (tilt) axes represent different point of views of the underlying 3d biological objects, resulting in these objects appearing slightly different from one image to the next [184]. Cosine stretching is a method commonly used to mitigate these differences [181, 182, 185], but while this works relatively well for low tilts and thin samples, this method quickly becomes inaccurate for high tilts and thicker samples [186]. Unfortunately, these inaccuracies are further increased by the systematic errors when computing the global translation of the images (see section 5.2.1). Despite its limitations, this technique is quite robust to large shifts and programs can safely assume a smooth trajectory of features from image to image once the coarse alignment is completed, which greatly facilitates subsequent alignment steps.

When it comes to finer alignments, one of the most widely used techniques is based on localising fiducial markers throughout the tilt series [187, 188]. These markers are often gold beads which are added to the sample and which are assumed to be immobile with respect to the biological objects. The spherical gold beads have two key characteristics that make them great markers for aligning cryoET samples. First, gold is opaque to the electron beam, which makes the beads highly contrasted features that can be easily detected, even at high tilts. Second, their projection looks like a disk from any angle, therefore, their centre always represents the same 3d point regardless of the tilt. As such, once we establish the 2d location of each marker and its correspondence across multiple images, which can be done entirely automatically on a coarsely aligned tilt series [186], we can then use standard optimisation techniques (e.g. robust regression [189]) to fit a geometric model of how the 3d points are projected onto the 2d images. As long as there's enough beads in the field of

view, these fiducial based alignments are fairly robust and require little to no user inputs. Furthermore, if there's a large enough number of beads, this method also allows a description of the local motion in the specimen. Unfortunately, there are situations where adding fiducial gold beads to the sample is prohibitive for either technical or biological reasons, such as in cryo-lamella sample preparation [190]. It is also unclear whether the use of such markers may interfere with the sample [191]. Maybe more importantly, these methods align the markers, not the biological objects, thus heavily relies on the assumption that the markers are immobile with respect to the objects of interest, which is (at least partially) inherently false when tracking local motion.

A variant of the marker-based alignment uses specimen features as virtual markers in cases where fiducial markers such as gold beads are undesirable. Similarly to the coarse alignment, these methods [192, 193] track 2d patches in successive tilt images using cross-correlation (and cosine stretching), and, similarly to the fiducial alignment, the centre of these patches is fed to an optimisation algorithm to fit a geometric model. Patch-tracking methods mostly differ in the size of the patches and in the way these patches are selected. Similar to fiducial alignment, patch-tracking methods can fit local motion, but inherently suffers from the same limitations as the coarse cross-correlation alignment. As a result, these methods tend to be inaccurate for the high tilts and can be inadequate for thick samples.

Similar to projection matching in SPA, the tilt series alignment can also be performed by aligning tilt images to reprojections of a 3d reconstruction made of already aligned images (see section 5.5) [187, 194, 195]. The advantage of this approach is that all specimen features within the region of interest naturally contribute to the alignment. Similar to patch-tracking, this approach can also be expanded to account for local motion by directly aligning subregions of the field-of-view [196] as part of a global optimisation algorithm (this will be further discussed in the next sections).

More recently, new methods for refining the tilt series geometry have appeared and are all

based on the idea that reprojections of the high SNR reconstructions from the subtomogram averaging can be cross-correlated with the projected particles in tilt images. These cross-correlations can be part of a global optimisation algorithm that aims to refine the initial geometric model of the tilt series. This is similar to particle polishing in SPA [53], but instead of refining the geometric model of the frames within a micrograph to polish the 2d particles, this refines the tilt series geometry to polish the 3d particles, *i.e.* the subtomograms, in the tomogram. Such refinement was first implemented in emClarity [197] and was later extended in M [45] and Relion [198]. While these refinement strategies can be extremely powerful, they are contingent upon having an already good initial tilt series alignment so that high quality subtomograms can be picked, aligned and averaged to a high enough quality reconstruction. Thus, while in good cases it is now possible to correct and improve the tilt series geometry, accurate starting alignments are still an important foundation for high-resolution STA.

5.1.3 Geometric model and naming conventions

The goal of the tilt series alignment is to find a good enough initial estimate of the geometric model, which encodes how points in each image in the tilt series are projected in the reconstruction, and vice-versa. The complexity of the geometric model varies from one program to the next, but in this initial phase of development, we will start with a simple model that encodes the sample as a rigid-body, which will hopefully pave the way for more complex models, e.g. deformation models.

As such, we have defined a volume-space transformation, which includes a 3d rotation and 2d translation, with no spatial resolution (it's a rigid body) and a per-tilt temporal resolution. The rotation can be expressed using extrinsic ZYX Euler angles. The z-rotation is the stage rotation, *i.e.* the rotation of the tilt-axis. The y-rotation is the tilt, and the x-rotation is the pitch, often referred to as tilt-x. Note that the z-translation is not necessary since the rotated specimen is projected along the z axis (that is assuming that the beam is parallel

to the z-axis).

For these initial experiments, we have purposely decided to ignore eventual magnification changes between images, as well as the per-tilt anisotropic magnification [38], since we believe these aberrations to not be significant for the resolution considered during the initial tilt series alignment (between 20 and 10 Å). Furthermore, we will assume that the tilt increment between images is known and accurate from the microscope stage, and that there are no additional pitch errors between images (the pitch has no temporal resolution). As a result, the tilt series alignment should solve for the global stage rotation, tilt and pitch, as well as the per-tilt stage rotation error and per-tilt 2d translation.

5.2 Coarse stage rotation and per-tilt 2d translation alignment

5.2.1 Translation alignment using pairwise cross-correlation

For the initial coarse alignment, the main focus is to roughly align the images to a common field-of-view by finding a good initial guess of the per-tilt 2d translations. This can be done by cross-correlating adjacent tilts together to find the relative shift between each pair of images [184]. In order to minimise the difference in the images due to their tilt, it is indeed best to align each image against a close tilt-neighbour, where the tilt difference is only a few degrees. As such, we are cross-correlating each tilt with its lower tilt-neighbour, and by this definition, the lowest tilt becomes the global reference and, as we will see, every image in the tilt series will ultimately be aligned to this reference.

Before cross-correlation, images are Fourier cropped to about 16 Å (or to a size of 1024 pixels, whichever comes first), tapered to ensure that the edges don't dominate the cross-correlation, highpass filtered to remove large contrast gradients, and are normalised. The cross-correlation is computed in Fourier space, which is inexpensive to compute, and the

subpixel position of the highest CC peak is retrieved by fitting a 3-points parabola along both axes around the maximum CC value [186]. The peak position is adjusted to account for the tilt difference between the two images. By subtracting the CC centre, we then obtain the 2d translation, which sets by how much the high tilt image should be translated to best match the low tilt image.

By the end of this procedure, we have computed the relative translation between each pair of images. To get the global translation of an image, *i.e.* the translation relative to the global reference (the lowest tilt), we need to add the relative translations of the lower tilts together, while accounting for the tilt difference between each pair. In order to maximise the field-of-view of the tilt series, we then normalise the average 2d translations to zero. The main disadvantage of this method is that the errors of each pairwise cross-correlation are accumulated to the higher tilts. These residual errors are indeed increasing systematically both with tilt angle and from the centre outwards [186], resulting in expected long-range systematic misalignments for the high tilts. For instance, the final translation of the highest tilt image is the result of the sum of the translations, and their attached residual errors, of the lower tilts up to the global reference.

Nevertheless, with no a priori knowledge other than the tilt increment(s) between images, this method can reliably compute a smooth trajectory from image to image, and is quite accurate for the low tilts images (see fig. 5.2). Moreover, it is robust against large shifts, and can in theory solve for shifts up to half the image size, or more if an additional padding is applied before the cross-correlation.

5.2.2 Stage rotation alignment using cosine stretching

A rough estimate of the stage rotation for the tilt series is usually known before starting the alignment. However, this estimate is rarely accurate and in order to further automate the tilt series alignment, we would like to be able to solve for the stage rotation while relying on as little user inputs as possible. Fortunately, the stage rotation can be derived from the

foreshortening of the projected features in the field-of-view [181]. As we know by now, the projected images are viewing the sample from different tilts. If the specimen were to be a perfectly thin (2d) object, the difference due to the tilt between two projected images would be equal to a scaling factor along the normal of the tilt axis of $cos(a_0)/cos(a_1)$ [184], where a_0 and a_1 are the tilt angles of the first and second image, respectively. In other words, tilting a perfectly thin specimen effectively shrinks the projection along the normal of the tilt axis and by a factor that only depends on the cosine of the tilt. Equivalently, projections from a non-tilted 2d specimen are a (cosine) stretched version of the projection of the tilted specimen. While things are obviously more complicated for 3d specimens since in this case the 3d transformation resulting from the tilt cannot accurately be modelled with a simple 2d transformation, we can still use the cosine scaling as an estimate of how the tilt affects projections.

In the context of finding the stage rotation, we cosine stretch the tilted images along the normal of a given tilt axis, and cross-correlate these stretched images with the reference, which is set to be the lowest tilt projection. If the reference has a zero degree tilt, the cosine scaling factor is $1/cos(a)$, where a is the tilt angle of the image about to be stretched. We then try to find the stage rotation that maximises the average normalised CC between the stretched tilts and the reference. This can be solved using a simple grid search over a range of stage rotations, but more efficient optimisation algorithms are also possible. We are currently using the subplex algorithm [199] (a variant of Nelder-Mead [200]) from the NLopt package [201], which works well for functions with few parameters (one in this case).

This method is more accurate with thin samples and if high tilts are included (40 degrees and higher) where the cosine varies more as a function of the tilt. We have never found this solution to fail, and it seems to be accurate enough for the coarse alignment (within one degree at most compared to the rotation from the projection matching, see section 5.5.3). Our implementation can be easily accelerated on the GPU and with GPU textures, to the point where we consider this step to be almost instantaneous. Note however if a GPU

isn't available, this method becomes significantly more expensive than a common line based alignment [202], which have been shown to work well (at least for the coarse alignment) in other packages [196]. Further experiments are required, but we are considering switching to using common lines for better performance. Furthermore, note that both this and the common line methods cannot distinguish between a stage rotation x and $x + 180$ degrees, since the cosine stretching is exactly the same for these two rotations. Section 5.3.2 presents how we can use the defocus gradient in the images to select the correct rotation.

5.2.3 Refining the per-tilt translations using cosine stretching

Once a good estimate of the stage rotation is available, we can use the pairwise alignment described in section 5.2.1, but with the cosine stretching the lower tilt image to compensate for the tilt difference between the pair of images. As discussed in the previous section, the cosine stretching is only a 2d approximation of how the tilt, a 3d transformation in volume-space, affects the projections. Furthermore, while it is (in theory) better than nothing, this correction works best for thin samples and quickly becomes inaccurate for thicker specimens, and therefore for high tilts where the sample is thicker relative to the beam (see fig. 5.2).

Practically, if no estimate of the stage rotation is provided, the coarse alignment starts with the simple per-tilt translation alignment discussed in section 5.2.1, and is immediately followed by the stage rotation alignment from section 5.2.2. At this point, we can start to mask the images to enforce a common FOV throughout the tilt series. This will be particularly relevant for the projection matching alignment (see section 5.5), but we have also found that applying such FOV mask during the coarse alignment helps to reduce the systematic drift in high tilts. Once the initial average stage rotation is found, or if the user provides it, we start the per-tilt translation alignment with the cosine stretching activated, then use the updated translations to run (a new pass of) the stage rotation alignment. We repeat this for 3 iterations, by which the rotation and translations should have fully converged. Note that after each alignment, the common FOV mask is updated.

To conclude, the coarse alignment can find a good initial estimate of the average stage rotation and per-tilt translation. At this point, per-tilt rotation errors, as well as the specimen tilt and pitch, have not been considered. Importantly, this alignment is lightweight and takes only a few seconds (most of which is spent on the stage rotation alignment) to complete, as long as a GPU is available on the system, and requires little user inputs.

5.3 Specimen orientation within the microscope using CTF fitting

5.3.1 Initial CTF fitting on average power-spectrum

Before solving for the specimen orientation, we first need to derive the average defocus, astigmatism, and optionally the phase-shift, from the average spectrum of the tilt series [33, 34, 203]. Moreover, as the imaged sample content does not change significantly throughout the tilt series, the 1d background and envelope, that will later be used for a more complex optimisation, can be derived from the same spectrum.

To reduce the negative interference between regions with different defoci, whether due to the defocus difference between tilt images or due to the defocus gradient within a tilt image, the average 1d spectrum is computed only from the patches that are within a given z-range from the centre of the tilt series' FOV (the default is `[-50,50]`nm, where the pivot point is at 0) [197]. Moreover, due to their increased defocus gradient and increased thickness, we also added the option to exclude patches coming from the high tilts or patches that are above a given accumulated electron exposure.

Once the average 2d power spectrum is ready, the (isotropic) rotational average is computed, and a B-spline with three control points is fitted through it and subtracted to remove most of the background [162]. Then, the defocus, and optionally the phase shift, are estimated by maximising the normalised cross-correlation between simulated CTF curves

and the background-subtracted 1d spectrum.

The background and envelope are then more accurately estimated by fitting a new B-spline with three control points through the CTF zeros and peaks of a slightly gaussian smoothed version of original 1d average spectrum. The frequencies of the zeros and peaks are derived from the newly estimated CTF parameters. The optimisation is then restarted with the new background and envelope, but this time also accounting for the astigmatism. We use the method described in [204] and compute the anisotropy corrected rotational average allowing us to fit the astigmatism while still using the same 1d approach as described above. While this method requires recomputing the rotational average at each evaluation of the cost function, it necessitates almost twice as less memory and uses 1d simulated CTFs as opposed to 2d CTFs. In the absence of astigmatism, the anisotropy corrected rotational averaging is of course equivalent to the classic rotational averaging over circles and only needs to be computed once for the entire optimisation.

In summary, the cost function computes the anisotropy corrected rotational average of the 2d average spectrum, subtracts the background and applies the envelope. Then, the normalised cross-correlation is computed between the corrected 1d spectrum and the simulated 1d CTF for the current defocus and phase shift. The L-BFGS algorithm [132] is used to maximise the cost and the derivatives are estimated numerically using the central finite difference method. Since, the background and envelope correction depends on the accuracy of the optimised parameters (that is the defocus, astigmatism and phase shift), the optimisation is repeated a few times, and both the background and envelope are updated each time.

5.3.2 Fitting 3d specimen orientation using global CTF fitting

At this point, an average defocus, astigmatism, and optionally phase shift, as well as the 1d background and envelope, have all been derived from the average power spectrum. These can then be used as a starting point for a more complex optimisation where both the per-tilt defocus and the specimen orientation can be derived. Note that while the stage rotation

can technically be fitted, we have found it to be less accurate than the initial alignment (section 5.2.2) and are thus limiting this optimisation to the specimen inclination angles, *i.e.* the tilt and pitch.

Each tilt image is divided into overlapping 2d patches, giving us the spatial resolution needed to fit the specimen inclination. Fitting a CTF to the spectrum of each patch individually would be inaccurate due to the little signal present in each patch, *i.e.* Thon rings are barely above the noise level. To improve the fit accuracy, the spectrums are instead subjected to a common set of constraints and are analysed as part of the same optimisation function. Wrap was the first to introduce such optimisation [162], and we are implementing a similar algorithm. Briefly, the 2d overlapping patches are extracted and their spectrums are computed. To save memory, only the values within the fitted frequency range are saved. The cost function consists in computing the anisotropy corrected rotational average of each spectrum, and cross-correlating each 1d spectrum with its corresponding simulated CTF, which depends on the currently tested inclination angles, astigmatism, phase shift (optionally), and the defocus of the tilt image the spectrum belongs to. The final cost is simply the average of the normalised CC score between the spectrum and simulated CTF of every patch and of every tilt.

We are using the L-BFGS algorithm [132] to maximise the cost and the derivatives are estimated numerically using the central finite difference method. Interestingly, computing the gradients for each per-tilt defocus can be simplified. Indeed, instead of computing the gradients by reevaluating the cost function for each defocus one by one, we use the fact that the defocus of a tilt only affects the spectrums of that tilt, meaning that these gradients can be estimated in parallel by computing the cost function only twice. As a result, to compute the gradients, we evaluate the cost function twice for each global parameter (that is the two inclination angles, the two astigmatism parameters and the phase shift) and twice for the per-tilt defoci.

The current implementation stops here, however, we are planning to add a few changes that should hopefully increase the accuracy of the fit and lead to a more accurate per-tilt defocus and specimen inclination angles. First, the phase-shift and astigmatism are currently constant through the tilt series, but we could add a temporal resolution, like in Wrap [162], to model cases where the overall exposure is fractionated over a long time. Second, knowing that the background and envelope are only the result of the initial fit and that they depend on both the specimen orientation and the CTF parameters, recomputing a new and improved average power-spectrum and deriving a new background and envelope could lead to a more accurate fit, especially for cases with a large initial tilt offset. Indeed, since we now have a good estimate of the specimen orientation and per-tilt defocus, thereby knowing the defocus at the centre of every patch, we could scale the spectrums to a common defocus before taking the average, which should significantly increase the Thon rings, thus the accuracy of the fit. The optimisation could also benefit from the more accurate stage rotation coming from the projection matching alignment (see section 5.5.3), which in turn could benefit from a more accurate specimen inclination (see next section).

5.3.3 Defocus gradient

The direction of the tilt axis is often a source of confusion and the wrong direction is sometimes entered, or similarly, one image axis can be inadvertently flipped during the image acquisition, or tilt angles can be entered incorrectly in the inverse order.

We have reimplemented the defocus gradient check, also called the handedness check, that was originally implemented in emClarity [197]. This check simply consists in recomputing the initial CTF fit of the average power spectrum (section 5.3.1), but using two average spectrums, one from patches above the pivot point (by default, within the `[50,100]` nm z-range), and the other from patches below the pivot point (by default, within `[-100,-50]` nm z-range). The spectrum above the pivot point should be closer to focus than the original spectrum at the pivot point, which in turn should be closer to focus than the spectrum

below the pivot point. If this is not the case, the user is warned that something is likely wrong with the provided tilt geometry. Note that in cases where the stage rotation is not specified and fully computed from the cross-correlation alignment (section 5.2.2), the user is notified and the rotation is flipped to match the defocus gradient.

5.4 Estimating the specimen thickness

In order to prepare for the projection matching alignment, we need to find the position of the specimen within the tomogram, and thus the specimen thickness, so that the regions above and below the specimen can be masked out before reprojecting the tomogram. If the thickness is underestimated, some parts of the specimen will not be forward projected and will be missing from the projected reference, thus from the alignment. On the other hand, if the thickness is overestimated, noise will be added to the protected reference, resulting in a large distortion perpendicular to the tilt (see fig. 4.5). In both cases, inaccurate thickness estimates can lead to significantly less accurate projection matching alignments. Some programs usually default to a fixed value (in number of pixels, in the worst case), which of course doesn't fit all samples, or ask users to provide a thickness estimate for the projection matching. We instead would like to automatise this process.

We have started to develop an algorithm to both estimate the thickness of a non-inclined specimen and its z-centre. Note however that we are still not satisfied with this algorithm, as we have found it to be less robust than anticipated.

The current implementation starts by reprojecting the xz-profile of a low resolution tomogram. This is done efficiently thanks to the fused reprojection we have developed (see section 4.3) by means of creating a virtual Fourier volume containing every central slice of the tilt series and sampling the central slice rotated around the u-axis by 90 degrees. The reprojection is then slightly cropped to remove small reconstruction artefacts, and a 2d median filter and 2d gaussian blur are applied to remove most of the background noise.

As presented in the previous section, a good estimate of the specimen tilt and pitch can be derived from the Thon rings of the tilt images, thus resulting in the specimen being placed horizontally in the tomogram. Therefore, we can assume that the backward projected specimen is aligned with the rows of the current reprojection. As a result, we compute the variance of every row in the reprojection, resulting in a 1d vector describing the variability of the projected density along the depth of the tomogram.

We now need to find the region of that vector that belongs to the specimen. We start by z-centering the specimen using the centre of mass of the variances. We then initially tried to fit a step or Tuckey window [173] to the variances, but this was abandoned as it seemed inaccurate and unreliable, especially for specimens with density distributions that do not match these windows. We instead chose to 1) compute the gradient of the variances, to remove the background systematic noise and reconstruction artefacts, and focus on the density variations across the depth of the tomogram, and 2) select the highest peaks in the gradient to seed the window, and then 3) dilate the window to the neighbouring significant peaks (see fig. 5.1). This current method can be quite accurate, but as it is not as robust as we would like, further work will be needed to reliably automate this process.

5.5 Refining the stage rotation and per-tilt translation

5.5.1 Projection matching in Fourier space using fused reprojection

The last step of our tilt series alignment consists in aligning tilt images to reprojections of a 3d reconstruction made of already aligned images. This is called projection matching and already has been implemented in other tilt series alignment packages [187, 195, 196]. Our original goal was to improve the efficiency of these alignments to then fit a more complex geometry. Our implementation mostly differs from existing implementations in that the 3d reconstruction is never actually reconstructed, but instead the entire backward projection of the reference images and the forward projection of the resulting reconstruction is fused

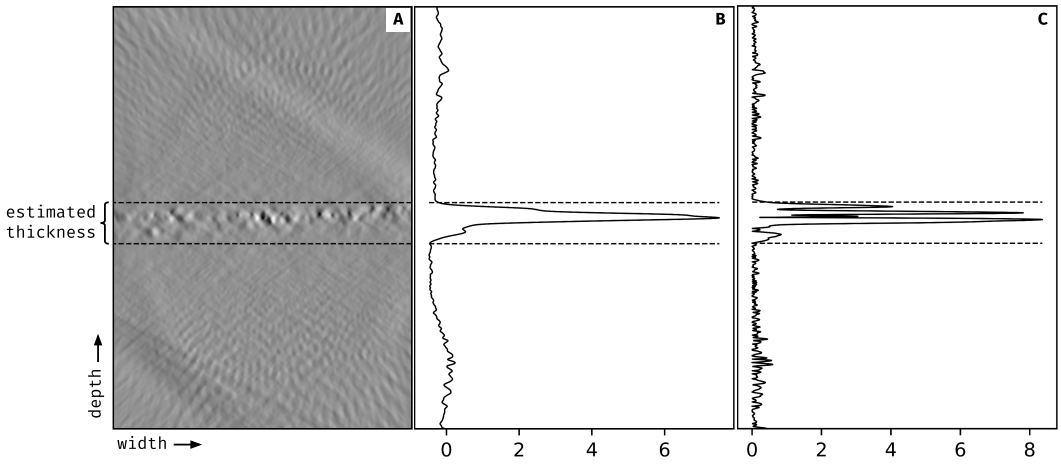


Figure 5.1: Automated thickness estimate. The preliminary tomogram (using the per-tilt translations of the coarse alignment and the specimen tilt and pitch from the CTF alignment) is reprojected along its height (the y-axis) (**A**). The variance along the width (**B**) and the gradient of these variances (**C**) are computed and L2-normalised. The current algorithm selects the first and last peak from **C** above $\sigma = 1.5$, then dilate the window by selecting the neighbouring peaks (up to 10 peaks away) that are above $\sigma = 0.5$. `Dataset=EMPIAR-10304-tilt1`.

into a single operation in Fourier space. This was extensively described in section 4.3, and in this chapter we will see how this is used in the tilt series alignment.

There are various strategies in determining which tilt images are used as references to align a given target image. For instance, a set of randomly chosen images could be made references, and the remaining targets could be aligned against the (virtual) tomogram made of these references. As a starting point, we have instead chosen to follow the strategy used by existing packages [195, 196], which consists in aligning tilt images sequentially from low to high (absolute) tilts, with the lowest tilt being set as the global reference. When an image is aligned, it is added to the list of reference images used to compute the projected reference against which the next (higher) tilt image will be aligned. As such, more reference images are used for the alignment of the higher tilts, with the projected reference of the last tilt containing every other image in the tilt series (the target is never included in its projected reference). The advantage of this strategy is that it makes the projection matching non-iterative. In other words, the alignment of a given target tilt image only depends on the already aligned lower tilts.

5.5.2 Per-tilt translation alignment

Here is a brief description of how the translation between a projected reference and its corresponding target image is computed. First, the reference images and the target image are zero-tapered to prepare for the zero-padding, highpass filtered to remove large contrast gradients, L2-normalise and the common FOV mask is applied. The mask is particularly important to make sure that the same region of the sample is being compared. Then images are zero-padded by a factor of two to oversample the Fourier space and increase the accuracy of the interpolation that will be used for the projection. The DFT is computed and the central slices are phase-shifted to the origin to account for the rotation centre. At this point the sampling function, aka weights, of the target and of each reference is computed and applied. These weights contain the CTF, the exposure filter [51] approximated using the

CTF B-factor, and a per-tilt weighting to approximate the relative SNR from the electron mean-free-path [205]. A per-tilt CTF is used, which should still be appropriate for the entire image given that the FOV mask drastically reduces the defocus gradient in the high tilts and that only frequencies up to 10Å are used for the alignment. Then, the projected reference and its corresponding projected weights (squared) are computed using the fused reprojection. To prepare for the cross-correlation, the target is then multiplied with the weights of the projected reference, and the projected reference is in turn multiplied with the weights of the target. Finally, both spectrums are phase-shifted back to their rotation centre, the inverse DFT is computed, and the resulting images are cropped back to their original size. At this point, this becomes a classic Fourier-based normalised cross-correlation of two images, where the highest cross-correlation peak is selected to derive the translation that should be applied to the target image.

Various optimisations have been implemented to improve the performance and memory profile of the alignment. One of which being that only the newly added reference is prepared and the other references are reused from the alignment of the previous target images. As a result, finding the translation of every tilt usually takes between 3 to 15 seconds, depending on the resolution, number of images in the tilt series and hardware available on the machine. Most of this time is spent during the fused reprojection, more specifically, during the w-windowed sinc convolution applied during the forward projection (see section 4.3). Interestingly, this results in thicker specimens being faster to reproject since the size of the convolution window is equal to $1/n$, where n is the thickness of the specimen.

We hope to further improve the runtime performance of this algorithm by supporting higher precision interpolation methods for the Fourier space interpolation. Indeed, while a 4x4 or 8x8 windowed sinc interpolation, e.g. a Lanczos interpolation [206], is more expensive to compute than the linear interpolation we currently use, it allows to reduce the oversampling to the minimum requirement ($\sqrt{2}$ oversampling), thereby greatly reducing the memory footprint of the alignment and having less frequencies to sample, which would hopefully

result in a better overall runtime performance.

The advantage of a Fourier based projection is that, as we have seen, the weights of the tilt images can be easily backward and forward projected alongside the data. These forward projected weights can then be used to estimate how much signal is in each Fourier component being cross-correlated. While we originally included the per-tilt CTF in the sampling functions, we ended up (temporarily) removing it after observing that the CTF weighting was removing too much of the valuable low frequencies perpendicular to the tilt axis. Indeed, most of the forward projected signal perpendicular to the tilt axis is at the low to middle frequencies, where the CTF is mostly close to zero. Applying the CTF removes most of this projected signal, resulting in large distorsions perpendicular to the tilt axis and making the alignment too sensitive to noise and unreliable. When the CTF is ignored, the CC peak is significantly less distorted perpendicular to the tilt axis, resulting in sharper and stronger peaks that strongly anchor the images while still allowing the significant translation (systematic) errors from the coarse alignment to be resolved (see fig. 5.2). To still benefit from the CTF weighting, we are planning to only use the CTF above a certain frequency. For instance, only the CTF values after the first CTF peak could be used. Another solution could be to apply a smooth wedge mask that could cancel the CTF for the frequencies close to the normal of the tilt axis.

5.5.3 Difficulties with fitting the per-tilt stage rotation

When extracting the CC peak position to derive the translation between each tilt image and their corresponding projected reference, we also retrieve the normalised CC peak value. These values are accumulated, giving a score to the alignment. We have added an optimisation that then tries to maximise this score by varying the stage rotation of the tilt series. Once the optimiser finds the best stage rotation, we simply retrieve the corresponding per-tilt translations that were found for this stage rotation.

While this works and should in theory be more robust than a common line alignment [202],

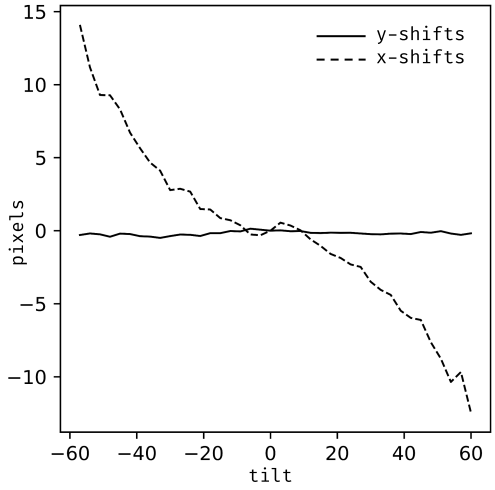


Figure 5.2: Shifts, in unbinned pixels, added to the coarse geometry during the projection matching alignment. The tilt axis is rotated and aligned with the y-axis. The shifts parallel to the tilt axis (*solid*) are close to zero, *i.e.* the coarse alignment was accurate, at least according to the projection matching alignment. On the other hand, the shifts orthogonal to the tilt axis (*dashed*) are subject to significant systematic errors increasing both with tilt angle and from the centre outwards. `Dataset=EMPIAR-10304-tilt1`.

it revealed that solving for a per-tilt stage rotation would be impractical given the current implementation. Indeed, with the current setup, the optimised cost is the result of projection matching the entire tilt series (except the global reference). In the case of an average stage rotation, the optimiser only needs to evaluate the cost function a few times (20 times at most). However, as the mechanical rotation of the microscope stage and the estimated orientation of the tilt axis are imperfect, the assumed stage orientation can be randomly off in every tilt, thus the optimiser would need to optimise not one parameter, but one for every tilt.

In the future, as we start to introduce more complex geometries, e.g. a deformation model to account for sample distortions as well as magnification anisotropy, the need for efficient projection matching will become even more relevant. We are currently investigating a way to accelerate both the projection matching and the computation of the derivatives so that such optimisation would be possible. Other non-iterative strategies, for instance, strategies based on common-lines [202] and the Fourier Mellin transform [207, 208, 209], are also considered.

In the meantime, as a temporary solution and realising that the tilt axis could vary ever so smoothly from tilt to tilt, we are fitting a B-spline of 2 to 5 points through the tilt range, resulting in an alignment between 5 and 25 minutes (which is still far from satisfying). This revealed a larger than expected variation in the microscope stage, where the tilt axis seem to vary by up to 0.5 degrees across the tilt series in the worst cases. These results correspond to the geometry from the fiducial alignment in IMOD [186]. The fact that these errors vary with the tilt is still unclear to us, but could be the result of an hysteresis in the mechanical stage.

5.6 Fourier-based tiled reconstruction

While this is not the role of the tilt series alignment per say, programs usually offer the possibility to reconstruct full tomograms for visualisation or for further processing, e.g. denoising, segmentation or particle picking [210, 211, 212, 213]. We currently offer a very minimal setup regarding the reconstruction; tomograms are reconstructed with no CTF correction, no deconvolution and no denoising.

The current implementation uses the Fourier-based tiled reconstruction from Wrap [162], where a uniform grid of small, cubic and overlapped volumes are reconstructed and the central part of these cubes are inserted into the final tomogram. EMAN2 employs a similar strategy but uses a different method for inserting the cubes in the final tomogram [214]. Wrap uses a gridding-algorithm (see section 4.1.1) to reconstruct the cubes, we instead use the grid-driven insertion method described in section 4.1.2 with a large windowed sinc to further minimise the apodisation effect and remove all wrap-around effects.

5.7 Measuring the quality of the alignment

Unfortunately we were not able to provide any validation metric at this stage of the development. The simple (but still expensive) approach could be to test, for a wide range of samples, our program against existing reference alignments. To do so, one could process small datasets through a standardised STA pipeline and analyse, at various stages of the pipeline, the quality of the reconstructions by measuring the Fourier Shell Correlation (FSC) between the reconstructions obtained with our alignment and the reconstructions from reference alignments (e.g. IMOD [186], AreTomo2 [196]).

We would also like to develop a method that automatically evaluates the quality of the alignment on a per tilt series basis. While measuring the quality of the alignment when fiducial markers are present is well established, it is much more difficult to accurately and

automatically measure when using projection matching.

Bibliography

[1] Jeremy Berg et al. *Biochemistry*. 10th edition. W. H. Freeman, May 15, 2023. 1120 pp. ISBN: 978-1-319-49850-4.

[2] Tracy L. Nero, Michael W. Parker, and Craig J. Morton. "Protein structure and computational drug discovery". In: *Biochemical Society Transactions* 46.5 (Sept. 2018), pp. 1367–1379. ISSN: 0300-5127. DOI: 10.1042/BST20180202.

[3] Engin Cukuroglu et al. "Hot spots in protein-protein interfaces: towards drug discovery". In: *Progress in Biophysics and Molecular Biology* 116.2 (2014), pp. 165–173. ISSN: 1873-1732. DOI: 10.1016/j.pbiomolbio.2014.06.003.

[4] Joseph M. Jez. "Revisiting protein structure, function, and evolution in the genomic era". In: *Journal of Invertebrate Pathology* 142 (Jan. 2017), pp. 11–15. ISSN: 1096-0805. DOI: 10.1016/j.jip.2016.07.013.

[5] Dapeng Xiong et al. "Implications of disease-related mutations in protein-protein interfaces". In: *Current opinion in structural biology* 72 (Feb. 2022), pp. 219–225. ISSN: 0959-440X. DOI: 10.1016/j.sbi.2021.11.012.

[6] Vladimir N. Uversky. "Protein intrinsic disorder and structure-function continuum". In: *Progress in Molecular Biology and Translational Science* 166 (2019), pp. 1–17. ISSN: 1878-0814. DOI: 10.1016/bs.pmbts.2019.05.003.

[7] Helen M. Berman, Brinda Vallat, and Catherine L. Lawson. "The data universe of structural biology". In: *IUCrJ* 7 (Pt 4 July 1, 2020), pp. 630–638. ISSN: 2052-2525. DOI: 10.1107/S205225252000562X.

[8] R. M. Sweet. "Introduction to crystallography". In: *Methods in Enzymology* 114 (1985), pp. 19–46. ISSN: 0076-6879. DOI: 10.1016/0076-6879(85)14004-8.

[9] Marco Mazzorana, Elizabeth J. Shotton, and David R. Hall. "A comprehensive approach to X-ray crystallography for drug discovery at a synchrotron facility - The

example of Diamond Light Source". In: *Drug Discovery Today. Technologies* 37 (Dec. 2020), pp. 83–92. ISSN: 1740-6749. DOI: 10.1016/j.ddtec.2020.10.003.

[10] Isabelle Boscaro-Clarke, François Cesmat, and Sarah Tridgell. *Diamond-II: Advancing Science*. 2019.

[11] Graeme Winter et al. "DIALS as a toolkit". In: *Protein Science* 31.1 (2022), pp. 232–250. ISSN: 1469-896X. DOI: 10.1002/pro.4224.

[12] Stephen K. Burley et al. "Protein Data Bank: A Comprehensive Review of 3D Structure Holdings and Worldwide Utilization by Researchers, Educators, and Students". In: *Biomolecules* 12.10 (Oct. 4, 2022), p. 1425. ISSN: 2218-273X. DOI: 10.3390/biom12101425.

[13] Hong-Wei Wang and Jia-Wei Wang. "How cryo-electron microscopy and X-ray crystallography complement each other". In: *Protein Science: A Publication of the Protein Society* 26.1 (Jan. 2017), pp. 32–39. ISSN: 1469-896X. DOI: 10.1002/pro.3022.

[14] G. McMullan, A. R. Faruqi, and R. Henderson. "Chapter One - Direct Electron Detectors". In: *Methods in Enzymology*. Ed. by R. A. Crowther. Vol. 579. The Resolution Revolution: Recent Advances In cryoEM. Academic Press, Jan. 1, 2016, pp. 1–17. DOI: 10.1016/bs.mie.2016.05.056.

[15] Sjors H. W. Scheres. "A Bayesian View on Cryo-EM Structure Determination". In: *Journal of Molecular Biology* 415.2 (Jan. 13, 2012), pp. 406–418. ISSN: 0022-2836. DOI: 10.1016/j.jmb.2011.11.010.

[16] J. Dubochet et al. "Electron microscopy of frozen water and aqueous solutions". In: *Journal of Microscopy* 128.3 (1982), pp. 219–237. ISSN: 1365-2818. DOI: 10.1111/j.1365-2818.1982.tb04625.x.

[17] Robert M. Glaeser and Richard J. Hall. "Reaching the Information Limit in Cryo-EM of Biological Macromolecules: Experimental Aspects". In: *Biophysical Journal* 100.10

(May 18, 2011), pp. 2331–2337. ISSN: 0006-3495. DOI: 10.1016/j.bpj.2011.04.018.

[18] Richard Henderson. "Avoiding the pitfalls of single particle cryo-electron microscopy: Einstein from noise". In: *Proceedings of the National Academy of Sciences* 110.45 (Nov. 5, 2013), pp. 18037–18041. DOI: 10.1073/pnas.1314449110.

[19] Joachim Frank. "The envelope of electron microscope transfer functions for partially coherent illumination". In: *Optik* 38 (1973), pp. 519–536.

[20] John C. H. Spence. *Experimental high-resolution electron microscopy*. Ed. by Internet Archive. New York : Oxford University Press, 1988. 458 pp. ISBN: 978-0-19-505405-7.

[21] Robert M. Glaeser et al. "Precise beam-tilt alignment and collimation are required to minimize the phase error associated with coma in high-resolution cryo-EM". In: *Journal of Structural Biology* 174.1 (Apr. 1, 2011), pp. 1–10. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2010.12.005.

[22] F. Zemlin et al. "Coma-free alignment of high resolution electron microscopes with the aid of optical diffractograms". In: *Ultramicroscopy* 3 (Jan. 1, 1978), pp. 49–60. ISSN: 0304-3991. DOI: 10.1016/S0304-3991(78)80006-0.

[23] Jasenko Zivanov et al. "New tools for automated high-resolution cryo-EM structure determination in RELION-3". In: *eLife* 7 (2018), e42166. ISSN: 2050-084X. DOI: 10.7554/eLife.42166.

[24] R. Henderson. "The potential and limitations of neutrons, electrons and X-rays for atomic resolution microscopy of unstained biological molecules". In: *Quarterly Reviews of Biophysics* 28.2 (May 1995), pp. 171–193. ISSN: 0033-5835. DOI: 10.1017/s003358350000305x.

[25] Koji Yonekura et al. "Electron energy filtering significantly improves amplitude contrast of frozen-hydrated protein at 300kV". In: *Journal of Structural Biology* 156.3 (Dec. 1, 2006), pp. 524–536. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2006.07.016.

[26] Takanori Nakane et al. "Single-particle cryo-EM at atomic resolution". In: *Nature* 587.7832 (Nov. 2020), pp. 152–156. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2829-0.

[27] R. M. Glaeser. "Specimen Behavior in the Electron Beam". In: *Methods in Enzymology* 579 (2016), pp. 19–50. ISSN: 1557-7988. DOI: 10.1016/bs.mie.2016.04.010.

[28] David B. Williams and C. Barry Carter. "Elastic Scattering". In: *Transmission Electron Microscopy: A Textbook for Materials Science*. Ed. by David B. Williams and C. Barry Carter. Boston, MA: Springer US, 2009, pp. 39–51. ISBN: 978-0-387-76501-3. DOI: 10.1007/978-0-387-76501-3_3.

[29] David B. Williams and C. Barry Carter. "High-Resolution TEM". In: *Transmission Electron Microscopy: A Textbook for Materials Science*. Ed. by David B. Williams and C. Barry Carter. Boston, MA: Springer US, 2009, pp. 483–509. ISBN: 978-0-387-76501-3. DOI: 10.1007/978-0-387-76501-3_28.

[30] D. J. De Rosier and A. Klug. "Reconstruction of Three Dimensional Structures from Electron Micrographs". In: *Nature* 217.5124 (Jan. 1968), pp. 130–134. ISSN: 1476-4687. DOI: 10.1038/217130a0. URL: https://www.nature.com/articles/217130a0.

[31] Peter W. Hawkes. "The Electron Microscope as a Structure Projector". In: *Electron Tomography: Three-Dimensional Imaging with the Transmission Electron Microscope*. Ed. by Joachim Frank. Boston, MA: Springer US, 1992, pp. 17–38. ISBN: 978-1-4757-2163-8. DOI: 10.1007/978-1-4757-2163-8_2.

[32] Joachim Frank. *Electron Tomography: Three-Dimensional Imaging with the Transmission Electron Microscope*. Boston, MA: Springer US, 1992. 1 p. ISBN: 978-1-4757-2163-8. DOI: 10.1007/978-1-4757-2163-8.

[33] Alexis Rohou and Nikolaus Grigorieff. "CTFFIND4: Fast and accurate defocus estimation from electron micrographs". In: *Journal of Structural Biology*. Recent Advances

in Detector Technologies and Applications for Molecular TEM 192.2 (Nov. 1, 2015), pp. 216–221. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2015.08.008.

[34] Kai Zhang. "Gctf: Real-time CTF determination and correction". In: *Journal of Structural Biology* 193.1 (2016), pp. 1–12. ISSN: 10958657. DOI: 10.1016/j.jsb.2015.11.003.

[35] F. Zernike. "How I Discovered Phase Contrast". In: *Science* 121.3141 (Mar. 11, 1955), pp. 345–349. DOI: 10.1126/science.121.3141.345.

[36] "Data processing". In: *Single-particle Cryo-EM of Biological Macromolecules*. Ed. by Robert M Glaeser, Eva Nogales, and Wah Chiu. 2053-2563. IOP Publishing, 2021, 4-1 to 4–91. ISBN: 978-0-7503-3039-8. DOI: 10.1088/978-0-7503-3039-8ch4.

[37] F. Thon. "Notizen: Zur Defokussierungsabhängigkeit des Phasenkontrastes bei der elektronenmikroskopischen Abbildung". In: *Zeitschrift für Naturforschung A* 21.4 (Apr. 1, 1966), pp. 476–478. ISSN: 1865-7109. DOI: 10.1515/zna-1966-0417.

[38] J. Zivanov, T. Nakane, and S. H. W. Scheres. "Estimation of high-order aberrations and anisotropic magnification from cryo-EM data sets in RELION-3.1". In: *IUCrJ* 7.2 (Mar. 1, 2020), pp. 253–267. ISSN: 2052-2525. DOI: 10.1107/S2052252520000081.

[39] Earl J. Kirkland. *Advanced Computing in Electron Microscopy*. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-33259-4 978-3-030-33260-0. DOI: 10.1007/978-3-030-33260-0.

[40] M. M. J. Treacy and D. Van Dyck. "A surprise in the first Born approximation for electron scattering". In: *Ultramicroscopy*. Special Issue: Gertrude F. Rempfer 100th Birthday Memorial 119 (Aug. 1, 2012), pp. 57–62. ISSN: 0304-3991. DOI: 10.1016/j.ultramic.2011.11.012.

[41] David J. DeRosier. "Correction of high-resolution data for curvature of the Ewald sphere". In: *Ultramicroscopy* 81.2 (Mar. 1, 2000), pp. 83–98. ISSN: 0304-3991. DOI: 10.1016/S0304-3991(99)00120-5.

[42] P. P. Ewald. "Introduction to the dynamical theory of X-ray diffraction". In: *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography* 25.1 (Jan. 1, 1969), pp. 103–108. ISSN: 0567-7394. DOI: 10.1107/S0567739469000155.

[43] Matthias Wolf, David J. DeRosier, and Nikolaus Grigorieff. "Ewald sphere correction for single-particle electron microscopy". In: *Ultramicroscopy* 106.4 (Mar. 1, 2006), pp. 376–382. ISSN: 0304-3991. DOI: 10.1016/j.ultramic.2005.11.001.

[44] Christopher J. Russo and Richard Henderson. "Ewald sphere correction using a single side-band image processing algorithm". In: *Ultramicroscopy* 187 (Apr. 1, 2018), pp. 26–33. ISSN: 0304-3991. DOI: 10.1016/j.ultramic.2017.11.001.

[45] Dimitry Tegunov et al. "Multi-particle cryo-EM refinement with M visualizes ribosome-antibiotic complex at 3.5 Å in cells". In: *Nature Methods* 18.2 (Feb. 2021), pp. 186–193. ISSN: 1548-7105. DOI: 10.1038/s41592-020-01054-7.

[46] Dongjie Zhu et al. "Pushing the resolution limit by correcting the Ewald sphere effect in single-particle Cryo-EM reconstructions". In: *Nature Communications* 9.1 (Apr. 19, 2018), p. 1552. ISSN: 2041-1723. DOI: 10.1038/s41467-018-04051-9.

[47] Joachim Frank. "Averaging of low exposure electron micrographs of non-periodic objects". In: *Ultramicroscopy* 1.2 (Jan. 1, 1975), pp. 159–162. ISSN: 0304-3991. DOI: 10.1016/S0304-3991(75)80020-9.

[48] Joachim Frank. "Single-Particle Reconstruction of Biological Molecules—Story in a Sample (Nobel Lecture)". In: *Angewandte Chemie International Edition* 57.34 (2018), pp. 10826–10841. ISSN: 1521-3773. DOI: 10.1002/anie.201802770.

[49] Robert M. Glaeser. "Limitations to significant information in biological electron microscopy as a result of radiation damage". In: *Journal of Ultrastructure Research* 36.3 (Aug. 1, 1971), pp. 466–482. ISSN: 0022-5320. DOI: 10.1016/S0022-5320(71)80118-1.

[50] Axel F. Brilot et al. "Beam-induced motion of vitrified specimen on holey carbon film". In: *Journal of Structural Biology* 177.3 (Mar. 2012), pp. 630–637. ISSN: 1095-8657. DOI: 10.1016/j.jsb.2012.02.003.

[51] Timothy Grant and Nikolaus Grigorieff. "Measuring the optimal exposure for single particle cryo-EM using a 2.6 Å reconstruction of rotavirus VP6". In: *eLife* 4 (May 29, 2015). Ed. by Wesley I Sundquist, e06980. ISSN: 2050-084X. DOI: 10.7554/eLife.06980.

[52] Tanmay A. M. Bharat et al. "Advances in Single-Particle Electron Cryomicroscopy Structure Determination applied to Sub-tomogram Averaging". In: *Structure* 23.9 (Sept. 1, 2015), pp. 1743–1753. ISSN: 1878-4186. DOI: 10.1016/j.str.2015.06.026.

[53] J. Zivanov, T. Nakane, and S. H. W. Scheres. "A Bayesian approach to beam-induced motion correction in cryo-EM single-particle analysis". In: *IUCrJ* 6.1 (Jan. 1, 2019), pp. 5–17. ISSN: 2052-2525. DOI: 10.1107/S205225251801463X.

[54] Haixiao Gao et al. "Dynamics of EF-G interaction with the ribosome explored by classification of a heterogeneous cryo-EM dataset". In: *Journal of Structural Biology*. Time-Resolved Imaging of Macromolecular Processes and Interactions 147.3 (Sept. 1, 2004), pp. 283–290. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2004.02.008.

[55] Charles Barchet et al. "Focused classifications and refinements in high-resolution single particle cryo-EM analysis". In: *Journal of Structural Biology* 215.4 (Dec. 1, 2023), p. 108015. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2023.108015.

[56] Michael Radermacher. "Weighted Back-Projection Methods". In: *Electron Tomography: Three-Dimensional Imaging with the Transmission Electron Microscope*. Ed. by Joachim Frank. Boston, MA: Springer US, 1992, pp. 91–115. ISBN: 978-1-4757-2163-8. DOI: 10.1007/978-1-4757-2163-8_5.

[57] Joseph W. Goodman. *Introduction to Fourier Optics*. Roberts and Company Publishers, 2005. 520 pp. ISBN: 978-0-9747077-2-3.

[58] Richard Anthony Crowther, D. J. DeRosier, and Aaron Klug. "The reconstruction of a three-dimensional structure from projections and its application to electron microscopy". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 317.1530 (1970), pp. 319–340. DOI: 10.1098/rspa.1970.0119.

[59] Friedrich Förster and Reiner Hegerl. "Structure Determination *In Situ* by Averaging of Tomograms". In: *Methods in Cell Biology*. Vol. 79. Cellular Electron Microscopy. Academic Press, Jan. 1, 2007, pp. 741–767. DOI: 10.1016/S0091-679X(06)79029-X.

[60] John AG Briggs. "Structural biology *in situ* — the potential of subtomogram averaging". In: *Current Opinion in Structural Biology*. Theory and simulation / Macromolecular assemblies 23.2 (Apr. 1, 2013), pp. 261–267. ISSN: 0959-440X. DOI: 10.1016/j.sbi.2013.02.003.

[61] V. Knauer, R. Hegerl, and W. Hoppe. "Three-dimensional reconstruction and averaging of 30 S ribosomal subunits of Escherichia coli from electron micrographs". In: *Journal of Molecular Biology* 163.3 (Jan. 25, 1983), pp. 409–430. ISSN: 0022-2836. DOI: 10.1016/0022-2836(83)90066-9.

[62] Friedrich Förster, Bong-Gyoon Han, and Martin Beck. "Chapter Eleven - Visual Proteomics". In: *Methods in Enzymology*. Ed. by Grant J. Jensen. Vol. 483. Cryo-EM, Part C: Analyses, Interpretation, and Case studies. Academic Press, Jan. 1, 2010, pp. 215–243. DOI: 10.1016/S0076-6879(10)83011-3.

[63] Peijun Zhang. "Advances in cryo-electron tomography and subtomogram averaging and classification". In: *Current Opinion in Structural Biology*. Cryo electron microscopy - Biophysical and computational methods - Biophysical and computational methods - Part B 58 (Oct. 1, 2019), pp. 249–258. ISSN: 0959-440X. DOI: 10.1016/j.sbi.2019.05.021.

[64] Vladan Lučić, Friedrich Förster, and Wolfgang Baumeister. "STRUCTURAL STUDIES BY ELECTRON TOMOGRAPHY: From Cells to Molecules". In: *Annual Review*

of Biochemistry 74.1 (2005), pp. 833–865. DOI: 10.1146/annurev.biochem.73.011303.074112.

[65] B. F. McEwen, K. H. Downing, and R. M. Glaeser. "The relevance of dose-fractionation in tomography of radiation-sensitive specimens". In: *Ultramicroscopy* 60.3 (Oct. 1995), pp. 357–373. ISSN: 0304-3991. DOI: 10.1016/0304-3991(95)00082-8.

[66] Timothy Grant, Alexis Rohou, and Nikolaus Grigorieff. "cisTEM, user-friendly software for single-particle image processing". In: *eLife* 7 (Mar. 7, 2018). Ed. by Edward H Egelman, e35383. ISSN: 2050-084X. DOI: 10.7554/eLife.35383.

[67] Florian K. M. Schur et al. "An atomic model of HIV-1 capsid-SP1 reveals structures regulating assembly and maturation". In: *Science (New York, N.Y.)* 353.6298 (July 29, 2016), pp. 506–508. ISSN: 1095-9203. DOI: 10.1126/science.aaf9620.

[68] *Python Software Foundation. Python Language Reference*. URL: http://www.python.org.

[69] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2.

[70] Pauli Virtanen et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature Methods* 17.3 (Mar. 2020), pp. 261–272. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0686-2.

[71] *PyPI, The Python Package Index*. URL: https://pypi.org/.

[72] Rafael Camara et al. "Agile Global Software Development: A Systematic Literature Review". In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. SBES '20. New York, NY, USA: Association for Computing Machinery, Dec. 21, 2020, pp. 31–40. ISBN: 978-1-4503-8753-8. DOI: 10.1145/3422392.3422411.

[73] Rubén Grande, Aurora Vizcaíno, and Félix O. García. "Is it worth adopting DevOps practices in Global Software Engineering? Possible challenges and benefits". In: *Computer Standards & Interfaces* 87 (Jan. 1, 2024), p. 103767. ISSN: 0920-5489. DOI: 10.1016/j.csi.2023.103767.

[74] Ida Bagus Kerthyayana Manuaba. "Combination of Test-Driven Development and Behavior-Driven Development for Improving Backend Testing Performance". In: *Procedia Computer Science*. The 4th International Conference on Computer Science and Computational Intelligence (ICCSCI 2019) : Enabling Collaboration to Escalate Impact of Research Results for Society 157 (Jan. 1, 2019), pp. 79–86. ISSN: 1877-0509. DOI: 10.1016/j.procs.2019.08.144.

[75] João Paulo Biazotto et al. "Technical debt management automation: State of the art and future perspectives". In: *Information and Software Technology* 167 (Mar. 1, 2024), p. 107375. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2023.107375.

[76] Martyn D. Winn et al. "Overview of the CCP4 suite and current developments". In: *Acta Crystallographica Section D: Biological Crystallography* 67.4 (2011), pp. 235–242. ISSN: 09074449. DOI: 10.1107/S0907444910045749.

[77] Graeme Winter et al. "DIALS: Implementation and evaluation of a new integration package". In: *Acta Crystallographica Section D: Structural Biology* 74 (2018), pp. 85–97. ISSN: 20597983. DOI: 10.1107/S2059798317017235.

[78] Stephan Nickell et al. "TOM software toolbox: Acquisition and analysis for electron tomography". In: *Journal of Structural Biology* 149.3 (2005), pp. 227–234. ISSN: 10478477. DOI: 10.1016/j.jsb.2004.10.006.

[79] "MRC2020: improvements to Ximdisp and the MRC image-processing programs". In: *IUCrJ* 10.5 (Sept. 1, 2023), pp. 579–583. ISSN: 2052-2525. DOI: 10.1107/S2052252523006309.

[80] Vandana Singh and Lila Holt. "Learning and best practices for learning in open-source software communities". In: *Computers & Education* 63 (Apr. 1, 2013), pp. 98–108. ISSN: 0360-1315. DOI: 10.1016/j.compedu.2012.12.002.

[81] Vishal Midha and Anol Bhattacherjee. "Governance practices and software maintenance: A study of open source projects". In: *Decision Support Systems* 54.1 (Dec. 1, 2012), pp. 23–32. ISSN: 0167-9236. DOI: 10.1016/j.dss.2012.03.002.

[82] Colin C. Venters et al. "Sustainable software engineering: Reflections on advances in research and practice". In: *Information and Software Technology* 164 (Dec. 1, 2023), p. 107316. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2023.107316.

[83] Helen M. Berman et al. "The Protein Data Bank". In: *Nucleic Acids Research* 28.1 (2000), pp. 235–242. ISSN: 03051048. DOI: 10.1093/nar/28.1.235.

[84] Ardan Patwardhan et al. "Data management challenges in three-dimensional EM". In: *Nature Structural & Molecular Biology* 19.12 (Dec. 2012), pp. 1203–1207. ISSN: 1545-9985. DOI: 10.1038/nsmb.2426.

[85] Mark D. Wilkinson et al. "Comment: The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* 3 (2016). ISSN: 20524463. DOI: 10.1038/sdata.2016.18.

[86] D. S. Hollman et al. "mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards". In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Nov. 2019, pp. 60–70. DOI: 10.1109/P3HPC49587.2019.00011.

[87] Maya B. Gokhale. "Chapter 10 - Programming Data Parallel FPGA Applications Using the SIMD/Vector Model". In: *Reconfigurable Computing*. Ed. by Scott Hauck and André Dehon. Systems on Silicon. Burlington: Morgan Kaufmann, Jan. 1, 2008, pp. 219–230. DOI: 10.1016/B978-012370522-8.50015-7.

[88] H. Carter Edwards and Christian R. Trott. "Kokkos: Enabling Performance Portability Across Manycore Architectures". In: *2013 Extreme Scaling Workshop (xsw 2013)*. Aug. 2013, pp. 18–24. DOI: 10.1109/XSW.2013.7.

[89] *Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms*. `https://gitlab.com/libeigen/eigen`. URL: https://gitlab.com/libeigen/eigen.

[90] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113. ISSN: 00010782. DOI: 10.1145/1327452.1327492.

[91] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. New York, NY, USA: Association for Computing Machinery, June 22, 2019, pp. 10–19. ISBN: 978-1-4503-6719-6. DOI: 10.1145/3315508.3329973.

[92] *Numpy, Universal functions (ufunc)*. URL: https://numpy.org/doc/stable/reference/ufuncs.html.

[93] John Nickolls et al. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53. ISSN: 15427730. DOI: 10.1145/1365490.1365500.

[94] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in Science and Engineering* 12.3 (2010), pp. 66–72. ISSN: 15219615. DOI: 10.1109/MCSE.2010.69.

[95] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. Feb. 5, 2018. DOI: 10.48550/arXiv.1502.05767. arXiv: 1502.05767[cs,stat].

[96] Konstantin Laufer and George K. Thiruvathukal. "Scientific Programming: The Promises of Typed, Pure, and Lazy Functional Programming: Part II". In: *Computing in Science & Engineering* 11.5 (Sept. 2009), pp. 68–75. ISSN: 1558-366X. DOI: 10.1109/MCSE.2009.147.

[97] J. Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (Jan. 1, 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98.

[98] Hartmut Kaiser et al. "HPX - The C++ Standard Library for Parallelism and Concurrency". In: *Journal of Open Source Software* 5.53 (Sept. 2, 2020), p. 2352. ISSN: 2475-9066. DOI: 10.21105/joss.02352.

[99] *C++ Expression Templates*. URL: https://en.cppreference.com/w/cpp/language/expression_template.

[100] Johan Mabille, Sylvain Corlay, and Wolf Vollprecht. *C++ tensors with broadcasting and lazy computing*. https://github.com/xtensor-stack/xtensor. 2016. URL: https://github.com/xtensor-stack/xtensor.

[101] Alex Suhan et al. *LazyTensor: combining eager execution with domain-specific compilers*. Feb. 25, 2021. DOI: 10.48550/arXiv.2102.13267. arXiv: 2102.13267[cs].

[102] Ao Li et al. "Automatic Horizontal Fusion for GPU Kernels". In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Apr. 2022. DOI: 10.1109/CGO53902.2022.9741270.

[103] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. Mar. 16, 2016. DOI: 10.48550/arXiv.1603.04467. arXiv: 1603.04467[cs].

[104] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. DOI: 10.48550/arXiv.1912.01703. arXiv: 1912.01703[cs, stat].

[105] Stefan Behnel et al. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 31–39. ISSN: 1558-366X. DOI: 10.1109/MCSE.2010.118.

[106] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2021. DOI: 10.1109/CGO51591.2021.9370308.

[107] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. New York, NY, USA: Association for Computing Machinery, Nov. 15, 2015, pp. 1–6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162.

[108] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. ISSN: 00361445. DOI: 10.1137/141000671.

[109] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (Apr. 2019), pp. 827–841. ISSN: 1558-2183. DOI: 10.1109/TPDS.2018.2872064.

[110] *Swift Differential Programming*. URL: https://github.com/apple/swift/blob/main/docs/DifferentiableProgramming.md.

[111] *JuliaDiff*. URL: https://juliadiff.org.

[112] *rust-gpu*. URL: https://github.com/EmbarkStudios/rust-gpu.

[113] *Mojo programming language*. URL: https://github.com/modularml/mojo.

[114] Tianqi Chen et al. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. Oct. 5, 2018. DOI: 10.48550/arXiv.1802.04799. arXiv: 1802.04799[cs].

[115] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[116] Ian Healy, Peter Giordano, and Wafa Elmannai. "Branch Prediction in CPU Pipelining". In: *2023 IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. Oct. 2023, pp. 0364–0368. DOI: 10.1109/UEMCON59035.2023.10316163.

[117] Sebastien Varrette et al. "Management of an academic HPC cluster: The UL experience". In: *Proceedings of the 2014 International Conference on High Performance Computing and Simulation, HPCS 2014* (2014), pp. 959–967. DOI: 10.1109/HPCSim.2014.6903792.

[118] John D. Owens et al. "A survey of general-purpose computation on graphics hardware". In: *Computer Graphics Forum* 26.1 (2007). ISSN: 14678659.

[119] Butler W. Lampson and Kenneth A. Pier. "A processor for a high-performance personal computer". In: *Proceedings - International Symposium on Computer Architecture* (1980), pp. 146–160. ISSN: 10636897. DOI: 10.1145/800053.801920.

[120] Alan Jay Smith. "Cache Memories". In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530. ISSN: 15577341. DOI: 10.1145/356887.356892.

[121] Linda Null. *Essentials of Computer Organization and Architecture*. 6th edition. Burlington, MA: Jones & Bartlett Learning, Apr. 27, 2023. 750 pp. ISBN: 978-1-284-25943-8.

[122] *xsimd, C++ wrappers for SIMD intrinsics*. URL: https://github.com/xtensor-stack/xsimd.

[123] D. Lemire and L. Boytsov. "Decoding billions of integers per second through vectorization". In: *Software - Practice and Experience* 45.1 (2015), pp. 1–29. ISSN: 1097024X. DOI: 10.1002/spe.2203.

[124] *Auto-vectorization in GCC*. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[125] *Auto-vectorization in LLVM*. URL: https://llvm.org/docs/Vectorizers.html.

[126] Georg Hager and Gerhard Wellein. "Introduction to high performance computing for scientists and engineers". In: *Introduction to High Performance Computing for Scientists and Engineers* (2010), pp. 1–323. DOI: 10.1201/EBK1439811924.

[127] *CUDA compute plateform*. URL: https://developer.nvidia.com/about-cuda.

[128] Sulaymon Eshkabilov. "MEX Files, C/C++, and Stand-Alone Applications". In: *Beginning MATLAB and Simulink: From Beginner to Pro*. Ed. by Sulaymon Eshkabilov. Berkeley, CA: Apress, 2022, pp. 299–320. ISBN: 978-1-4842-8748-4. DOI: 10.1007/978-1-4842-8748-4_4.

[129] *AMD ROCm, Open software stack that includes a broad set of programming models, tools, compilers, libraries, and runtimes for AI and HPC solution development on AMD GPUs*. URL: https://github.com/ROCm.

[130] *oneAPI, an open, cross-industry, standards-based, unified, multiarchitecture, multi-vendor programming model that delivers a common developer experience across accelerator architectures*. URL: https://github.com/oneapi-src.

[131] Woosuk Shin, Kwan-Hee Yoo, and Nakhoon Baek. "Large-Scale Data Computing Performance Comparisons on SYCL Heterogeneous Parallel Processing Layer Implementations". In: *Applied Sciences* 10.5 (Jan. 2020), p. 1656. ISSN: 2076-3417. DOI: 10.3390/app10051656.

[132] Jorge Nocedal. "Updating quasi-Newton matrices with limited storage". In: *Mathematics of Computation* 35.151 (1980), pp. 773–782. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1980-0572855-7.

[133] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. arXiv.org. Sept. 15, 2016. URL: https://arxiv.org/abs/1609.04747v2.

[134] Soeren Laue. *On the Equivalence of Automatic and Symbolic Differentiation*. arXiv.org. Apr. 5, 2019. URL: https://arxiv.org/abs/1904.02990v4.

[135] Allan M. M. Leal. *autodiff, a modern, fast and expressive C++ library for automatic differentiation*. https://autodiff.github.io. 2018. URL: https://autodiff.github.io.

[136] Robin Hogan. *Adept 2: Combined array and automatic differentiation library in C++*. https://github.com/rjhogan/Adept-2. 2012. URL: https://github.com/rjhogan/Adept-2.

[137] Roy Frostig, Matthew Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing". In: 2018. URL: https://mlsys.org/Conferences/doc/2018/146.pdf.

[138] Laurent Hascoet and Valérie Pascual. "The Tapenade automatic differentiation tool: Principles, model, and specification". In: *ACM Transactions on Mathematical Software* 39.3 (May 3, 2013), 20:1–20:43. ISSN: 0098-3500. DOI: 10.1145/2450153.2450158.

[139] William Moses and Valentin Churavy. "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 12472–12485.

[140] *OpenACC, a programming standard for parallel computing designed to simplify parallel programming of heterogeneous CPU/GPU systems*. URL: https://www.openacc.org.

[141] *CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda.

[142] *Variant: a type-safe union for C++17 (v8)*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0088r3.html.

[143] *Link time optimization in GCC*. URL: https://gcc.gnu.org/onlinedocs/gccint/LTO.html.

[144] *function_ref: a type-erased callable reference*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p0792r14.html.

[145] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. ISSN: 10709924. DOI: 10.1109/99.660313.

[146] *NVIDIA nvcc*. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[147] *NVIDIA nvc++*. URL: https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html.

[148] *GCC, the GNU Compiler Collection*. URL: https://gcc.gnu.org.

[149] *Clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org.

[150] *NVIDIA nvrtc, Runtime Compiler*. URL: https://docs.nvidia.com/cuda/nvrtc/index.html.

[151] Ben Barsdell and Kate Clark. *A single-header C++ library for simplifying the use of CUDA Runtime Compilation (NVRTC)*. `https://github.com/NVIDIA/jitify`. 2016. URL: https://github.com/NVIDIA/jitify.

[152] *CMake: A Powerful Software Build System*. URL: https://cmake.org.

[153] Alex Rogozhnikov. "Einops: Clear and Reliable Tensor Manipulations with Einstein-like Notation". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=oapKSVM2bcj.

[154] *C++ `reinterpret_cast`*. URL: https://en.cppreference.com/w/cpp/language/reinterpret_cast.

[155] *C++ `std::shared_ptr`*. URL: https://en.cppreference.com/w/cpp/memory/shared_ptr.

[156] *C++ Trival Class*. URL: https://en.cppreference.com/w/cpp/language/classes#Trivial_class.

[157] Neil MacIntosh and Stephan Lavavej. "P0122R7, span: bounds-safe views for sequences of objects". In: *Programming Language C++, Library Working Group* (2018). URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0122r7.pdf.

[158] Matteo Frigo and Steven G. Johnson. "The design and implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. ISSN: 00189219. DOI: 10.1109/JPROC.2004.840301.

[159] Pawel A. Penczek. "Chapter One - Fundamentals of Three-Dimensional Reconstruction from Projections". In: *Methods in Enzymology*. Ed. by Grant J. Jensen. Vol. 482. Cryo-EM, Part B: 3-D Reconstruction. Academic Press, Jan. 1, 2010, pp. 1–33. DOI: 10.1016/S0076-6879(10)82001-4.

[160] Guang Tang et al. "EMAN2: An extensible image processing suite for electron microscopy". In: *Journal of Structural Biology*. Software tools for macromolecular mi-

croscopy 157.1 (Jan. 1, 2007), pp. 38–46. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2006.05.009.

[161] Ali Punjani et al. "cryoSPARC: algorithms for rapid unsupervised cryo-EM structure determination". In: *Nature Methods* 14.3 (Mar. 2017), pp. 290–296. ISSN: 1548-7105. DOI: 10.1038/nmeth.4169.

[162] Dimitry Tegunov and Patrick Cramer. "Real-time cryo-electron microscopy data preprocessing with Warp". In: *Nature Methods* 16.11 (Nov. 2019), pp. 1146–1152. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0580-y.

[163] John M. Pauly. "Non-Cartesian Reconstruction". In: 2005. URL: https://api.semanticscholar.org/CorpusID:26324888.

[164] J. D. O'Sullivan. "A fast sinc function gridding algorithm for fourier inversion in computer tomography". In: *IEEE transactions on medical imaging* 4.4 (1985), pp. 200–207. ISSN: 0278-0062. DOI: 10.1109/TMI.1985.4307723.

[165] J. G. Pipe and P. Menon. "Sampling density compensation in MRI: rationale and an iterative numerical solution". In: *Magnetic Resonance in Medicine* 41.1 (Jan. 1999), pp. 179–186. ISSN: 0740-3194. DOI: 10.1002/(sici)1522-2594(199901)41:1<179::aid-mrm25>3.0.co;2-v.

[166] Pawel A. Penczek, Robert Renka, and Hermann Schomberg. "Gridding-based direct Fourier inversion of the three-dimensional ray transform". In: *Journal of the Optical Society of America. A, Optics, Image Science, and Vision* 21.4 (Apr. 2004), pp. 499–509. ISSN: 1084-7529. DOI: 10.1364/josaa.21.000499.

[167] P.J. Beatty, D.G. Nishimura, and J.M. Pauly. "Rapid gridding reconstruction with a minimal oversampling ratio". In: *IEEE Transactions on Medical Imaging* 24.6 (June 2005), pp. 799–808. ISSN: 1558-254X. DOI: 10.1109/TMI.2005.848376.

[168] Steven W. Smith. "Windowed-Sinc Filters". In: *Digital Signal Processing* (2003), pp. 285–296. DOI: 10.1016/b978-0-7506-7444-7/50053-4.

[169] R. B. Blackman and J. W. Tukey. "The Measurement of Power Spectra from the Point of View of Communications Engineering — Part I". In: *Bell System Technical Journal* 37.1 (1958), pp. 185–282. ISSN: 1538-7305. DOI: 10.1002/j.1538-7305.1958.tb03874.x.

[170] Elmar Zeitler. "Reconstruction with Orthogonal Functions". In: *Electron Tomography: Three-Dimensional Imaging with the Transmission Electron Microscope*. Ed. by Joachim Frank. Boston, MA: Springer US, 1992, pp. 63–89. ISBN: 978-1-4757-2163-8. DOI: 10.1007/978-1-4757-2163-8_4.

[171] David Gottlieb and Chi Wang Shu. "On the Gibbs phenomenon and its resolution". In: *SIAM Review* 39.4 (1997), pp. 644–668. ISSN: 00361445. DOI: 10.1137/S0036144596301390.

[172] Torben Andersen and Anita Enmark. "Fourier Transforms and Interpolation". In: *Integrated Modeling of Telescopes*. Ed. by Torben Andersen and Anita Enmark. Astrophysics and Space Science Library. New York, NY: Springer, 2011, pp. 45–76. ISBN: 978-1-4614-0149-0. DOI: 10.1007/978-1-4614-0149-0_4.

[173] F.J. Harris. "On the use of windows for harmonic analysis with the discrete Fourier transform". In: *Proceedings of the IEEE* 66.1 (Jan. 1978), pp. 51–83. ISSN: 1558-2256. DOI: 10.1109/PROC.1978.10837.

[174] E. Maeland. "On the comparison of interpolation methods". In: *IEEE transactions on medical imaging* 7.3 (1988), pp. 213–217. ISSN: 0278-0062. DOI: 10.1109/42.7784.

[175] URL: https://github.com/dtegunov/gtom/blob/master/src/Reconstruction/RecFourierAdd.cu.

[176] URL: https://github.com/3dem/relion/blob/master/src/acc/cuda/cuda_kernels/BP.cuh.

[177] W. Wan and J. a. G. Briggs. "Cryo-Electron Tomography and Subtomogram Averaging". In: *Methods in Enzymology* 579 (2016), pp. 329–367. ISSN: 1557-7988. DOI: 10.1016/bs.mie.2016.04.014.

[178] Xing Zhang and Z. Hong Zhou. "Limiting factors in atomic resolution cryo electron microscopy: no simple tricks". In: *Journal of Structural Biology* 175.3 (Sept. 2011), pp. 253–263. ISSN: 1095-8657. DOI: 10.1016/j.jsb.2011.05.004.

[179] Wim J. H. Hagen, William Wan, and John A. G. Briggs. "Implementation of a cryo-electron tomography tilt-scheme optimized for high resolution subtomogram averaging". In: *Journal of Structural Biology* 197.2 (Feb. 2017), pp. 191–198. ISSN: 1095-8657. DOI: 10.1016/j.jsb.2016.06.007.

[180] Ricardo M. Sanchez et al. "Subnanometer-resolution structure determination in situ by hybrid subtomogram averaging - single particle cryo-EM". In: *Nature Communications* 11.1 (July 24, 2020), p. 3709. ISSN: 2041-1723. DOI: 10.1038/s41467-020-17466-0.

[181] W. O. Saxton, W. Baumeister, and M. Hahn. "Three-dimensional reconstruction of imperfect two-dimensional crystals". In: *Ultramicroscopy* 13.1 (Jan. 1, 1984), pp. 57–70. ISSN: 0304-3991. DOI: 10.1016/0304-3991(84)90057-3.

[182] Joachim Frank and Bruce F. McEwen. "Alignment by Cross-Correlation". In: *Electron Tomography: Three-Dimensional Imaging with the Transmission Electron Microscope*. Ed. by Joachim Frank. Boston, MA: Springer US, 1992, pp. 205–213. ISBN: 978-1-4757-2163-8. DOI: 10.1007/978-1-4757-2163-8_9.

[183] Marin van Heel, Michael Schatz, and Elena Orlova. "Correlation functions revisited". In: *Ultramicroscopy* 46.1 (Oct. 1, 1992), pp. 307–316. ISSN: 0304-3991. DOI: 10.1016/0304-3991(92)90021-B.

[184] R. Guckenberger. "Determination of a common origin in the micrographs of tilt series in three-dimensional electron microscopy". In: *Ultramicroscopy* 9.1 (Jan. 1, 1982), pp. 167–173. ISSN: 0304-3991. DOI: 10.1016/0304-3991(82)90239-X.

[185] J. Frank et al. "Three-dimensional tomographic reconstruction in high voltage electron microscopy". In: *Journal of Electron Microscopy Technique* 6.2 (June 1987), pp. 193–205. ISSN: 0741-0581, 1553-0817. DOI: 10.1002/jemt.1060060210.

[186] David N. Mastronarde and Susannah R. Held. "Automated tilt series alignment and tomographic reconstruction in IMOD". In: *Journal of Structural Biology* 197.2 (Feb. 2017), pp. 102–113. ISSN: 1095-8657. DOI: 10.1016/j.jsb.2016.07.011.

[187] Joachim Dengler. "A multi-resolution approach to the 3D reconstruction from an electron microscope tilt series solving the alignment problem without gold particles". In: *Ultramicroscopy* 30.3 (July 1, 1989), pp. 337–348. ISSN: 0304-3991. DOI: 10.1016/0304-3991(89)90063-6.

[188] Kendra E. Leigh et al. "Chapter 11 - Subtomogram averaging from cryo-electron tomograms". In: *Methods in Cell Biology*. Ed. by Thomas Müller-Reichert and Gaia Pigino. Vol. 152. Three-Dimensional Electron Microscopy. Academic Press, Jan. 1, 2019, pp. 217–259. DOI: 10.1016/bs.mcb.2019.04.003.

[189] Jing Liu, Pamela C. Cosman, and Bhaskar D. Rao. "Robust Linear Regression via \ell_0 Regularization". In: *IEEE Transactions on Signal Processing* 66.3 (Feb. 2018), pp. 698–713. ISSN: 1941-0476. DOI: 10.1109/TSP.2017.2771720.

[190] Alexander Rigort et al. "Focused ion beam micromachining of eukaryotic cells for cryoelectron tomography". In: *Proceedings of the National Academy of Sciences of the United States of America* 109.12 (Mar. 20, 2012), pp. 4449–4454. ISSN: 1091-6490. DOI: 10.1073/pnas.1201333109.

[191] Renmin Han et al. "A marker-free automatic alignment method based on scale-invariant features". In: *Journal of Structural Biology* 186.1 (Apr. 1, 2014), pp. 167–180. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2014.02.011.

[192] Fernando Amat et al. "Markov random field based automatic image alignment for electron tomography". In: *Journal of Structural Biology* 161.3 (Mar. 2008), pp. 260–275. ISSN: 1095-8657. DOI: 10.1016/j.jsb.2007.07.007.

[193] Daniel Castaño-Díez et al. "Alignator: A GPU powered software package for robust fiducial-less alignment of cryo tilt-series". In: *Journal of Structural Biology* 170.1 (Apr. 1, 2010), pp. 117–126. ISSN: 1047-8477. DOI: 10.1016/j.jsb.2010.01.014.

[194] Cameron H. Owen and William J. Landis. "Alignment of electron tomographic series by correlation without the use of gold particles". In: *Ultramicroscopy* 63.1 (Apr. 1, 1996), pp. 27–38. ISSN: 0304-3991. DOI: 10.1016/0304-3991(95)00154-9.

[195] Hanspeter Winkler and Kenneth A. Taylor. "Accurate marker-free alignment with simultaneous geometry determination and reconstruction of tilt series in electron tomography". In: *Ultramicroscopy* 106.3 (Feb. 1, 2006), pp. 240–254. ISSN: 0304-3991. DOI: 10.1016/j.ultramic.2005.07.007.

[196] Shawn Zheng et al. "AreTomo: An integrated software package for automated marker-free, motion-corrected cryo-electron tomographic alignment and reconstruction". In: *Journal of Structural Biology: X* 6 (2022), p. 100068. ISSN: 2590-1524. DOI: 10.1016/j.yjsbx.2022.100068.

[197] Benjamin A. Himes and Peijun Zhang. "emClarity: software for high-resolution cryo-electron tomography and subtomogram averaging". In: *Nature Methods* 15.11 (Nov. 2018), pp. 955–961. ISSN: 1548-7105. DOI: 10.1038/s41592-018-0167-z.

[198] Dari Kimanius et al. "New tools for automated cryo-EM single-particle analysis in RELION-4.0". In: *Biochemical Journal* 478.24 (Dec. 16, 2021), pp. 4169–4185. ISSN: 0264-6021. DOI: 10.1042/BCJ20210708.

[199] Thomas Harvey Rowan. "Functional stability analysis of numerical algorithms". PhD thesis. Austin, TX: Department of Computer Science, University of Texas at Austin, 1990.

[200] J. A. Nelder and R. Mead. "A simplex method for function minimization". In: *The Computer Journal* 7 (1965), pp. 308–313. DOI: 10.1093/comjnl/7.4.308.

[201] Steven G. Johnson. *The NLopt nonlinear-optimization package*. https://github.com/stevengj/nlopt. 2007.

[202] Yu Liu et al. "A marker-free alignment method for electron tomography". In: *Ultramicroscopy* 58.3 (June 1, 1995), pp. 393–402. ISSN: 0304-3991. DOI: 10.1016/0304-3991(95)00006-M.

[203] Joseph A. Mindell and Nikolaus Grigorieff. "Accurate determination of local defocus and specimen tilt in electron microscopy". In: *Journal of Structural Biology* 142.3 (June 2003), pp. 334–347. ISSN: 1047-8477. DOI: 10.1016/s1047-8477(03)00069-8.

[204] Pawel A. Penczek et al. "CTER-rapid estimation of CTF parameters with error assessment". In: *Ultramicroscopy* 140 (May 2014), pp. 9–19. ISSN: 1879-2723. DOI: 10.1016/j.ultramic.2014.01.009.

[205] R. Grimm et al. "Determination of the inelastic mean free path in ice by examination of tilted vesicles and automated most probable loss imaging". In: *Ultramicroscopy* 63.3 (July 1996), pp. 169–179. ISSN: 0304-3991. DOI: 10.1016/0304-3991(96)00035-6.

[206] Cornelius Lanczos. *Applied analysis*. Ed. by Internet Archive. Englewood Cliffs, N.J., Prentice-Hall, 1956. 570 pp.

[207] Jérémie Bigot, Fabrice Gamboa, and Myriam Vimond. "Estimation of Translation, Rotation, and Scaling between Noisy Images Using the Fourier–Mellin Transform". In: *SIAM Journal on Imaging Sciences* 2.2 (Jan. 2009), pp. 614–645. DOI: 10.1137/070691231.

[208] Georgios Tzimiropoulos et al. "Robust FFT-Based Scale-Invariant Image Registration with Image Gradients". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.10 (Oct. 2010), pp. 1899–1906. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2010.107.

[209] Jignesh N Sarvaiya, Suprava Patnaik, and Salman Bombaywala. "Image registration using log-polar transform and phase correlation". In: *TENCON 2009 - 2009 IEEE Region 10 Conference*. Jan. 2009, pp. 1–5. DOI: 10.1109/TENCON.2009.5396234.

[210] Tristan Bepler et al. "Positive-unlabeled convolutional neural networks for particle picking in cryo-electron micrographs". In: *Nature Methods* 16.11 (Nov. 2019), pp. 1153–1160. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0575-8.

[211] Thorsten Wagner et al. "SPHIRE-crYOLO is a fast and accurate fully automated particle picker for cryo-EM". In: *Communications Biology* 2.1 (June 19, 2019), pp. 1–13. ISSN: 2399-3642. DOI: 10.1038/s42003-019-0437-z.

[212] Emmanuel Moebel et al. "Deep learning improves macromolecule identification in 3D cellular cryo-electron tomograms". In: *Nature Methods* 18.11 (Nov. 2021), pp. 1386–1394. ISSN: 1548-7105. DOI: 10.1038/s41592-021-01275-4.

[213] Yun-Tao Liu et al. "Isotropic reconstruction for electron tomography with deep learning". In: *Nature Communications* 13.1 (Oct. 29, 2022), p. 6482. ISSN: 2041-1723. DOI: 10.1038/s41467-022-33957-8.

[214] Muyuan Chen et al. "A complete data processing workflow for cryo-ET and subtomogram averaging". In: *Nature Methods* 16.11 (Nov. 2019), pp. 1161–1168. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0591-8.

Acronyms

1d 1-dimensional 33, 34, 56, 67, 69, 74, 84, 96, 100, 103, 104, 122–124, 127

2d 2-dimensional 22, 24–26, 40, 64, 89, 91, 97–100, 102, 107, 110, 113, 115–124, 126

3d 3-dimensional 1, 18, 22–28, 40, 64, 65, 89, 91, 92, 94, 96, 97, 99, 100, 102–104, 106–110, 114–117, 120, 121, 127

4d 4-dimensional 64, 66, 67

AD Automatic (or Algorithmic) Differentiation 36, 44–46

API Application Programming Interface 34, 35, 39, 45, 50, 51, 56, 66, 74–76

AST Abstract Syntax Tree 36, 38

BDHW Batch-Depth-Height-Width 64, 65, 80

BIM Beam Induced Motion 23

CC Cross-Correlation 115, 119, 120, 124, 131

CPU Core Processing Unit 41–43, 50, 51, 56, 63, 74

cryoEM Cryo Electron Microscopy 1, 10, 13, 14, 17–19, 21, 22, 24, 26, 29–32, 38–40, 43, 44, 46, 49, 80, 88, 89

cryoET Cryo Electron Tomography 49, 92, 113, 115

CTF Contrast Transfer Function 16, 17, 19, 23, 25, 122–125, 129–131

DFT Discrete Fourier Transform 83, 94, 104, 107, 129, 130

DSL Domain-Specific Language 35, 36, 38, 45, 46, 57

EWS Ewald Sphere 19, 21, 22, 26, 27, 97

FFT Fast Fourier Transform 83, 84

FOV Field Of View 18, 27, 28, 121, 122, 129, 130

FSC Fourier Shell Correlation 134

GPGPU General-Purpose Computing on GPU 41, 43

GPU Graphical Processing Unit 29, 34, 36, 39–43, 50, 51, 56, 59, 60, 63, 74, 82, 91, 99

HPC High-Performance Computing 58

IR Intermediary Representation 46

JIT Just-In-Time 59, 60, 63

MLIR Medium-Level Intermediary Representation 37, 38

PDB Protein Data Bank 12

PSF Point Spread Function 15, 16

RAM Random-Access Memory 33, 41

rFFT Real Fast Fourier Transform 83, 84

SIMD Single Instruction Multiple Data 42, 43, 56

SIMT Single Instruction Multiple Threads 43

SNR Signal to Noise Ratio 22, 23, 28, 114, 117, 130

SPA Single Particle Analysis 22–24, 27, 28, 91, 100, 117

SSNR Spectral Signal to Noise Ratio 23

STA Subtomogram Averaging 27, 117

TEM Transmission Electron Microscope 13