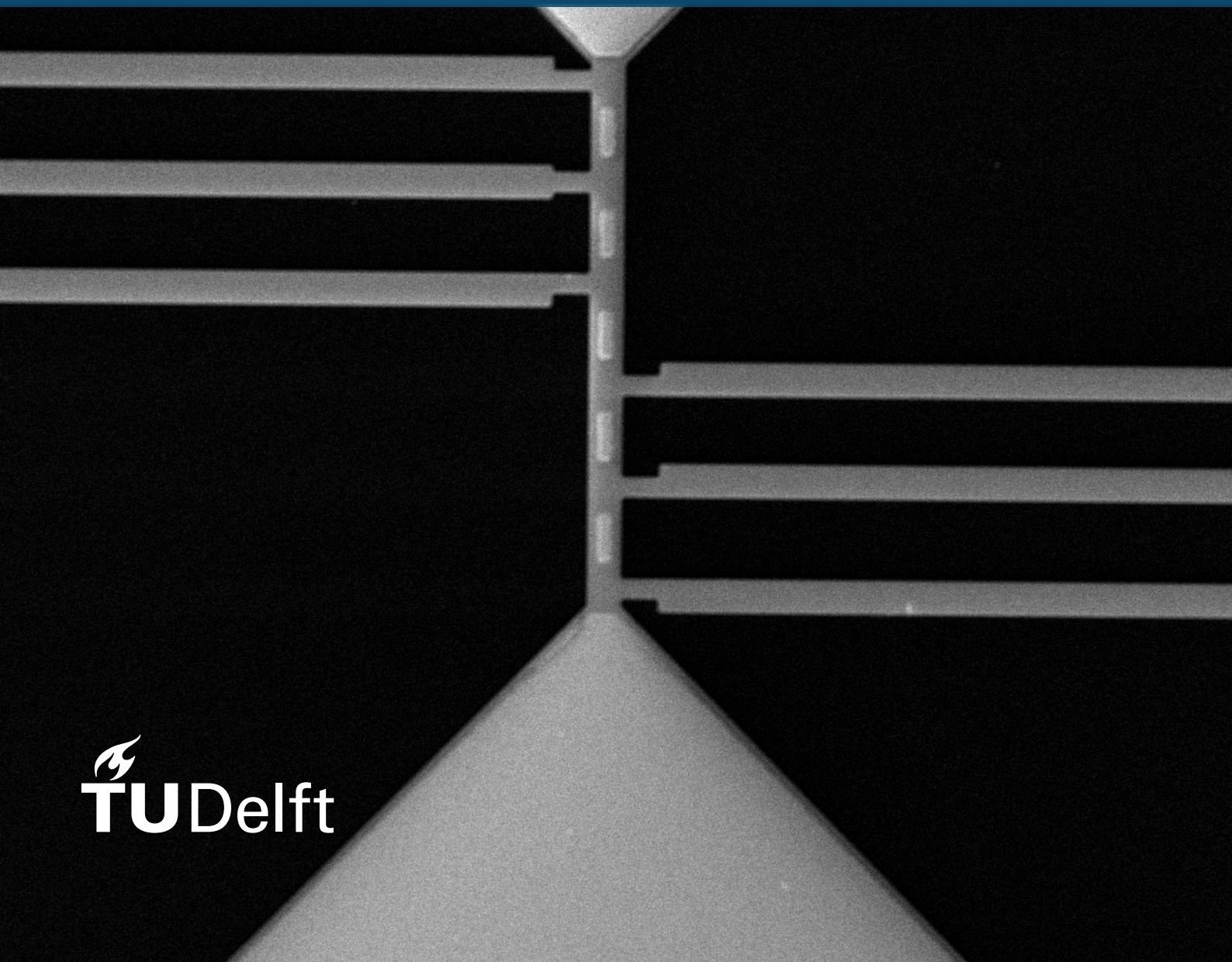
Analysis and Processing of SEM Images for Stabilised Video Generation

BSc Thesis

Thijs Heezen, Mats Deckers & Gijs de Bruijn



TU Delft

Analysis and Processing of SEM Images for Stabilised Video Generation

BSc Thesis

Thesis report

by

Thijs Heezen, Mats Deckers & Gijs de Bruijn

Thesis committee:	Dr. ir. S. Vollebregt
	Prof. dr. ir. L. Abelmann
	Dr. ing. M. Shahraki Moghaddam, PhD
Supervisors:	Dr. ir. S. Vollebregt
	Dr. Yaqian Zhang
Place:	Faculty of Electrical Engineering, Mathematics and Computer Science, Delft
Project Duration:	October 2025 – December 2025
Student numbers:	Thijs Heezen: 5328144
	Mats Deckers: 5645875
	Gijs de Bruijn: 5271142

Faculty of Electrical Engineering, Mathematics and Computer Science · Delft University of Technology

Abstract

This thesis addresses the challenge of generating long-duration, high-quality videos using the XL30 FEG/S-FEG/SIRION Scanning Electron Microscope (SEM), a task traditionally hindered by slow acquisition speeds, sample drift, and the absence of automated frame scheduling. As a result, researchers lack practical tools to observe nanoscale dynamics over extended experiments. The goal of the project for this subsystem was to develop an image-processing pipeline capable of stabilising SEM recordings and enabling automated video creation from sequential still images.

To achieve this, software was developed to process frames acquired at user-defined intervals, correct drift, and assemble images into a cohesive video. Drift was mitigated through a dual-layer strategy: small displacements were compensated using purely software based frame stabilisation techniques, while larger misalignments were addressed by calculating beam shift vectors to support mechanical correction in collaboration with the SEM Control subsystem. The drift between two images was calculated using a phase correlation based algorithm.

The resulting prototype successfully stabilised long image sequences and produced smooth, high-definition SEM videos, even in the presence of significant sample drift. These outcomes demonstrate the feasibility of automated SEM video generation and provide a modular framework that can be extended with improved robustness for challenging imaging conditions.

Preface

This thesis is the result of the Bachelor Graduation Project at the Faculty of Electrical Engineering, Mathematics and Computer Science at Delft University of Technology. The project was carried out in collaboration with the Else Kooi Laboratory and focuses on the development of an automated system for long-duration imaging using a scanning electron microscope.

The overall goal of the project was to enable the automated acquisition, stabilisation, and visualisation of SEM images over extended periods of time. Due to the limitations of the hardware and software of the XL30 SEM, the project resulted in the design and implementation of a prototype software that interfaces with the existing setup without requiring modifications to the microscope itself.

We would like to thank our supervisors Dr. Ir. S. Vollebregt, and Dr. Yaqian Zhang for their guidance and support throughout the project and for their feedback and constructive input during the design, implementation, and evaluation phases of the work. We are also grateful for the samples provided by Yaqian, which we could use for our testing purposes. Finally, we would like to thank our fellow project members Tijmen ten Berge, and Matthijs Spijker for the pleasant collaboration.

Thijs Heezen, Mats Deckers & Gijs de Bruijn
Delft, december 2025

Contents

List of Figures

List of Tables

1

Introduction

1.1. Scanning Electron Microscopes

SEMs, or Scanning Electron Microscopes, are extremely versatile tools that are used by researchers in all kinds of fields. From cell research in biology or mineral research in geology to material science and semiconductor research. Here, at the faculty of electrical engineering, mathematics and computer science, one such instrument is the FEI XL30 SFEG SEM. A picture of the setup in EWI is shown in figure 1.1. This SEM is located next to the ground-floor hallway of the laagbouw, near the entrance to the Else Kooi Laboratory

Figure 1.1: The scanning electron microscope at EWI

Unlike optical cameras, a SEM does not capture an image in a single instant. Instead, the microscope forms an image by scanning a focused electron beam across a sample horizontally, line by line. At every pixel position, the detector measures a signal intensity produced by electronsample interactions, and the full image is shown only after the entire scan has completed[1, pp. 422–424]. High-quality SEM images therefore require relatively long scanning times per line to ensure sufficient signal-to-noise ratio. If the scanning time is reduced in order to produce frames at real-time video rates, the detector collects far fewer electrons, resulting in images with substantially more noise, reduced contrast, and visible artefacts. For this reason, true high-resolution, low-noise SEM video cannot be produced at live-video frame rates, but instead, individual high-quality frames must be acquired at slower intervals and combined into a video afterwards.

Despite their widespread use, electron microscopes are very large, expensive, and complex machines. As a consequence, older models are still widely used. The XL30, for example, is almost 25 years old. While these older microscopes may lack some of the advanced detectors or more modern automation options, these older models generally still offer adequate image quality for a lot of the research that they are used for.

One of the automation features that is absent on the XL30 is the functionality to automatically capture images at fixed time intervals. At present, researchers using an XL-series SEM have limited options for detailed analysis of a sample over an extended period. They can either manually capture numerous images over the course of several hours, or take a single image at the beginning and another at the end of the experiment. Both approaches have drawbacks: the first is time-consuming and inaccurate, while the second sacrifices detail of analysis or any occurrences between the two captured frames. As a result, neither method makes effective use of the researchers valuable time. Our goal is to create a module that fully automates this workflow, from automatic image acquisition, to the production of a high-quality video. This will be done by providing the microscope with a set of instructions to generate multiple images at a predetermined time interval, after which the images are compiled into a single video file.

Because the sample will be recorded for several hours, and the experiment operates on the nanoscale, drift is unavoidable. Over long periods, the sample can shift by hundreds of nanometres due to thermal fluctuations, charge build-up, or even vibrations from people walking near the microscope. Since something as simple as nearby foot traffic can already cause noticeable drift, it is reasonable to expect that the introduction of trams

on the Mekelweg could contribute additional disturbance. To manage this, a two-tier drift-correction strategy is used. Small displacements are handled in software through frame-to-frame image stabilisation, while larger shifts are corrected by sending beam-shift commands to the SEM control module. Without this correction strategy, prolonged sample drift could cause the region of interest to completely shift out of frame which would make the entire process of long time imaging completely ineffective.

Another challenge is the legacy computing environment on which the XL30 operates. The microscope is controlled by a Windows 2000 PC, while a separate Windows 7 machine is available for other equipment such as the EDAX detector. These two systems communicate via a serial connection, making modern integration and automation far from straightforward.

These limitations drive the need for the development of a system capable of automating image acquisition, correcting sample drift, and compiling the resulting images into a coherent video. Designing such a system that enables long-duration SEM experiments without constant supervision is the focus of the Bachelor Graduation Project of which this thesis is a part. The details of how this project is divided into subsystems will be clarified in a subsequent section.

1.2. State of the Art Analysis

Image stabilisation of videos is a field where a lot of research has been done. However, many of the reports focus on 3D motion estimation for use in applications such as smartphones. In [2], a lot of stabilisation methods are described. For this thesis, the 2.5D and 3D methods add unnecessary computational resources, as the image only moves in the 2 directions orthogonal to the electron beam. The process of stabilisation can further be split into three steps:

Motion estimation → Motion compensation → Image warp

Motion estimation is the process of finding a vector that describes the amount that an image has moved compared to a reference image. The five most commonly used methods are block matching, optical flow, bitplane matching, phase correlation, and feature point matching. These methods all have different computational requirements and varying levels of noise susceptibility.

The motion compensation step acts on the data from the motion estimation step and tries to find a smooth way to shift the images. This step is especially important in the context of handheld video recordings in combination with the 2D stabilisation methods. Because of rotation or third-dimensional movement, some parts of the image may move faster than other parts of the image. The motion compensation step uses parametric filtering or trajectory smoothing to make sure that the final video frames are compensated smoothly.

Finally, the image warp step uses a 2D-transformation, mosaic warp, motion restoration or cropping transformation to change the image according to the output of the motion compensation step. These methods can result in the introduction of unknown pixels. These are also filled in by the image warp step. The way they are filled in differs per option, but these options can be categorised into: not filling in (leaving the pixel blank), estimation based on the current frame or estimation based on the surrounding frames.

The authors of [3] demonstrate an algorithm to find the rotation and scale change between two pictures using the correlation operation. During an experiment, no changes in scale are expected, and the rotational component of the drift is likely far less than the translational drift. This means that the full algorithm presented here may not apply to this thesis, but the demonstration of performing a correlation on 2-dimensional datasets and using that to estimate change between two images is a valuable resource.

Within the field of video stabilisation, deep-learning-based methods are also widely researched. Methods like those summarised in [4] can offer better support for low-quality or blurry images. They can also be trained to simultaneously perform other tasks on the video, such as deblurring, de-noising or super sampling to increase the output resolution. However, these methods can have the drawbacks of sometimes introducing artefacts or requiring powerful hardware to run these deep-learning models. Another implication of using a deep-learning-based stabilisation algorithm is the need for a good dataset. Such a dataset needs to be big enough to train the model well, but also varied enough to make sure that the model can handle a lot of different research subjects. Because of the need for powerful hardware, these algorithms are unlikely to run fast enough and perform well on the computers present in the room of the SEM.

1.3. Subdivision of the System

In this project, the work is divided into two cooperating subsystems: the SEM control subsystem and the image processing subsystem. Together, they create an automated workflow for long-duration SEM imaging, as shown in figure 1.2.

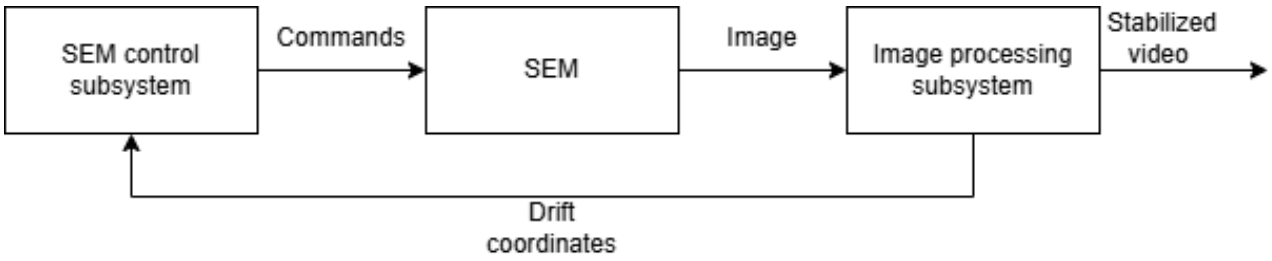


Figure 1.2: System Overview

The SEM control subsystem communicates with the XL30 microscope through its serial interface. It sends all commands required for automated image acquisition, such as capturing frames at fixed intervals, and applies beam shift corrections when drift is detected.

The image processing subsystem receives each image produced by the SEM and performs drift estimation, confidence assessment, and frame stabilisation. It then compiles the stabilised frames into a high-quality video. When a shift is detected, the subsystem returns drift coordinates to the SEM control subsystem, forming a feedback loop that keeps the region of interest centred during long experiments.

This division allows both subgroups to work independently on their respective tasks while maintaining a unified system capable of automated, long-duration SEM imaging.

1.4. Document Outline

This thesis is structured as follows. Chapter 2 presents the programme of requirements. It outlines the assumptions, functional and non-functional requirements, and trade-offs that guided the design of both the full system and the image processing subsystem.

Chapter 3 reviews existing techniques relevant to the image processing tasks. Methods for image shift estimation, video generation, and confidence assessment are evaluated, leading to the selection of phase correlation, OpenCV-based video assembly, and the peak-to-sidelobe ratio as the most suitable approaches.

Chapter 4 describes the design and implementation of the image processing software. It covers software constraints, system architecture, communication mechanisms, drift estimation, confidence handling, stabilisation logic, and video generation.

Chapter 5 evaluates the performance and robustness of the implemented system in relation to the system requirements. Experimental results are presented to assess drift correction accuracy, confidence metric behaviour, and operational limits under degraded imaging conditions.

Finally, Chapter 6 concludes the thesis with a summary of the results and discusses limitations and recommendations for future work.

2

Programme of requirements

To reach our goal of building a module that can automate the capture of images while ensuring that the object of interest remains in the frame, it is important to know the assumptions and the requirements that were set.

The requirements can be divided into full system requirements, which apply to this thesis and the thesis of the SEM control subgroup, and the image processing requirements, which are the requirements that are relevant only to the product of this thesis.

2.1. Assumptions

To compose the requirements applicable to the system, a few assumptions were kept in mind:

[1.1] The SEM allows a maximum beam shift range of 20 μm in either direction along both the x and y axes[5].

[1.2] The microscope is controlled by a Windows 2000 PC, while a separate Windows 7 machine is available through serial connection.

[1.3] The resolution of the beam is 1.5 nm at 10 kV or higher and 2.5 nm at 1 kV.[5]

[1.4] The image stabilisation system assumes that all input images are provided in `.TIF` (TIFF) format. No other image formats are required to be supported by the system and TIFF metadata will always be available [6].

2.2. Full System Requirements

The requirements that are applicable to both subsystems are divided into the functional requirements, non-functional requirements and trade-off requirements.

2.2.1. Functional Requirements

[2.1] The order of the frames of the output video must be equal to the order in which the frames are captured on the SEM

[2.2] Both subsystems (SEM control & Image Processing) must be able to run separately as well as combined.

[2.3] The control software GUI must store and report the following settings to the image processing software: frame interval (seconds), output video FPS, output location, max shift percentage, outpainting method, video file name, and video file codec/extension.

2.2.2. Non-functional Requirements

[3.1] The final product must not result in the need to reconfigure the cables at the back of the setup.

[3.2] The product must be designed using one or more pieces of software.

[3.3] The software must be developed to work on Windows 7.

[3.4] The implementation must not require additional software to be installed on the Windows 2000 PC.

[3.5] The region of interest must not shift more than 2.5% in the x-direction during playback of the output video.

[3.6] The region of interest must not shift more than 2.5% in the y-direction during playback of the output video.

[3.7] The software won't support real-time video recording from the SEM.

[3.8] When the sample has moved completely out of frame in one step, the software will not do any searching to catch such extreme movements.

2.2.3. Trade Off Requirements

[4.1] The implementation should not result in extra hardware to be added to the SEM setup.

[4.2] The user should be able to specify a desired output FPS.

[4.3] The user specified beam shift threshold should be no larger than the system's capability limits set in requirements [B.4] and [B.5].

[4.4] The processing time of the image processing software should not influence the timing between the capture of two images.

[4.5] The image processing software should prioritise sending a beam shift request to the control software to limit the time control has to wait for a potential shift.

[4.6] The software should be able to perform drift correction both physically (up to the beam shift limits in requirement [1.1]) and digitally (up to the user-specified threshold).

2.3. Image Processing Requirements

The requirements that apply to the subsystem that this thesis is about are also divided into functional requirements, non-functional requirements and trade-off requirements.

2.3.1. Functional Requirements

[A.1] The image processing software must be able to receive SEM images incrementally in the correct format (.TIFF extension, 8-bit colour).

[A.2] The image processing software must be able to receive SEM images at arbitrary time intervals.

[A.3] The image processing software must be able to calculate the drift in pixels between the current image and a reference image.

[A.4] The image processing software must be able to convert a calculated pixel shift into a corresponding physical shift in millimetres.

[A.5] The image processing software must spatially align the processed images based on the calculated drift, such that the resulting accumulative alignment error does not exceed the limits specified in requirements [3.5] and [3.6].

[A.6] The image processing software must be able to notify the control software when the object has shifted more than the set threshold.

[A.7] The image processing software must be able to receive a command which signals that the experiment is done and the final video should be created.

2.3.2. Non-Functional Requirements

[B.1] The output video of the image processing software must have at least the same resolution as the input images(712x484 pixels for Standard Definition, or 1424x968 pixels for High Definition).

[B.2] The image processing software must be able to export the video in a common compressed format and in a lossless format (to ensure an option for data preservation).

[B.3] The image processing software must keep an experiment running, even if erroneous data is processed.

[B.4] The software must be able to correct horizontal physical shift up to a distance equivalent to one-third of the horizontal pixel resolution.

[B.5] The software must be able to correct vertical physical shift up to a distance equivalent to one-third of the vertical pixel resolution.

[B.6] The computed shift must have an error no larger than an absolute distance of 10 pixels for SD(Standard Definition) images[1]

[B.7] The computed shift must have an error no larger than an absolute distance of 20 pixels for HD(High Definition) images[2]

[B.8] The calculated size of a pixel can deviate at most 2.5% from the pixel size according to the existing SEM software.

[1]Note that requirements [B.6] and [B.7] are very similar to requirements [3.5] and [3.6]. Requirements [B.6] and [B.7] are slightly tighter definitions of requirements [3.5] and [3.6]. Where [3.5] and [3.6] relate to the region of interest shift in the output video, and [B.6] and [B.7] relate to the error in the computed shift of the actual sample. The reason these new requirements were added is to make it easier to determine if these requirements are fulfilled.

[2]See footnote 1

[B.9] The image stabilisation software must ensure stabilisation across the full sequence, ensuring no cumulative drift errors.

[B.10] The image processing software will not perform deep learning based algorithms that require more hardware performance than the PCs at the SEM have.

2.3.3. Trade Off Requirements

[C.1] The image processing software should handle the edges of the images in a way that is clear to the user.

[C.2] The image processing software should be able to handle the edges of the images in a visually pleasing way.

[C.3] The image processing software should be able to handle intensity fluctuations within the image without increasing the error to more than 125% of the specified error in requirements [B.6] and [B.7]

3

Existing Techniques

In the previous chapters, the context and requirements of the SEM image processing subsystem were introduced. The challenges associated with long-duration SEM imaging, such as sample drift, inconsistent frame timing, intensity fluctuations, and the need for reliable automation, place specific constraints on the image processing subsystem. Based on these requirements, the choice of suitable algorithms has to be guided not only by accuracy, but also by robustness, efficiency, and compatibility with the legacy hardware and software present in the SEM laboratory.

This chapter examines a selection of existing techniques that address three core tasks of the subsystem: calculating image shifts between successive frames, verifying the reliability of these calculated shifts, and converting stabilised frames into a video. For image shift calculation, a variety of methods exist, such as block matching, optical flow, feature matching, and correlation-based approaches, each with different computational demands and sensitivity to noise. Given the performance and robustness requirements of this project, these options must be evaluated critically.

Similarly, the task of assembling processed frames into a coherent video can be implemented through custom algorithms or by using established libraries. Since the system must run on the limited hardware and software available in the SEM room, and must integrate seamlessly with the control subsystem, the trade-off between implementing a custom-made solution and using well-tested tools has to be evaluated.

Finally, any practical stabilisation system must include a mechanism to quantify confidence in its calculated shifts. Without this, bogus drift estimates caused by noise, low contrast, or sudden changes in imaging conditions could cause errors throughout the video. A range of confidence estimation methods is available, and identifying one that is reliable for our purposes is crucial for upholding system stability during long experiments.

The remainder of this chapter evaluates these techniques in detail and highlights the motivation behind selecting phase correlation, `OpenCV`-based [7] video assembly, and peak-to-sidelobe ratio confidence assessment as the most suitable methods for this project.

3.1. Image Shift Calculation

Accurate estimation of the translational shift between consecutive SEM images is essential for stabilising long-duration recordings. Many well-established techniques exist for calculating motion or displacement between frames. However, not all approaches are equally suitable for SEM data or for the constraints of the XL30 setup.

A common class of methods is block matching, in which an image is divided into a grid of blocks and each block is compared against a search region in the following frame [8]. Although simple, block matching requires scanning many candidate positions and therefore becomes computationally expensive for high-resolution images or large drift ranges. Furthermore, SEM images often contain large uniform regions or repeating patterns, making block matching susceptible to uncertain matches.

Another commonly used group of methods is optical flow, which estimates motion by looking at how pixel intensities change between frames. Classical optical-flow algorithms such as Lucas-Kanade and Horn-Schunck assume that brightness stays mostly constant and that motion changes smoothly across the image. These assumptions often hold for natural video, so the methods work well when motions are small and continuous[9]. In SEM images, however, brightness can vary, the signal-to-noise ratio is often low, and contrast can behave in nonlinear ways. Because of this, classical optical-flow methods usually perform poorly and give less reliable motion estimates for the SEM data relevant to this thesis.

A third type of method uses feature point detection and matching, where algorithms find keypoints in each image and try to match them across frames. These approaches work well for natural scenes but tend to perform poorly on some SEM images. That is because SEM imagery may lack clearly defined corners or junctions, which

are the kinds of structures most detectors rely on. As a result, too few reliable keypoints are found, and even those that are detected can be unstable or mismatched due to variations in contrast or noise[10].

Finally, correlation-based approaches, like phase correlation, operate in either spatial or frequency domains to determine the displacement that maximises similarity between frames. Phase correlation exploits the Fourier shift theorem to estimate translation as a distinct peak in the cross-power spectrum, first proposed in [11]. This method is robust against global illumination changes, works well on noisy or low-contrast data, and is computationally efficient due to the use of FFTs[12, 13, 14]. These properties make it a strong candidate for the drift characteristics observed in SEM imaging.

Because drift between SEM frames is largely translational, and because of the hardware constraints, and the fact that drift calculation needs to happen between the capturing of frames, a lightweight and reliable method is required; phase correlation best satisfies the systems requirements. It provides subpixel accuracy, low sensitivity to intensity changes, and stable performance on typical SEM images. Also, early tests with XL30 SEM images showed remarkable accuracy in calculating how much shift occurred between images. For these reasons, phase correlation was selected as the core method for image shift estimation in this project. More specifics on this rationale can be found in section 4.11.

3.1.1. Specifics of Phase Correlation

Phase correlation is a frequency-domain technique based on the Fourier Shift Theorem, which states that a spatial translation in an image corresponds to a linear phase shift in the Fourier domain[11, 12, 13, 14]. Considering two images treated as signals $f(x, y)$ and $g(x, y)$, where

$$g(x, y) = f(x - \Delta x,\ y - \Delta y) \tag{3.1}$$

their Fourier transforms $F(u, v)$ and $G(u, v)$ satisfy:

$$G(u, v) = F(u, v)\, e^{-j2\pi\left(\frac{u\Delta x}{M} + \frac{v\Delta y}{N}\right)} \tag{3.2}$$

where $(\Delta x, \Delta y)$ is the translation between the two images and M, N are the horizontal and vertical image dimensions, respectively.

Cross-Power Spectrum

Phase correlation computes the normalised cross-power spectrum:

$$R(u, v) = \frac{F(u, v)\, G^*(u, v)}{|F(u, v)\, G^*(u, v)|} \tag{3.3}$$

where $G^*(u, v)$ is the complex conjugate of $G(u, v)$. Normalising by the magnitude removes amplitude information and preserves only the phase difference, which encodes the translation. The benefit of the normalisation step is to make the relation between the two images independent of their brightness.

Taking the inverse Fourier transform of $R(u, v)$ yields:

$$r(x, y) = \mathcal{F}^{-1}\{R(u, v)\} \tag{3.4}$$

For purely translational motion, $r(x, y)$ ideally becomes a Dirac delta function centred at the translation:

$$r(x, y) = \delta(x - \Delta x,\ y - \Delta y) \tag{3.5}$$

Thus, the location of the peak in $r(x, y)$ directly gives the pixel shift.

In practice, however, these operations will not return a perfect Dirac delta function like in equation 3.5. For a practical implementation, however, it still suffices to find the highest peak of the spectrum of $r(x, y)$, as the shape of that function still somewhat resembles a Dirac delta function.

3.2. Converting Frames Into Video

Once the individual SEM frames have been stabilised and spatially aligned, they must be combined into a single continuous video file. Although generating such a video may appear straightforward, several requirements of this project constrain the set of viable approaches. The video-generation stage must meet the criteria outlined in the image-processing requirements (see Requirements [B.1],[B.2], [B.3]). These constraints make it impractical to design a custom video encoder, as doing so would require implementing complex standards such as MPEG-4, H.264, or various lossless formats.

Instead, the `OpenCV` library [7] was selected as the foundation for this subsystem. `OpenCV` provides a well-tested implementation of the `VideoWriter` interface, which supports a wide range of codecs and container formats. Its video-writing functionality is built on top of the widely used FFmpeg backend, enabling compatibility with standard multimedia players and analysis tools without requiring the user to install additional software on the Windows 7 system. This aligns with the project requirements that limit modifications to the SEMs Windows 2000 machine and limit the installation of new software on the SEM's computers (see Requirements [3.4], [3.3]).

The workflow for generating a video using `OpenCV` is straightforward. After stabilisation, each processed frame is converted to a three-channel image format accepted by the `VideoWriter`. A `VideoWriter` instance is then initialised with an appropriate codec, such as `MP4V` for compressed output or `FFV1` for lossless output, along with a user-specified target frame rate and an output resolution equal to that of the stabilised frames. All stabilised frames are then sequentially appended to the video stream. Once the final frame has been written, the `VideoWriter` is released, finalising the output file.

Besides its simplicity, `OpenCV` offers several practical advantages. It guarantees consistent frame ordering, enforces resolution matching across all frames, and passes off encoding operations to optimised backends. This reliability is essential for multi-hour SEM experiments, where the system must construct a valid video even when occasional frames are imperfect or (although very unlikely) partially corrupted.

For the reasons of robustness, codec flexibility, performance, and ease of integration, `OpenCV` provides an ideal solution for assembling stabilised SEM frames into high-quality video files. Its use allows the project to focus on the core challenges of image stabilisation and drift correction rather than on implementing a custom multimedia pipeline.

3.3. Confidence

Even when phase correlation produces a well-defined peak, the resulting estimate is not always equally reliable. Variations in noise, low-contrast structures, or charging artefacts may produce noisy correlation surfaces. For this reason, a confidence metric is required to quantify how trustworthy a detected peak is.

Classic formulations, including the peak-to-sidelobe ratio (PSR), peak-to-second-peak ratio (P2SR), peak-to-average ratio (PAR), and peak Sharpness metrics were considered in choosing which metric fits best for this subsystem.

Phase correlation yields a correlation surface whose global maximum indicates the estimated shift. A good confidence metric should therefore increase when the main peak is dominant and decrease when competing peaks suggest a poorly defined translation. Below, the four evaluated metrics are summarised and compared.

Peak-to-Sidelobe Ratio (PSR)

The PSR measures how strongly the primary peak stands out compared to the surrounding sidelobe region. It is defined as

$$\mathrm{PSR} = \frac{C(x_p, y_p) - \mu_{\mathrm{sl}}}{\sigma_{\mathrm{sl}}} \tag{3.6}$$

where $C(x_p, y_p)$ is the peak value, and $\mu_{\mathrm{sl}}, \sigma_{\mathrm{sl}}$ are the mean and standard deviation of all correlation values outside a small exclusion window around the peak. This corresponds directly to the "peak-to-sidelobe" sharpness measure discussed in [15, pp. 2998–2999], where a higher PSR indicates a sharper correlation peak.

The motivation is that a reliable translational shift produces a narrow and isolated peak, whereas noise produces low-contrast, diffuse peaks that reduce σ_{sl} separation. In our implementation, an exclusion radius of five is used. This reduces the effect of the ramping around the value on the correlation surface. The PSR metric also exhibits good stability under SEM noise because it evaluates global sidelobe statistics rather than relying on a single secondary maximum.

Peak-to-Average Ratio (PAR)

The PAR metric

$$\mathrm{PAR} = \frac{C(x_p, y_p)}{|\overline{C}|} \tag{3.7}$$

compares the peak to the average absolute correlation magnitude. A functionally similar peak-to-average formulation appears in [16, p. 5]. While simple and computationally inexpensive, PAR does not distinguish between structured sidelobes and noise. This could make PAR less robust for SEM imagery, where the mean correlation level may be influenced by noise.

Peak-to-Second-Peak Ratio (P2SR)
The P2SR compares the dominant peak to the largest remaining peak in the correlation surface:

$$\text{P2SR} = \frac{C(x_{p1}, y_{p1})}{C(x_{p2}, y_{p2})} \tag{3.8}$$

Where $C(x_{p2}, y_{p2})$ is the second-largest peak on the correlation surface that does not lie within a small exclusion window. This metric is closely related to the "primary-to-secondary peak ratio" described in [15].

While P2SR captures whether the main peak is truly unique, it suffers from high sensitivity to noise. A single random noise spike can artificially reduce confidence. Since SEM images can contain textured regions with natural high-frequency content, false secondary peaks are able to occur, making this metric less reliable for this subsystems use case. Also, when the image stabilisation software is running stand-alone on pre-existing data that includes the built-in databar enabled, the existence of the databar in the images will always result in a peak on the correlation surface at the point $(0, 0)$.

Peak Sharpness Metric
Peak sharpness evaluates the energy concentrated in a local window W around the peak relative to the total correlation energy:

$$\text{Sharpness} = \frac{\sum_{(x,y)\in W} C(x,y)^2}{\sum_{x,y} C(x,y)^2}. \tag{3.9}$$

This metric is mathematically similar to the PCE (Peak-to-Correlation-Energy) measure discussed in [15]. Because SEM correlation peaks can be irregular due to noise or charging effects, basing confidence solely on the sharpness of a peak can be inconsistent.

Selection of PSR for This Project
Among the evaluated metrics, PSR proved the most stable for this project, as further discussed in appendix B.1. Its reliance on global sidelobe statistics makes it robust to secondary peaks and high-frequency texture in SEM data. Literature on correlation-filter performance also highlights closely related PSR definition as a strong indicator of peak reliability under noise and distortion conditions common in imaging scenarios[15][14].

For these reasons, PSR is adopted as the primary confidence measure in our drift-estimation pipeline and is applied consistently across all acquired frames to determine whether a detected shift is sufficiently reliable.

3.4. Summary

In this chapter, several existing techniques relevant to the SEM image-processing subsystem were evaluated. For the task of image shift calculation, multiple classes of algorithms were considered, including block matching, optical flow, feature-based matching, and correlation-based methods. Due to the noise characteristics, contrast variations, and largely translational drift present in SEM imagery, phase correlation was chosen as the most suitable approach. Its robustness to illumination changes, computational efficiency, and proven accuracy on SEM data make it a strong fit for the limitations of the XL30 setup.

Next, the problem of converting stabilised frames into a video was examined. Rather than implementing a custom encoder, the `OpenCV` library was selected for its reliable `VideoWriter` interface, codec flexibility, FFmpeg backend, and compatibility with the legacy hardware present in the SEM laboratory. This choice allows the system to reliably generate high-quality video files while keeping the implementation simple.

Finally, several confidence estimation metrics were compared to determine how reliably phase correlation peaks indicate image shifts. From the evaluated methods of PSR, PAR, P2SR, and peak sharpness, the Peak-to-Sidelobe Ratio proved the most suitable for SEM imagery. Its reliance on global sidelobe statistics makes it robust to secondary peaks, repetitive textures, and variations in contrast.

Overall, this chapter established phase correlation, `OpenCV`-based video assembly, and PSR confidence estimation as the techniques best suited to the challenges and requirements of long-duration SEM imaging. These methods form the foundation for the implementation described in the following chapters.

4

Design and Implementation

In this chapter, the design and implementation of the image processing software will be described. At first, a broad overview of the whole system will be described. Then the design considerations and choices of the subsystems will be highlighted.

4.1. Software Restrictions

The first important design choice to be made is the choice of what programming language to use. Regarding this choice, the requirements that should be considered are requirements [1.2], [3.1], [3.2], [3.3], [3.4] and [A.1]. Apart from these requirements, it is also important to consider what functions will be used a lot, and which language might have nice structures, features or packages available to ease the development. Because this software will need to work a lot with both signal processing, image processing and potentially image editing, a logical choice is to use Python for numerical signal and image processing tasks. This allows us to use packages such as NumPy, PIL and OpenCV [7, 17]. Which integrate smoothly with modern microscope-control ecosystems [18]. However, since the requirements require the software to work on Windows 7, the most recent version of Python (version 3.14) is not available. The most recent Windows 7 compatible is the Python 3.8 series [19]. So this project will be designed and tested on Python version 3.8.

4.2. Architecture Overview

The first step in designing the image stabilisation software is to come up with an architecture that allows for easy testing and easy implementation of new features. Another important consideration is that of requirements [2.1] and [2.2]. Ideally, a good architecture allows to keep most of the main logic the same, and requires only a small amount of code that is only used when running in one of these modes. Considering this and also requirement [A.2], the use of a message-based, or interrupt-based, approach was decided.

Such a message-based approach offers the benefit that it enables the ability to design a base communication class that runs the communication of the software. That base class can then be extended to a class that enables running combined with the control software and a class that enables running separately. The design of these classes is further explained in section 4.3.

These communication classes are used to shuttle messages between the control software and the image stabilisation software, or provide a prebuilt queue of messages based on the dataset in a given folder. Depending on the type of message received (see subsection 4.3.1), different parts of the software are executed.

The design of the software architecture can be seen in figure 4.1. In the subsequent sections, the modules in this figure will be explained further

4.3. Communication

To be able to change the program between its combined running mode to stand-alone running mode, the communication method will be swapped. To facilitate the programming of these classes, one abstract base class is created. This class contains empty `connect`, `send`, `receive`, `message_available`, `check_connected` and `close` functions. This class is then inherited by two classes that implement these functions. One class implements them for stand-alone operation, and another class implements them for running combined with the control software. In general, the `connect` and `close` functions register and deregister a certain name in the class. To then check if a name is present (connected), the `check_connected` function can be used. These three functions form a framework for both the control software and the image stabilisation software to check for each other's presence before starting the communication. For these checks, the control software (or its emulator, see subsection 4.3.3) is identified as `CONTROL` and the image stabilisation software is identified as `ANALYSIS`. The `message_available` function checks if a message is available for a certain receiver. The `send` function appends

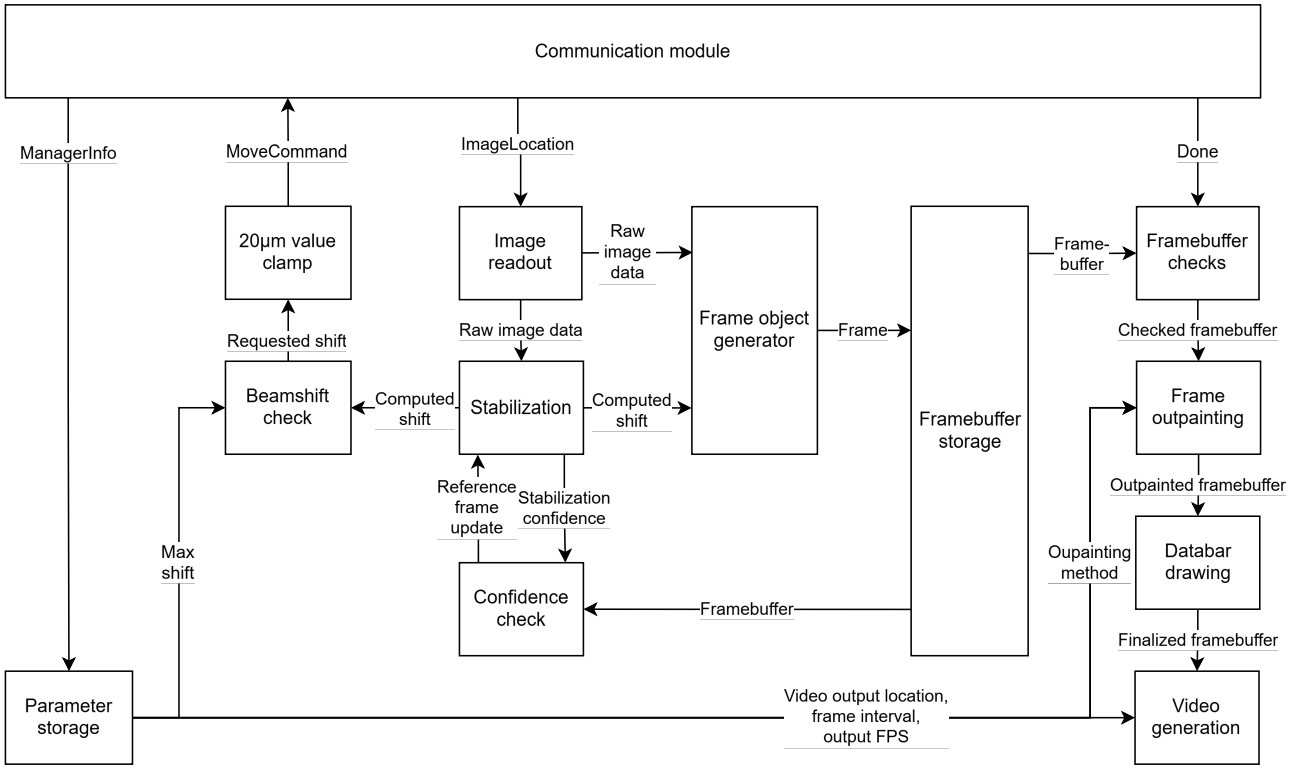


Figure 4.1: The architecture of the image stabilisation software

a message to the message queue. And finally, the `receive` function reads the oldest message from the queue and returns that message. This message is then immediately removed from the message queue.

Although both classes implement these functions in the same way, some subtle differences exist between the way that they work. The main difference in how they work is the way the handle certain messages. Therefore, it is first relevant to go over the messages that can be sent.

4.3.1. Messages

To facilitate the communication, a `Message` class was created. A message object contains three things: the type of message, the corresponding data and the receiver for that message. This receiver can either be `CONTROL` if the message is meant for the control software, or `ANALYSIS` if the message is meant for the image stabilisation software.

There are 6 different message types. These are:

- `ManagerInfo`: This message contains a list of strings. Each string contains the setting for one of the parameters that the image stabilisation software relies on. An overview of the parameters and their purposes is placed 4.1
- `ControlMoved`: This message contains a boolean to confirm that the control software has performed a beam shift.
- `ImageLocation`: This message contains a string with a file path to an image that was taken on the SEM. After receiving this message, the image in the given file path will be processed.
- `MoveCommand`: This message contains a tuple with two floating point numbers. These two numbers are the x and y coordinates (in millimetres) that the beam should be shifted to compensate for the drift. The code that sends this message should be responsible for making sure that these values do not exceed the physical limits described in [1.1].
- `Cancel`: This message contains a boolean to ask the control software to shut down the capturing process.
- `Done`: This message contains a boolean to signal that the capture is done. When this message is received, the generation of the final output video can be started.

One may notice that the software architecture shown in figure 4.1 does not depict any blocks or arrows that send or receive the `ControlMoved` and the `Cancel` commands. This is not an error. `ControlMoved` was implemented for the case that it was useful to know when a bean shift had occurred. However, during

Table 4.1: Overview of configuration parameters used by the image stabilisation manager.

Parameter	Type	Purpose
`output_location`	String	Path to the folder where intermediate results and the final stabilised video are written.
`interval`	Float / Integer	Time between successive image acquisitions, expressed in seconds. Used to control the temporal sampling of the time-lapse.
`fps`	Float / Integer	Target frame rate (frames per second) of the final output video. This controls how fast the time-lapse is played back.
`max_shift_percentage`	Float / Integer	Maximum allowed software shift before a beam shift is triggered, expressed as a percentage of the image size. If the computed drift compensation exceeds this value, a `MoveCommand` message is sent to the control software in order to recentre the beam.
`Out-painting`	String	With this field, the user can specify which outpainting strategy they want to use.
`File name`	String	Base name used for the final output video, without file extension.
`Extention`	String	File extension of the final output video (The software currently supports MP4 and MKV extensions). This determines the container/codec used for exporting the stabilised sequence as a video. `Extention` is the correct name of the parameter, not Extension.

development, no edge case came up where knowing this was needed. `Cancel` was implemented for when the image stabilisation software could no longer recover the shift between two images. However, with the goal of requirement [B.3] in mind and no situation coming up during testing where this looked like desired behaviour, no conditions are implemented that can send a `Cancel` command. Nevertheless, these functions are implemented in the control software. When the control software sends a `ControlMoved` message, this message is simply discarded.

4.3.2. Combined Implementation

On initialisation of the combined communication class, the initialising code (in the case of this BAP project: the control software) is required to already provide a first `ManagerInfo` message. This is done to ensure that a message with some parameters needed by the image stabilisation software is already prepared. This way, once the initialised communication class is passed to the image stabilisation software, the chance of errors due to missing data is reduced.

4.3.3. Stand-Alone Implementation

When the code is running with the stand-alone communication class, the class expects to be initialised with a string containing the folder location where the experiment data is stored. On initialisation, a fake connection with the (in this mode non-existing) control software is also initialised.

Then, when the connect function is called, the contents of the previously provided folder location are read. First, the code tries to search for a `.txt` file to read the `ManagerInfo` message from. Afterwards, all `.TIF` files in the folder (as stated in Assumption [1.4] the software only processes `.TIF` files) are put in `ImageLocation` messages. Also, if the filename of the `.TIF` file ends in `_M`, this signals that a beam shift was performed right before this image was captured. This causes a `ControlMoved` message to be placed in the message queue first. Also, since there is no communication to close, the `close` function does not do anything.

4.4. Frames

During regular operation of the image stabilisation software (in both modes), the most commonly sent message is an `ImageLocation`. Once an image is received, it should be stored and represented in some way in the image stabilisation software. To do this, the `Frame` class was designed. Once a file location of any `.TIF` file is given to

this class, it will open the image (if needed), convert it to grayscale, and store it as a NumPy array.

Within this class, some other helper functions also exist. Such as functions that automatically extract the metadata that is embedded in the `.TIF` file as seen in 4.6.1, and functions to get the size of the image. With the help of these functions, and some other parameters, the original size of the image, the current size of the image, and some of the relevant stabilisation data (both in millimetres and in pixels) are all stored as properties of any `Frame` object.

4.5. Stabilisation Module Design and Control Logic

Building on the phase-correlation framework in section 3.1 and the classical literature on phase correlation [20, 13, 21, 22, 14], the stabiliser module implements a specific API and control logic for SEM drift correction.

4.5.1. Implementation Summary

The implementation follows standard formulations of phase correlation and phase-only matched filtering [11, 21, 22, 13, 14] and uses conventional windowing and Fourier-processing practices [23, 24, 25, 26, 20].

1. Convert the image to grayscale (if needed) and centre-crop R, I (the reference image and the image to be stabilised) to (H, W) (the minimum height and width of both the images). In the current system, all images share the same resolution; the crop acts as a fail-safe.
2. Subtract the mean from the images to get rid of the main 'DC' component and apply a Hanning window to both images. The purpose of the Hanning window is to reduce the influence of the edges of the image on the correlation operation.
3. Compute the 2-dimensional Fourier transform of the reference image: $F = \mathcal{F}\{R\}$, and the 2-dimensional Fourier transform of the image to be stabilised: $G = \mathcal{F}\{I\}$. Then calculate the cross power spectrum: $C = \dfrac{F\,\overline{G}}{|F\,\overline{G}| + \epsilon}$. The factor ϵ is added here to eliminate the rare case of division by zero. The value of ϵ used in the code is 10^{-12}.
4. Perform an inverse Fourier transform on the cross power spectrum: $c = \mathcal{F}^{-1}\{C\}$
5. Convert the peaks of the cross power spectrum c $((p_y, p_x))$ to the corresponding pixel shift $((d_y, d_x))$ via some wrap-around rules that account for the larger size of the c matrix.
6. Set the corrective shift $(\Delta x, \Delta y) = (-d_x, -d_y)$. These values are rounded to integers (because shifts are limited to whole pixels) and attach this tuple to `frame.stabilization_data`.

4.5.2. Confidence Measure and Reference Update Strategy

Besides providing a displacement estimate, the phase-correlation stabiliser also outputs a scalar confidence measure, primarily the Peak-to-Sidelobe Ratio (PSR) of the correlation surface. PSR-type metrics are widely used to assess the quality of correlation peaks in correlation-filter and registration systems [14, 15, 27]. In typical SEM sequences, a well-registered frame pair should yield a very sharp and unambiguous correlation peak with a high PSR value.

In the implemented system, PSR is used exclusively to decide when to update the reference image, not to trigger a beam shift. We chose to implement the beam shift threshold based on the displacement of the incoming frame because the result has to be a video that is nice to watch. Therefore, we do not want to make big shifts in the software even if the software is capable of correcting these shifts. So the PSR can remain high while we still choose to perform a beam shift.

Based on the degradation and shift experiments reported in subsection 5.1.1 and section E.3, we configured two fixed, resolution-dependent PSR thresholds that determine when to update the reference image:

$$\mathrm{PSR}_{\mathrm{thr}} = \begin{cases} 65, & \text{for SD images,} \\ 130, & \text{for HD images.} \end{cases} \tag{4.1}$$

Whenever the instantaneous confidence for a new frame falls below the appropriate threshold,

$$\mathrm{PSR}_t < \mathrm{PSR}_{\mathrm{thr}}, \tag{4.2}$$

the current image can no longer be stabilised reliably with the current reference frame. In that case, the controller updates the reference frame: the new reference is taken to be the *previously received image that gives a good confidence value.* This design choice ensures that a persistent decrease in confidence caused by a

genuine change in the specimen (e.g. intentional patterning, damage, or charging) does not permanently disable stabilisation; instead, the system adapts by re-anchoring the reference to the new state of the sample, a strategy consistent with recommendations in registration surveys [27].

Beam shift decisions are driven purely by the drift estimate: if the magnitude of the estimated corrective shift exceeds the maximum software shift capacity allowed in its given direction (expressed as a percentage of the field of view, see section 4.8), a beam shift command is issued. PSR does not enter this decision.

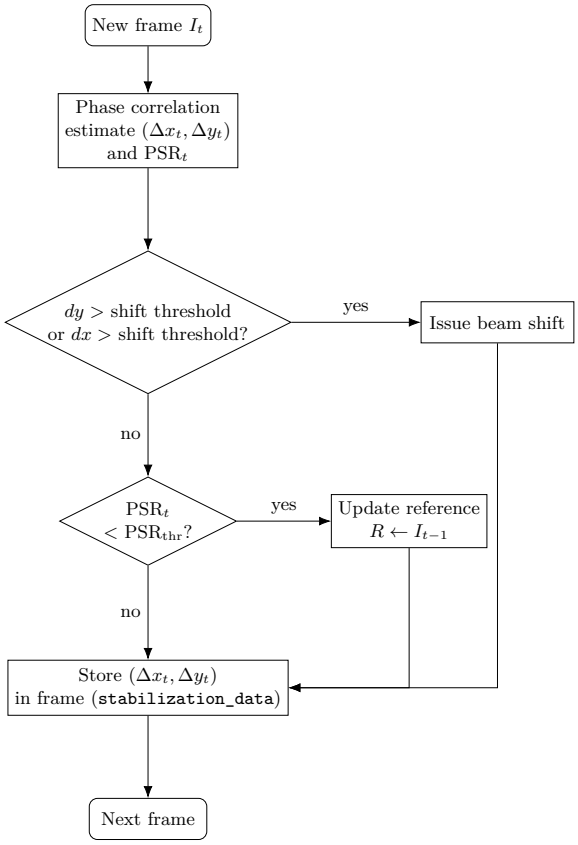


Figure 4.2: Logical flow of the stabiliser decision process. For each new frame, the stabiliser estimates a shift and a PSR-based confidence value. If the geometric shift threshold is exceeded, a beam shift is issued. If the PSR falls below the configured threshold, the reference frame is updated to some previously received image. In all cases, the estimated shift $(\Delta x_t, \Delta y_t)$ is stored with the frame as stabilisation data; the actual image translation is applied later, in a separate outpainting step once the full sequence has been acquired.

4.6. Pixel Drift and Physical Displacement Conversion

As was described in section 4.5.2, the image stabilisation software is able to compute the amount of shift between two images in pixels. However, such a value is not directly usable in the feedback loop. To make this value usable, it needs to be converted to a shift in millimetres (as mentioned in Requirement [A.4]), as is standard in SEM drift-correction workflows [28, 29, 30]. The way this conversion is performed is described in this section.

4.6.1. Finding the Physical Displacement Using the SEM Metadata

The image data used in this project is stored using the Tagged Image File Format (TIFF), a well-established standard for lossless image storage [20]. We found that the SEM uses the metadata embedding functionality of this standard. The values that are stored in the FEI TIFF files by the microscope control software allow the image stabilisation software to derive the pixel size needed for this conversion.

The relative lateral displacement between two SEM frames in pixels is determined using phase correlation, as described in section 4.5.2 and in the phase-correlation literature [11, 13, 21, 22, 14].

Extraction of SEM Parameters from Metadata

Every SEM TIF file contains a structured metadata block, which includes the parameters required to compute the physical field of view (FOV) associated with the acquired image. The relevant entries are:

- `ImageDevice.SizeY`: the vertical size of the detector field in millimetres,
- `Vector.Magnification`: the microscope magnification used during acquisition,
- the recorded image resolution (N_x, N_y) can be taken from the Frame class after the image is loaded using the `.size` parameter.

These fields follow the conventions documented for FEI XL/ Sirion systems [31, 32, 33].

With these values, the physical height, or FOV, of the image is given by:

$$\mathrm{FOV}_y = \frac{\mathrm{SizeY}}{\mathrm{Magnification}}, \tag{4.3}$$

where FOV_y is expressed in millimetres. Assuming square pixels[1], the corresponding pixel size can be calculated as:

$$s_{\mathrm{px}} = \frac{\mathrm{FOV}_y}{N_y}, \tag{4.4}$$

where N_y is the number of pixels in the vertical direction. The quantity s_{px} represents the physical dimension of a single pixel in millimetres on the specimen surface.

Conversion of Pixel Shift to Physical Displacement
Given the calculated pixel shift $(\Delta x_{\mathrm{px}}, \Delta y_{\mathrm{px}})$ between two frames and the pixel size s_{px}, the real displacement of the electron beam or specimen is obtained by a direct, and simple transformation:

$$\Delta x_{\mathrm{mm}} = \Delta x_{\mathrm{px}} \cdot s_{\mathrm{px}} \tag{4.5}$$
$$\Delta y_{\mathrm{mm}} = \Delta y_{\mathrm{px}} \cdot s_{\mathrm{px}}. \tag{4.6}$$

These quantities represent the actual physical shift of the scanned area on the sample surface between the two acquisitions. Since the SEM raster defines the image coordinate system directly on the specimen, this conversion captures both beam drift and mechanical stage drift.

Because all required scale information is obtained directly from the SEM metadata, this procedure does not require any additional user-performed geometric calibration or manual use of scale bars. In section 4.7 we nonetheless perform an independent, empirical calibration of the pixel size to verify the accuracy and stability of the metadata-based conversion.

4.7. Empirical Calibration of SEM Pixel Size

Quantitative analysis of scanning electron microscopy (SEM) images requires a reliable mapping between pixel coordinates and real-space distances [28, 29, 30, 31, 33]. In the workflow described in section 4.6, an initial estimate of the pixel size, from this point denoted as p_{meta}, was obtained.

Experimental inspection revealed immediately after implementing this feature that measurements performed on the same images using the SEM software's internal scale bar were systematically inconsistent with p_{meta}, similar to the calibration issues discussed in SEM metrology and drift-correction studies [29, 30, 34]. Although the scale bar drawn by this software cannot be guaranteed to be perfectly calibrated (see section 4.7.5), it is assumed to be good enough for our requirements, as the default scale bar also suffers from similar problems. We therefore adopt the pixel size inferred from the scale bar, p_{scale}, as a practical reference against which the metadata-based estimate is calibrated.

4.7.1. Data and Ratio Definition

The SEM offers multiple raster formats (standard-definition and high-definition acquisition). To assess the consistency of the metadata-based calibration across a representative set of conditions, we assembled a dataset of $N = 21$ images covering a range of magnifications and dwell times and taken in both SD and HD. For each image i we determined:

- $p_{\mathrm{meta},i}$: pixel size estimated from TIFF metadata,
- $p_{\mathrm{scale},i}$: pixel size implied by the SEM scale bar,
- the ratio

$$r_i = \frac{p_{\mathrm{scale},i}}{p_{\mathrm{meta},i}}. \tag{4.7}$$

[1] FEI SEM raster scan geometry guarantees equal sampling in the horizontal and vertical directions.

To compute the pixel size of the scale bar, each image was opened individually in image editing software. Using this software, the length of the scale bar in pixels could be calculated. Because the ends of the bars were more than one pixel thick, an assumption was made that the true size of the scale bar is measured from the centre of the left vertical edge bar to the rightmost pixel of the right-side vertical edge bar. This reference was determined by using the true measurement tool in the existing SEM software. By then looking at the corresponding label of the scale bar, the pixel size inferred from the scale bar ($p_{\mathrm{scale},i}$) can be calculated using:

$$p_{\mathrm{scale},i} = \frac{L_{bar,mm}}{L_{bar,px}} \tag{4.8}$$

For presentation, the frames are labelled as Image-1 to Image-21. The complete dataset, including the corresponding file name, and the pixel sizes according to the metadata and the scale bar, is reported in Table 4.2.

Table 4.2: Metadata pixel size p_{meta}, scale-bar pixel size p_{scale}, and ratio $r = p_{\mathrm{scale}}/p_{\mathrm{meta}}$ for all calibration images. Values are given in millimetres.

ID	Image	p_{meta} (mm/px)	p_{scale} (mm/px)	r
Image-1	1 2.TIF	1.9112e-05	1.7094e-05	0.894433
Image-2	1.TIF	2.5482e-05	2.2989e-05	0.902144
Image-3	1000000.TIF	1.9112e-07	1.7301e-07	0.905265
Image-4	10MIN.TIF	7.3037e-05	6.5789e-05	0.900767
Image-5	1304.TIF	5.8805e-06	5.3191e-06	0.904543
Image-6	1500.TIF	1.2741e-04	1.1494e-04	0.902144
Image-7	2.TIF	1.9112e-05	1.7094e-05	0.894433
Image-8	200000.TIF	9.5558e-07	8.5470e-07	0.894433
Image-9	25MIN.TIF	2.9215e-05	2.6596e-05	0.910349
Image-10	2MIN.TIF	4.1736e-05	3.7594e-05	0.900767
Image-11	30MA_100.TIF	2.9215e-05	2.6596e-05	0.910349
Image-12	30MA_2.TIF	2.2473e-04	2.0202e-04	0.898947
Image-13	30MIN20.TIF	9.7383e-06	8.7719e-06	0.900767
Image-14	3500.TIF	5.4604e-05	4.9383e-05	0.904371
Image-15	350000.TIF	5.4604e-07	4.9383e-07	0.904371
Image-16	4.TIF	2.5482e-05	2.2989e-05	0.902144
Image-17	4MIN.TIF	4.1736e-05	3.8168e-05	0.914519
Image-18	5MIN.TIF	2.9215e-05	2.6596e-05	0.910349
Image-19	8000.TIF	2.3889e-06	2.1368e-06	0.894433
Image-20	8MIN.TIF	5.8430e-05	5.3191e-05	0.910349
Image-21	INITIAL.TIF	5.8430e-05	5.3191e-05	0.910349

4.7.2. Statistical Characterisation

For each calibration image, we define the ratio r_i as in Eq. 4.8, quantifying the discrepancy between metadata-based and scale-bar-based pixel sizes. For the $N = 21$ calibration images, we obtain

$$\bar{r} = 0.9033, \tag{4.9}$$

$$s = 0.0061, \tag{4.10}$$

$$\mathrm{CV} = \frac{s}{\bar{r}} \approx 0.67\%, \tag{4.11}$$

where s denotes the sample standard deviation of the ratios $\{r_i\}$, and CV is the coefficient of variation, defined as the ratio of the standard deviation to the mean [35, 36]. The ratios range from approximately 0.894 to 0.915, i.e. within about 2.3% of the mean.

Approximating the sampling distribution of $\bar{r}$ by a normal distribution via the central limit theorem [36], the corresponding 95% confidence interval for the mean ratio is

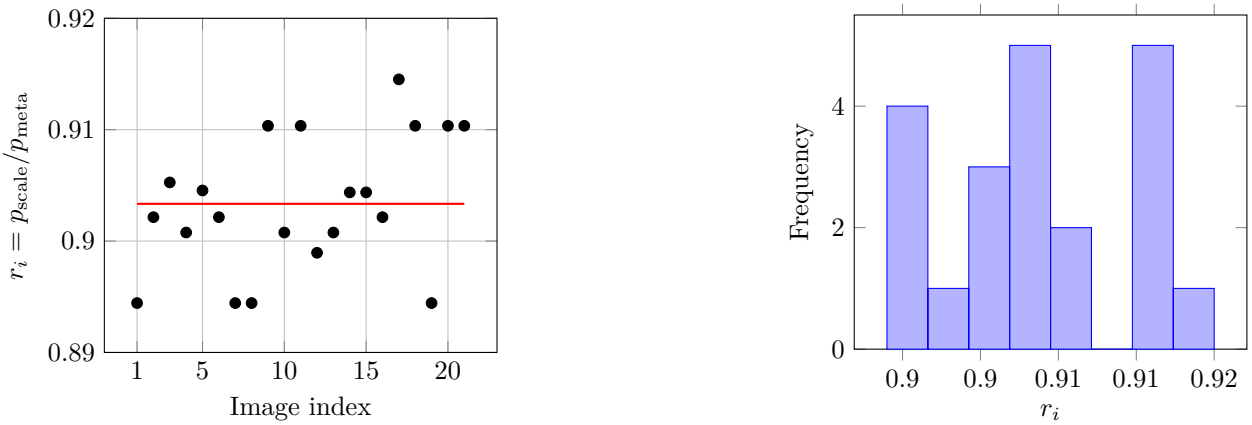
$$\bar{r} \pm 1.96\frac{s}{\sqrt{N}} \approx (0.901,\ 0.906), \tag{4.12}$$

following the standard formula for confidence intervals for a mean [36].

Thus, the metadata-derived pixel size is consistently *slightly larger* than the scale-bar-derived pixel size. On average one must multiply p_{meta} by a factor of about 0.90 to match the scale-bar. No statistically significant dependence of r_i on acquisition settings is visible in this dataset. All ratios cluster tightly around a single mean value with sub-per cent relative spread.

4.7.3. Graphical Summary

Figure 4.3 provides a graphical summary of the ratio data. Subfigure 4.3a shows the individual r_i values as a function of image index, together with the global mean ratio $\bar{r}$. All 21 images cluster tightly around the same mean, with no obvious outliers. Subfigure 4.3b displays a histogram of all ratios. The distribution is unimodal and approximately symmetric about $\bar{r}$ with a spread below 1%, which supports the interpretation that the discrepancy between metadata-based and scale-bar pixel-size estimates is well described by a single multiplicative factor.



(a) Ratios r_i by image index. The red line indicates the global mean $\bar{r}$.

(b) Histogram of ratios r_i for all calibration images.

Figure 4.3: Graphical analysis of the ratio $r_i = p_{\text{scale}}/p_{\text{meta}}$ between scale-bar and metadata-derived pixel sizes for the 21 calibration images.

4.7.4. Adopted Correction Model

Based on the foregoing analysis, the corrected pixel size will be modelled as:

$$p_{\text{corr}} = k\, p_{\text{meta}}, \tag{4.13}$$

with a global, instrument-specific correction factor

$$k = \bar{r} \approx 0.903. \tag{4.14}$$

Within the examined dataset, this single multiplicative factor reduces the systematic discrepancy between metadata-derived and scale-bar-derived pixel sizes to the level of random variation (below 1%), and it is valid across the range of magnifications and dwell times represented in the calibration set. This error in the pixel to millimetre conversion is also smaller than the maximum error requirement set in requirement [B.8]. All subsequent quantitative measurements in this work use p_{corr} as the operative pixel size.

4.7.5. Discretisation and Rounding of the Printed Scale Bar

Even with the calibrated pixel size p_{corr} from subsection 4.7.4, an overlaid scale bar cannot represent arbitrary real-world distances exactly. The scale bar is drawn as a finite number of pixels and labelled with a human-readable value in nm, μm, or mm. Both the bar length (in pixels) and the printed numerical value are therefore discretised, which leads to small, controlled deviations from the "ideal" continuous scale and from the scale bar drawn by the SEM software.

The algorithm that determines the pixel length of the scale bar is presented in appendix C. Its software implementation is given in appendix F.7.

In this algorithm, two independent discretisation sources are present:

- the label value v is restricted to the finite set $\mathcal{V}$,
- the rendered bar length N_{bar} must be an integer number of pixels.

As a consequence, the *effective* physical length of the drawn bar,

$$N_{\text{bar}}\, p_{\text{corr}}, \tag{4.15}$$

will, in general, differ slightly from

$$L_{\text{label}}, \tag{4.16}$$

and both may differ slightly from the scale bar produced by the SEM acquisition software, which uses its own (undocumented) rounding and formatting rules. This length difference results in an absolute error of:

$$|N_{bar}p_{corr} - L_{label}| \tag{4.17}$$

From the relationship between these parameters defined in equation C.4, it can be seen that the value in equation 4.17 is exactly the error introduced by rounding, which is at most half a pixel.

Now, if the relative error is computed, and considering the just-mentioned relationship:

$$\left|\frac{N_{\text{bar}}\, p_{\text{corr}} - L_{\text{label}}}{L_{\text{label}}}\right| \leq \frac{p_{\text{corr}}}{2L_{\text{label}}} \tag{4.18}$$

In general, the worst-case length of the bar is equal to 17% of the width of an SD image, so $0.17 \cdot 712 \approx 121$ pixels. With this minimum length, the error e can be computed as:

$$e \leq \frac{p_{corr}}{2L_{label}} = \frac{p_{corr}}{2 \cdot 121 p_{corr}} = \frac{1}{242} = 0.0041 = 0.41\% \tag{4.19}$$

However, in certain (rare) cases, due to the discretisation of the label value, this minimum length may be left out of consideration, resulting in the fact that the bar may get smaller than the minimum length limit. If the assumption is made that if this happens in the worst case this length will go all the way down to 2.5% of an image width, which in SD mode corresponds to 17.8 pixels, then calculating the error with $L_{label} = 18p_{corr}$ (rounding this value just like it would in the algorithm) will result in the error e given by:

$$e \leq \frac{p_{corr}}{2L_{label}} = \frac{p_{corr}}{2 \cdot 18 p_{corr}} = \frac{1}{36} = 0.02778 = 2.778\% \tag{4.20}$$

A direct comparison with Requirement [B.8] is now possible. From the statistical characterisation in section 4.7.4, the uncertainty on the calibrated pixel size p_{corr} is small: the coefficient of variation is only CV $\approx 0.67\%$, the 95% confidence interval for the mean ratio is $(0.901, 0.906)$, and even under pessimistic assumptions the resulting error in p_{corr} remains on the order of 2%. The additional discretisation of the scale bar itself contributes at most 0.41% for typical bar lengths, and only in a deliberately extreme worst case (very short bar combined with unfavourable rounding) reaches 2.78%. Taken together, these results show that the relative deviation between our rendered scale bar and the SEM softwares scale bar remains well below the 2.5% limit imposed by Requirement [B.8] for all practically relevant scenarios, with only a narrow quantified edge case in which this bound may be slightly exceeded if both calibration error and discretisation error simultaneously take on their worst admissible values.

4.7.6. Visual Comparison with the SEM Scale Bar

As a final sanity check, the calibrated pixel size p_{corr} and the discretised scale-bar drawing procedure were compared directly against the scale bar printed by the SEM software itself, in the spirit of existing SEM drift and distortion-correction work [28, 29]. Figure 4.4 shows a representative example: the original SEM frame with its native $5\,\mu$m scale bar (bottom) and the application-generated $5\,\mu$m bar (although the text is not visible in this image) overlaid above.

Figure 4.4: Direct comparison between the SEM software scale bar (top) and the application-generated scale bar (bottom), both labelled $5\,\mu$m (although not visible in this example). Both bars have essentially the same physical length when measured from end to end in pixels; any residual difference is on the order of a fraction of a pixel and visually negligible.

To quantify the error of the application-generated bar, both bars in figure 4.4 were measured using image editing software from the outer end of one vertical marker to the outer end of the other. The resulting pixel

lengths agree to within approximately one pixel, and in many of the other test cases are indistinguishable at this resolution. In relative terms, this corresponds to a sub-per cent deviation for the bar lengths used here, and is of the same order of magnitude as both the rounding error introduced by drawing the bar with an integer number of pixels (section 4.7) and the error in the corrected pixel sizes.

This end-to-end agreement confirms that the empirically calibrated pixel size p_{corr} and the scale-bar rendering procedure are generally consistent with the existing SEM software's own internal scale bar to within the limits set by the requirement [B.8] as the scale bar error in most cases depends mainly on the calculated pixel size. The only cases where the lengths were not in agreement were in the cases where the SEM software and this application chose a different display length (so choosing $2\,\mu$m instead of $5\,\mu$m). However, when these lengths were then manually scaled (so for the given example, multiplying the measured length by 2.5), the bars would still be accurate to within one pixel.

4.8. Drift Correction Capacity and Beam-Shift Limits

The calibrated pixel size p_{corr} allows us to express not only individual frame-to-frame shifts in physical units, but also the overall capacity of the stabilisation system. In practice, drift is first compensated in software by translating the image according to the output of the stabilisation function. A hardware beam shift of up to ± 20 μm in both directions is performed by the microscope control software when the stabiliser decides that purely software correction is no longer sufficient. This decision is based on the magnitude of the required software correction relative to the field of view.

Because the system can operate with both standard-definition (SD) and high-definition (HD) raster formats, the requirements were not defined in pixels. A displacement of 100 pixels has a very different physical meaning in SD than in HD. Instead, all shift requirements in this thesis are expressed as a percentage of the field of view, defined in a consistent way for any image size.

This section formalises these concepts, introduces the shift-percentage convention, and defines percentage-based quantities that allow the system's capabilities to be summarised concisely, in line with previous discussions of SEM drift and compensation ranges [28, 29, 30, 34, 37, 38].

4.8.1. Shift Percentage Convention

Let each frame have width W and height H in pixels, and define the image centre with:

$$(x_c, y_c) = \left(\frac{W}{2}, \frac{H}{2}\right) \tag{4.21}$$

The maximum possible displacement of the centre before it just reaches the image border is:

$$x_{max} = \frac{W}{2} \tag{4.22}$$

$$y_{max} = \frac{H}{2} \tag{4.23}$$

We define a dimensionless shift fraction $s \in [0, 1]$ such that

- $s = 0$ corresponds to no shift (centre stays at (x_c, y_c)),
- $s = 1$ corresponds to shifting the centre all the way to one of the borders along a given axis.

Equivalently, for a given shift fraction s (or "shift percentage" $100s\,\%$), the corresponding per-axis pixel shifts are

$$\Delta x_{\text{px}} = s\,\frac{W}{2} \tag{4.24}$$

$$\Delta y_{\text{px}} = s\,\frac{H}{2} \tag{4.25}$$

A "30% shift", therefore, means that the image centre has moved by $0.3 \times W/2$ pixels horizontally or $0.3 \times H/2$ pixels vertically towards some edge.

All shift thresholds and requirements in what follows are expressed in terms of this fraction of half the field of view. This makes the specification independent of the raster format: doubling the resolution will change the relevant values of W and H, but a given shift percentage s still corresponds to the same relative movement of the field of view.

4.8.2. Geometric Definitions and Threshold Parameters

We now introduce the notation used by the implementation. As in equation 4.21, the frame centre is at (x_c, y_c) with

$$x_{max} = \frac{W}{2} \tag{4.26}$$

$$y_{max} = \frac{H}{2} \tag{4.27}$$

The software stabiliser operates with a dimensionless threshold $0 < \tau \leq 1$, provided by the `ManagerInfo` message as a percentage $100\tau\,\%$. In the terminology above, τ is simply the maximum allowed shift fraction s_{max}:

$$\tau = s_{max}. \tag{4.28}$$

It defines the maximum centre displacement that is handled purely in software without issuing a beam shift command:

$$\Delta x_{\text{th}} = \tau\, x_{max} = \tau\, \frac{W}{2} \tag{4.29}$$

$$\Delta y_{\text{th}} = \tau\, y_{max} = \tau\, \frac{H}{2} \tag{4.30}$$

Comparing with equations 4.24 and 4.25, a configured shift percentage $100\tau\,\%$ is thus translated to per-axis pixel limits via $\Delta x_{\text{th}} = (W/2)\,\tau$ and $\Delta y_{\text{th}} = (H/2)\,\tau$.

The quantities Δx_{th} and Δy_{th} are the largest translations along x and y that will be applied to the framebuffer before the system decides that a hardware correction is necessary. Geometrically, they define a rectangle centred on (x_c, y_c) inside which the image centre is allowed to wander. Its size scales with the chosen shift percentage and with the image dimensions W and H.

Although the implementation applies the thresholds per axis, it is useful to define an isotropic effective threshold in the Euclidean sense,

$$\Delta r_{\text{th}} = \tau\, r_{max} \tag{4.31}$$

$$r_{max} = \sqrt{x_{max}^2 + y_{max}^2} \tag{4.32}$$

which represents the radius of the largest circular region around the image centre that can be compensated purely by software. We use this to define the maximum error around a point in requirements [B.6] and [B.7].

4.8.3. Software Correction Range as Percentage of Field of View

By its definition, the parameter τ is already interpretable as the fraction of half the field of view that can be corrected along each axis. This has a direct geometric interpretation: for example, if $\tau = 0.25$, the stabiliser can compensate for drifts that move the image centre by up to 25% of the half-width x_{max} (or half-height y_{max}) before a beam shift is triggered. In other words, the centre stays inside a rectangle that extends to 25% of the distance from the centre to each border.

Using the calibrated pixel size p_{corr} (in *millimetres per pixel*), the corresponding physical drift tolerance along each axis is

$$d_x^{\text{th}} = p_{\text{corr}}\, \Delta x_{\text{th}} \tag{4.33}$$

$$d_y^{\text{th}} = p_{\text{corr}}\, \Delta y_{\text{th}} \tag{4.34}$$

which can be converted to micrometres by multiplication with 10^3.

4.8.4. Beam-Shift Range and Usage Fraction

Independent of the software stabiliser, the microscope hardware offers a finite beam shift range. This range is defined in requirement [1.1]. In practice, this means that the beam shift is limited to:

$$B_{x,lim} = B_{y,lim} = \pm 0.02 \text{ mm} \tag{4.35}$$

If the required software correction exceeds Δx_{th} (equation 4.29) or Δy_{th} (equation 4.30), a beam shift is performed to bring the image back towards the centre. A similar approach is taken in other SEM drift-compensation schemes [28, 34, 37, 38].

The magnitude of the software threshold relative to the available beam shift range can be expressed as

$$R_x = \frac{d_x^{\text{th}}}{B_x} \times 100\% \tag{4.36}$$

$$R_y = \frac{d_y^{\text{th}}}{B_y} \times 100\% \tag{4.37}$$

where d_x^{th} and d_y^{th} are the threshold drifts in physical units (see previous subsection). These quantities specify what fraction of the hardware beam shift budget is "used up" by the maximum software correction for a given shift percentage τ.

Note that R_x describes how close the user-specified threshold τ allows the drift to approach the hardware limit B_x *at the moment a beam shift is triggered*. It does not measure how often beam shifts occur: in practice, a smaller τ leads to more frequent beam shifts (since the threshold is crossed sooner), even though the corresponding R_x is smaller.

These usage fractions can be used to check whether there is sufficient headroom for a beam shift at a given threshold level. Through the magnification dependence of the field of view (equations 4.3 and 4.4), R_x and R_y can be related to specific magnification settings. This allows the user to choose threshold levels that maintain enough beam shift headroom at a given magnification, ensuring that the system can still meet the correctable shift limits in requirements [B.4] and [B.5] and avoid visible artefacts such as non-optimal borders in the output video when no beam shifts are performed.

4.8.5. Summary of Quantities

Table 4.3 summarises the main quantities introduced in this section. Together, these definitions provide a compact, percentage-based description of how much drift the combined software stabiliser and beam shift controller can compensate, both in pixel units and in physically meaningful micrometres.

Table 4.3: Overview of drift-related quantities used to characterise the stabiliser and beam shift capacity

Symbol	Definition	Interpretation
τ	software threshold (0–1)	fraction of half-FOV tolerated in software
Δx_{th}	$\tau W/2$	max software shift along x (pixels)
d_x^{th}	$p_{\text{corr}} \Delta x_{\text{th}}$	max software drift along x (mm)
B_x	given (e.g. 20 μm)	beam shift limit along x
R_x	$(d_x^{\text{th}}/B_x) \times 100\%$	usage of beam shift range at threshold

4.9. Out-painting-Based Software Drift Correction

During acquisition, the stabiliser estimates for every incoming SEM frame a translational drift vector $(\Delta x_i, \Delta y_i)$ relative to a reference frame. This drift information is used in two ways:

- *Beam shift mode*: the estimated drift is used as an input for the control software to control the microscope to perform a beam shift, re-centring the field of view over the sample before taking the next picture. This approach is similar to other commonly used techniques [28, 34, 37, 38].
- *Software shift mode*: the images are stored as acquired in memory. Then the estimated drift is compensated after all frames have been captured by shifting the frames digitally before composing them in the final video.

This subsection focuses on the second option, which is always available, independent of microscope hardware and is also used as a fallback when the beam shift feedback loop is not used. A naive implementation of software drift correction would crop all frames to their common intersection after alignment, thereby discarding image content that drifts outside this region. However, for this thesis, this is not an approach that can be used, as requirement [B.1] states that the output resolution may not be lower than the resolution of the input images. Instead, an outpainting strategy is adopted that preserves the full field of view of each frame and fills missing regions. This is conceptually similar to border-handling and padding strategies in video-stabilisation and registration pipelines [2, 4, 39, 40, 41].

4.9.1. Global Padding Strategy

After a full sequence of images is captured, and each image has `stabilization_data`, the padding of the frames can start. To determine the final size that should be padded to, first the maxima and minima of the x and y components of the `stabilization_data` of all frames are found. With these values, the padded height and width can be determined:

$$H_{\text{pad}} = H + (\Delta y_{max} - \Delta y_{min}) \tag{4.38}$$
$$W_{\text{pad}} = W + (\Delta x_{max} - \Delta x_{min}) \tag{4.39}$$

Then, for each frame, the padding heights and padding widths on the top, bottom, left, and right of the frame can be calculated:

$$t_i = \Delta y_i - \Delta y_{min} \tag{4.40}$$
$$b_i = \Delta y_{max} - \Delta y_i \tag{4.41}$$
$$\ell_i = \Delta x_i - \Delta x_{min} \tag{4.42}$$
$$r_i = \Delta x_{max} - \Delta x_i \tag{4.43}$$
$$\tag{4.44}$$

Placing the original image I_i into the padded canvas $\tilde{I}_i$ with size $H_{\text{pad}}, W_{\text{pad}}$ via:

$$\tilde{I}_i[t_i : t_i + H,\ \ell_i : \ell_i + W] = I_i \tag{4.45}$$

will align all frames in a common coordinate system. After padding, the drift-corrected sequence $\{\tilde{I}_i\}$ can be directly written to a video file without any further cropping.

In the function `pad_frames`, which implements the above-mentioned padding algorithm, several padding modes are supported:

- constant black (`'black'`) or white (`'white'`) margins
- edge replication (`'same'`, which takes the first or last row or column and duplicates that over the area of the frame that does not yet have colour data
- mean padding (`'mean'`), filling each row and column with the mean of that row or column
- local-mean outpainting (`'mean_outpaint'`), which iteratively propagates local averages from the known region into the unknown margins. Essentially creating a natural blur. Details on this method are provided in Appendix E.1.

The first four options are useful for debugging and quick visualisation, and can immediately make clear which parts of the image are original and which ones are not. However, they introduce visually obvious artificial borders. Therefore, the local-mean outpainting scheme is generally recommended; this mode makes the transitions on the edges of the video smoother, and in turn, the video is much more pleasant to watch.

A schematic illustration of the frame padding geometry is shown in Fig. 4.5: the original frames (blue) are shifted according to their stabilisation vectors and embedded into a larger common canvas (dashed outline). The missing regions are subsequently filled in.

4.9.2. Preprocessing Checks

These outpainting methods form one (and the most important) part of the preprocessing pipeline that converts the captured frames into a set of frames that are ready to be stitched into a video. After the stabiliser has estimated the drift vectors for all frames, the following steps are executed:

1. The helper functions`check_stabilization_data` and `check_same_size` ensure that all frames contain valid stabilisation vectors and that their dimensions are consistent.
2. The function`pad_frames` computes the global padding widths from the stabilisation data and applies one of the outpainting methods mentioned in subsection 4.9.1 to each frame, producing a sequence of aligned, same-size images. After this, again a sanity check using `check_same_size` is done.
3. Finally, `generate_annotation` appends a metadata overlay below each frame, including acquisition parameters (accelerating voltage, magnification, working distance, etc.) and an automatically sized scale bar, as mentioned in 4.7.5 is added

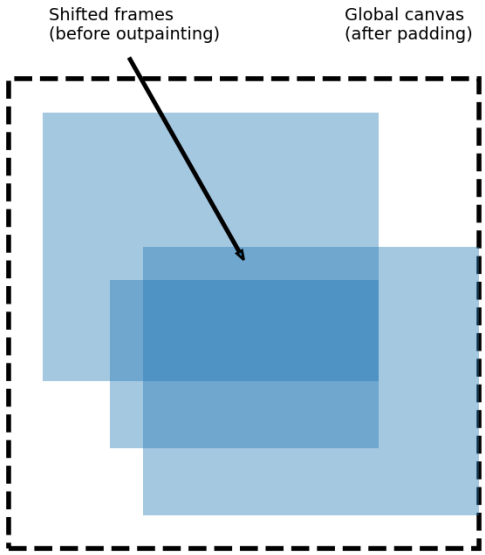


Figure 4.5: Schematic view of the outpainting strategy. Drift-corrected frames are embedded into a common canvas determined by the extrema of the estimated shifts. The resulting empty margins (white) are filled by an outpainting method instead of cropping the sequence to the common overlap.

4. Now all the frames are ready to be written into a video file. See section 4.10.

This design cleanly separates the physics-dependent drift estimation from the purely geometric and visual task of assembling a stable video. Beam shift and software shift are both driven by the same stabiliser output: in live acquisition, while running with the control software, the drift vector is used to recentre the beam (if the set threshold is exceeded), while in stand-alone mod,e the same vector determines the padding offsets and outpainting region. As a result, the final video faithfully reflects the corrected motion of the specimen without sacrificing the field of view.

4.10. Video Generation

After all the previously mentioned video processing steps have been performed, the framebuffer contains stabilised frames with a custom data bar. Now the framebuffer is ready to be processed into a video.

First of all, the video file extension is read from the settings that came from a `ManagerInfo` message at the beginning of program execution. Based on the extension, the applicable codec is chosen. The image stabilisation software supports the `mp4v`-codec (for `.MP4` files) and the `ffV1`-codec (for `.MKV` files). The `mp4V`-codec was implemented to be able to export video in a commonly used format. However, the `mp4v`-codec is a lossy format, meaning that some video data is compressed irrecoverably. For this reason, support for the `ffv1`-codec was also added to enable lossless video export.

Once the codec is determined and the location to store the final video is also read from the `ManagerInfo` parameters, a call is made to the OpenCV library (see subsection 3.2) to turn the images in the framebuffer into a video.

4.11. Design Alternatives and Rationale

In the sections above, all choices made are described, and the resulting implementations are explained. However, in developing the SEM stabilisation and video pipeline, several key design choices had to be made. This section discusses the main alternatives that were considered and explains why the final design ultimately uses

i frequency-domain phase correlation

ii outpainting rather than cropping

iii deferred application of frame shifts

iv the peak-to-sidelobe ratio (PSR) as the primary confidence measure

4.11.1. Stabiliser Algorithm

The fundamental task of the stabiliser is to estimate the translational shift between two images. As explained in section 3.1, many classes of methods can, in principle, be used for this, including block matching, optical flow, feature-based matching, and correlation-based approaches. A broader overview of these families can be found in standard image-registration surveys such as Zitová and Flusser [27].

For the XL30 system, the stabiliser must satisfy several constraints that follow directly from the programme of requirements:

- it must be fast enough to run between successive frame captures on legacy Windows 7 hardware, so that image processing does not disturb the acquisition timing (requirements [3.3] and [4.4]);
- it must be robust to the characteristic noise, contrast variations and charging artefacts of SEM images [30, 9], so that intensity fluctuations do not cause the shift error to exceed the specified bounds (requirements [C.3], [B.6] and [B.7]);
- it must provide a well-defined confidence measure to decide whether a shift estimate is reliable (Chapter 3.3), so that unreliable estimates can be rejected without aborting the experiment, in line with the requirement that processing of erroneous data must not interrupt a run (requirement [B.3]) and that residual motion in the output video remains within the allowed limits (requirements [3.5] and [3.6]).

Block-matching methods, which compare local image patches over a discrete search window [8, 27], were rejected in this project mainly due to their computational cost and their tendency to produce ambiguous local matches in repeated or low-texture regions, a known limitation of patch-based correlation methods [27]. Unlike block matching, phase correlation remains robust in the presence of repeated microstructure because the global phase alignment still produces a distinct correlation peak, provided the image contains sufficient frequency content [14]. Optical-flow techniques such as Lucas–Kanade and Horn–Schunck rely on brightness constancy and smooth motion fields; these assumptions are frequently violated for SEM images, where contrast can change non-linearly, and the signal-to-noise ratio can be low [9]. Feature-based approaches can also struggle on our SEM sequences, where many scenes contain only weak or repetitive structure, leading to few stable and repeatable keypoints and unreliable matches [10, 27]. For these reasons, the stabiliser focuses on correlation-based methods, which operate directly on image intensities and are well-suited to predominantly translational drift [27].

Spatial-Domain Normalised Cross-Correlation. The simplest correlation-based approach is spatial-domain normalised cross-correlation (NCC), which slides one frame over the other and computes a correlation coefficient for each candidate shift. This corresponds to classical template matching as described, for example, in [20, 42]. While NCC is conceptually straightforward and can provide accurate matches, it has two major drawbacks for high-resolution SEM imagery:

- A naive implementation has computational complexity $\mathcal{O}(N^2)$ per frame pair for an N-pixel image, and even optimised variants remain relatively expensive for large search windows, as discussed for block-matching and correlation-based motion estimation in [8, 27]. This makes real-time use challenging at HD resolutions on the available hardware.
- The correlation peak is sensitive to global intensity and contrast changes, as well as to low-frequency background variations. Charging effects and detector instabilities in SEM can introduce exactly such variations [28, 30], which degrade NCC robustness.

Phase Correlation. As described in section 3.1.1, phase correlation instead works in the Fourier domain. For pure translations, the cross power spectrum C ideally contains only phase ramps; its inverse FFT yields a sharp impulse at the true shift, largely independent of the magnitude of the Fourier coefficients. Phase correlation and its extensions are well-established for image registration and drift estimation [11, 13, 21, 22, 14, 27] and have also been applied explicitly to SEM drift compensation [37].

Compared to spatial-domain NCC, the advantages of phase correlation for this project are:

- FFT-based implementations scale as $\mathcal{O}(N \log N)$ per frame pair, which is more tractable for HD frames [4.4].
- The phase-only normalisation makes the method largely invariant to multiplicative brightness and contrast changes, which helps to satisfy requirement [C.3] and mitigates the impact of charging and detector gain fluctuations [13, 21].
- The resulting correlation surface naturally supports confidence measures such as PSR (section 3.3), which are essential for rejecting unreliable shift estimates in noisy or low-contrast conditions [15, 14].

Selection of Phase Correlation. Given the limitations of block matching, optical flow, and feature-based methods for SEM data (section 3.1), and the practical drawbacks of spatial-domain NCC, phase correlation offers the best balance between robustness, computational efficiency and implementation complexity for the XL30 setup. Empirical tests on representative SEM sequences (Chapter 5.1) showed that phase correlation provides accurate and stable shift estimates across a range of magnifications, drift magnitudes and intensity conditions. For these reasons, phase correlation was adopted as the core stabilisation algorithm in this project, and all subsequent drift-correction and confidence-estimation steps are built on top of this method.

4.11.2. Frame Support: Cropping vs. Outpainting

After estimating the drift, there are two main ways to construct a stabilised video:

1. Cropping all frames to their common overlap after alignment.
2. Outpainting each frame into a larger canvas that accommodates all observed shifts, filling the resulting empty regions with other values.

Cropping has the advantage of producing frames with completely valid data and no made-up data in every pixel. However, this comes at the cost of discarding potentially large parts of the original field of view, especially when long-term drift occurs. In SEM applications, this is undesirable because:

- features of interest near the edges may be cut off;
- the effective field of view depends on the (unknown in advance) drift trajectory;
- different stabilised videos of the same raw data (e.g. with different parameters) might not be directly comparable if they crop differently.

Similar trade-offs between cropping and padding appear in the video-stabilisation literature [2, 4, 39, 40].

The chosen design uses an outpainting strategy (section 4.9): all frames are embedded into a common canvas large enough to cover the extrema of the estimated shifts, and the empty margins are filled using one of the outpainting algorithms (subsection 4.9.1). Outpainting-like strategies are also used to avoid excessive cropping in video stabilisation and motion-compensated sequences [2, 40, 41].

This approach:

- preserves the full field of view of each frame, which corresponds to requirement [B.1]
- ensures that all frames in the stabilised video have the same dimensions, independent of the drift path
- produces visually smooth margins that do not interfere with interpretation of the specimen.

4.11.3. Timing of Outpainting

A further design choice is whether to apply the estimated shift immediately to each frame as it arrives, or to store the stabilisation data and apply all shifts in a separate post-processing step.

Applying the shift in real time has the apparent advantage that the framebuffer always contains already-aligned images. However, this design has several drawbacks:

- The original raw data are irreversibly modified; any change in the registration algorithm, parameters, or calibration would require re-acquisition.
- Numerical rounding in repeated resampling (e.g. if a frame is later shifted again) can accumulate and degrade image quality.
- Debugging and benchmarking of different stabilisation strategies becomes more difficult, because the exact shift history is not retained in a structured form.

In the implemented system, each `Frame` object stores the estimated stabilisation vector `stabilization_data`, but the image data are kept unaltered. After the full image capture sequence, the function `pad_frames` uses the stored shifts to construct the final aligned and outpainted frames in one step.

This deferred design offers several advantages:

- The raw SEM data are preserved, enabling re-processing with improved algorithms, different confidence thresholds, updated pixel-size calibrations, different outpainting methods or video export on a different output FPS.
- Both beam shift and software shift modes can use the same stabilisation data: in live operation, the drift vector is sent to the microscope, while the final preprocessing step uses exactly the same vector for outpainting.
- All geometric resampling is performed once, in a controlled step, avoiding repeated interpolation and rounding artefacts.

4.11.4. Confidence Measure

The phase-correlation output provides several scalar quantities that can be used as confidence measures for the estimated shift:

- PSR, peak-to-sidelobe ratio, measuring how dominant the main correlation peak is compared to the surrounding sidelobes.
- PAR, peak-to-average ratio, comparing the peak height to the average height of the correlation spectrum.
- P2SR, peak to second peak ratio, a very similar approach to PSR, however, this approach only looks at the second largest peak.
- sharpness, quantifying how concentrated the peak is in space.

Such measures are discussed in correlation-filter performance analyses and registration surveys [15, 14, 27] and are looked at in Section 3.3.

All four metrics were recorded in the stabiliser benchmark dataset CSV file found in Appendix B.2. For each frame pair, the localisation error e was computed from the difference between estimated and ground-truth shifts. We classify registrations with $e \leq 2$ px as *successful* and those with $e > 2$ px as *failures*. This is not based on a requirement for the system but purely as a means to be able to compare these different confidence measures.

A detailed statistical comparison of these metrics is reported in Appendix B.1. In summary, several metrics would be acceptable candidates from a purely statistical perspective, but PSR was selected as the primary confidence measure in the image stabilisation software for the reasons outlined in Appendix B.1.1

5

Verification and Evaluation

The goal of this chapter is to look at the performance of the algorithms in the stabilisation software, and to evaluate their performance based on the requirements set in chapter 2.

5.1. Phase-Correlation Performance with Degraded Image Quality

This section evaluates the performance and robustness of the phase-correlation stabiliser described in section 3.1 and subsection 4.5.2. The stabiliser is a core component of the real-time SEM acquisition pipeline, where it must detect and quantify inter-frame drift under strongly varying imaging conditions, in line with standard practice in image registration and Fourier-based correlation methods [11, 21, 14, 27]. Our goal is to determine the range of shifts, blur levels, noise levels, and structural changes for which the stabiliser remains accurate and for which its confidence metrics provide reliable indicators of failure.

All tests are based on experimentally acquired SEM images of size 712 px × 484 px. Synthetic transformations are applied to these reference images to generate controlled conditions that would be difficult or impossible to reproduce consistently on the physical microscope, as is common in registration benchmarks [14, 27]. For the dataset used here, the parameter grid produces $N = 2625$ transformed frames. Consisting of 945 frames with synthetic translational drift, 315 frames with pure Gaussian blur, 315 frames with additive Gaussian noise, 300 frames with Poisson noise and 750 frames with structural damage.

For each frame we store: the ground-truth shift $(\Delta y_{\text{true}}, \Delta x_{\text{true}})$, the estimated shift $(\Delta y_{\text{est}}, \Delta x_{\text{est}})$, the localization error e, the stabiliser confidence metrics (PSR, PAR, P2SR, sharpness) as descirbed in subsection 4.11.4, in analogy with correlation-filter performance measures [14, 15].

These frames are made from transformations on a dataset of $N = 15$ images.

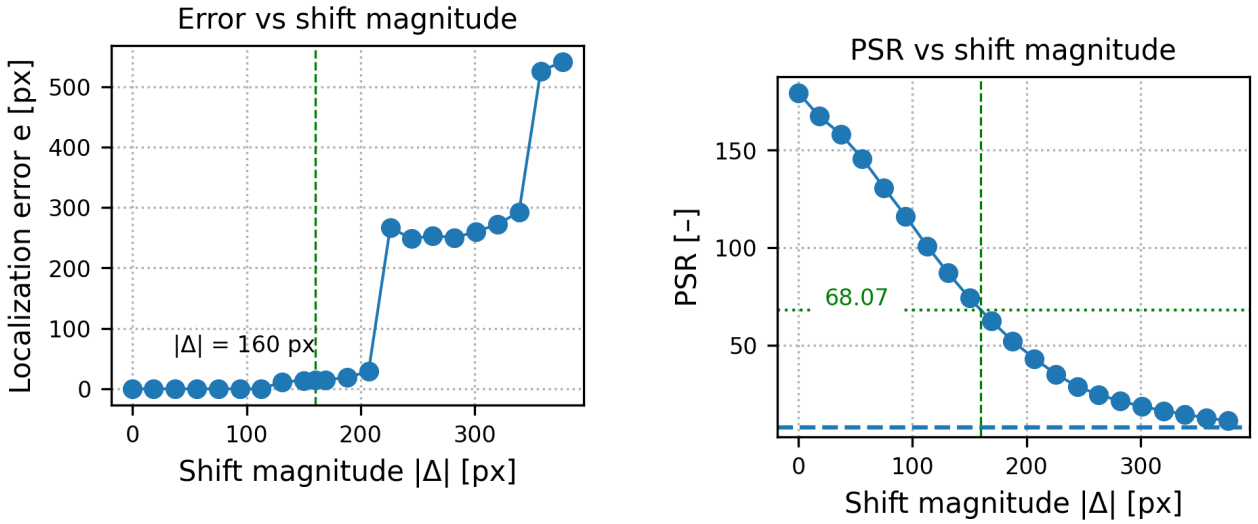
Representative transformed images generated by the Python test framework, as well as explanations of what these transformations are, is included in appendix B.3

5.1.1. Results: Effect of Translational Drift

Localisation error and PSR as a function of shift magnitude are shown in Figure 5.1. These plots are generated directly from the experiment CSV file found in Appendix B.2.

- The localisation error is essentially zero for small and moderate shifts: for all tested directions, the mean error remains exactly 0 px up to $|\Delta| \approx 113$ px. Beyond this point, the mean error increases rapidly with shift magnitude. At $|\Delta| \approx 131$ px the mean error is already ≈ 11 px, and at the largest tested shift of 377 px the mean error reaches ≈ 541 px. The worst-case error across all directions and magnitudes is ≈ 712 px, i.e. essentially the full image width, in line with overlap-related failure modes of phase correlation [11, 21, 14]. Considering requirements [B.4] and [B.5], the maximum shift that the image stabilisation software must be able to recover on the SD images in the dataset is 238 pixels in the horizontal direction, and 161 pixels in the vertical direction. For the vertical shift, the error is at the margin defined in requirement [B.6]. For the horizontal shift, this requirement was not met. However, it is clear from the figure 5.1 that up to the 1/3 shift, the error does not yet go up steeply.
- The minimum PSR observed over all shift tests is approximately 5.7, occurring around a shift magnitude of ≈ 301 px. The PSR decreases from about 180 at zero shift to roughly 5.7 at the largest shift. Thus, large drifts are accompanied by clearly "bad" confidence in the sense of our stabiliser logic (section 4.5.2): they lie well below the operational PSR level of PSR ≈ 65 for SD images used by our controller and below classical strong-confidence criteria used for correlation filters [15].

This means that although very large shifts cannot be accurately recovered up to the shifts in the requirements, the correlation peak remains strong and unambiguous even when the estimate fails. The failure mechanism is geometric: when the overlap becomes too small, the true shift becomes unrecoverable.



(a) Localisation error.

(b) PSR.

Figure 5.1: Effect of shift magnitude on stabiliser performance.

Growth of localisation error at large shifts An important observation is that the localisation error increases almost monotonically with shift magnitude once the displacement becomes large. This behaviour is consistent with the overlap-dependent failure mode of phase correlation [11, 21, 14]. As the true shift $|\Delta|$ increases, the area of overlap between the reference frame and the shifted frame decreases proportionally to $(W - |\Delta|)$ in the horizontal direction (and analogously in the vertical direction). When the overlap becomes small, the cross-power spectrum becomes dominated by whatever structure remains in the reduced common region rather than by the global image content. Consequently, the recovered peak drifts away from the true displacement in a manner that scales with the lost overlap area, producing the behaviour observed experimentally. Importantly, the PSR can remain relatively high in this regime because the peak can remain sharp even when it corresponds to an incorrect shift, a behaviour well documented in correlation-filter analysis [15, 14]. In our image stabilisation system, this is mitigated by the stricter confidence threshold (PSR ≈ 65). To add to this, in practice, such large drifts are already treated differently and generally already trigger a beam shift to avoid losing the area of interest.

5.1.2. Results: Sensitivity to Blur

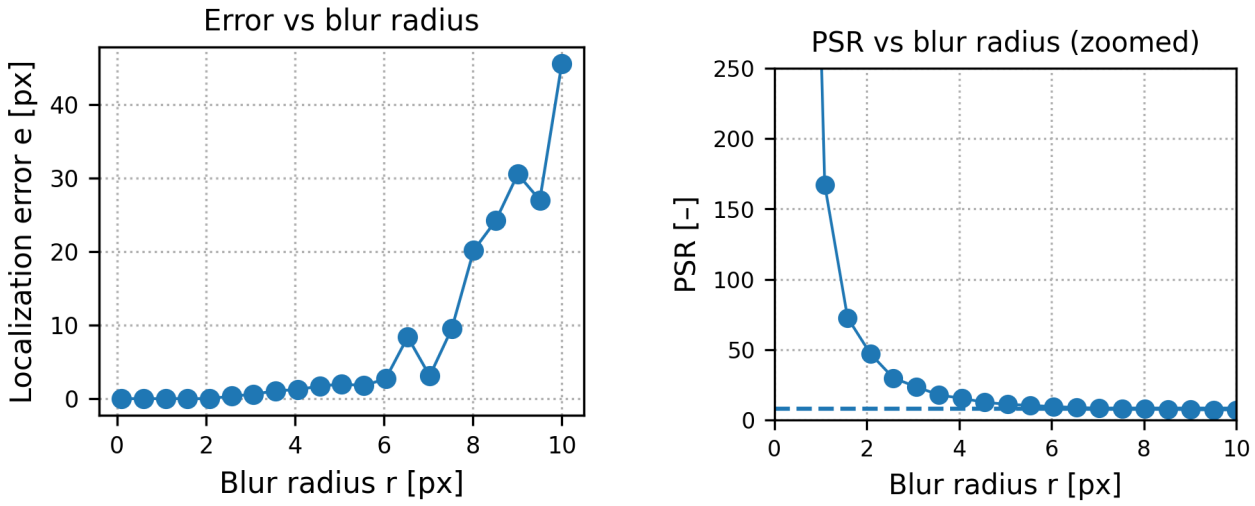
The blur sweep reveals a clear degradation threshold (Figure 5.2a):

$$\min(\text{PSR}) \approx 6.2 \text{ at } r = 10 \text{ px}.$$

From the CSV data, we find:

- For very small blur ($r \lesssim 1.1$ px) the PSR is high ($\gtrsim 100$) and the localisation error is exactly zero for all directions.
- For $1.1 \lesssim r \lesssim 3$ px, the PSR decreases rapidly from roughly 70 to around 24 and thus already enters the low-confidence regime relative to our operational threshold of PSR ≈ 65. Nevertheless, the mean localisation error in this range remains below about 1 px. The threshold is chosen in a such way that the system has enough headroom to meet our set requirements.
- For $3 \lesssim r \lesssim 6$ px, the PSR drops further into the range 18–10, and the mean localisation error increases to between roughly 1 and 3 px, with worst-case errors up to about 5 px.
- For stronger blur, in particular for $r \gtrsim 6.5$ px, occasional large errors appear: for instance at $r \approx 6.5$ px the worst-case error is ≈ 88 px, and at $r = 10$ px the mean error is ≈ 46 px with a maximum of ≈ 112 px. The minimum PSR over all blur tests is ≈ 6.2 at $r = 10$ px.

Thus, blur is the single most influential degradation factor affecting stabiliser reliability. From the perspective of the real-time stabilisation algorithm, blur radii already beyond $r \approx 1.5$ px lead to PSR values below the desired PSR ≈ 65 level, even though the localisation error remains small until about $r \approx 3$–4 px. For $r \gtrsim 6$ px, both PSR and localisation error indicate that the registration is becoming unreliable. This matches the theoretical



(a) Localisation error.

(b) PSR (our confidence metric).

Figure 5.2: Effect of blur magnitude on stabiliser performance.

expectation that Gaussian blur removes high-frequency content that is crucial for a sharp correlation impulse [25, 26, 20]. This means that our PSR confidence metric is a reliable tool for checking the focus of the SEM. This could prove useful for future developments. This also highlights the importance of an image that is well-focused. If the user does not focus the SEM well, the video output will be less stable than the video output would be if the SEM were well-focused.

5.1.3. Results: Sensitivity to Noise

Gaussian and Poisson noise tests (Figure 5.3) show that the stabiliser is very robust to both additive and shot-noise-dominated imaging conditions.

For Gaussian noise, the PSR decreases smoothly with increasing noise level. Over the full range $\sigma \in [0, 100]$ gray levels we observe

$$\min(\mathrm{PSR}_{\mathrm{Gaussian}}) \approx 19.5 \tag{5.1}$$

(at $\sigma = 100$), while the median PSR remains well above 50 even at the highest noise levels. For typical SEM noise levels ($\sigma \lesssim 40$), the minimum PSR over all runs is about 52, and throughout the entire Gaussian noise swee,p the localisation error remains identically zero.

For Poisson noise, the scale parameter s controls the mean counts per pixel. Small s corresponds to strongly noise-dominated images, while larger s increases the signal-to-noise ratio. Across the range $s \in [0.01, 1.43]$ we find

$$\min(\mathrm{PSR}_{\mathrm{Poisson}}) \approx 24.6 \tag{5.2}$$

at the lowest scale $s \approx 0.01$, increasing to median values of several hundred as s grows. Again, the localisation error is exactly zero for all tested Poisson noise levels in this dataset.

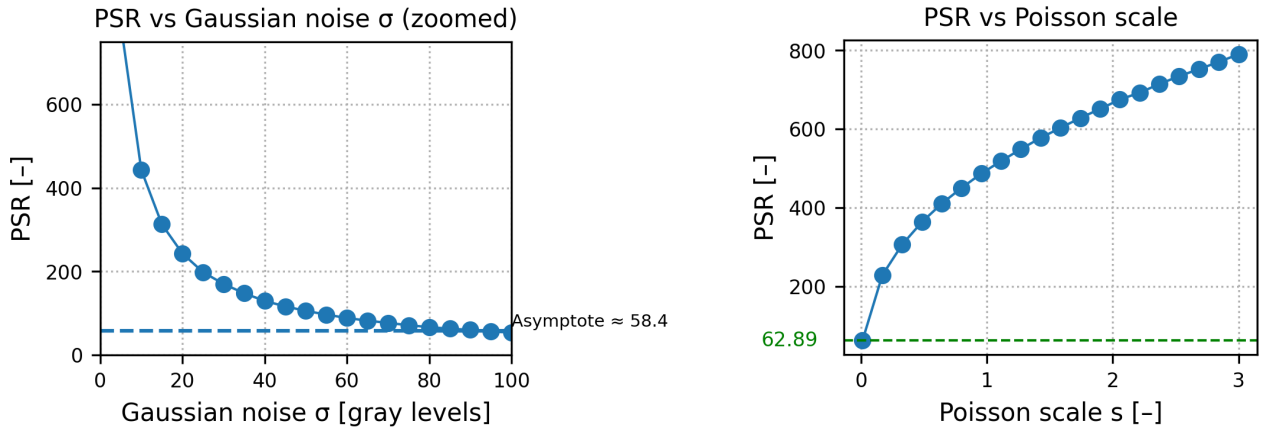
Conclusion Both Gaussian and Poisson noise over the tested ranges do not meaningfully degrade the performance of the stabiliser. The correlation peak remains strong even in the most noise-dominated regimes, and shifts are always estimated correctly in the synthetic tests. This is expected because phase correlation normalises the magnitude of the Fourier spectrum, making it inherently resilient to statistically independent noise sources [13, 21, 14, 25].

5.1.4. Results: Sensitivity to Structural Damage

Structural damage up to the largest tested fractions does not degrade the stabiliser in any practically significant way. Over all damage fractions in the CSV in appendix B.2.2 we find

$$\min(\mathrm{PSR}_{\mathrm{damage}}) \approx 30 \tag{5.3}$$

with typical PSR values in the range of $\mathcal{O}(10^2)$ for moderate damage. Even for the most serious simulated damage, the PSR remains comfortably above classical strong-confidence thresholds, and throughout the damage



(a) PSR vs Gaussian noise σ.

(b) PSR vs Poisson scale s.

Figure 5.3: PSR under Gaussian (left) and Poisson (right) noise. In both cases, the PSR remains well above classical strong-confidence thresholds and no mis-registrations occur [15].

sweep, no misregistrations were observed: the estimated shifts coincide with the ground truth, consistent with the robustness of Fourier-based methods for local intensity changes when sufficient structure remains unchanged [14, 27]. This means that requirement [C.3] is fulfilled. Although it does not specify how large the fluctuations can be, from the example image in figure B.5, it is clear that if such a bad degradation is still recoverable, a high amount of intensity fluctuations can, in general, be recovered.

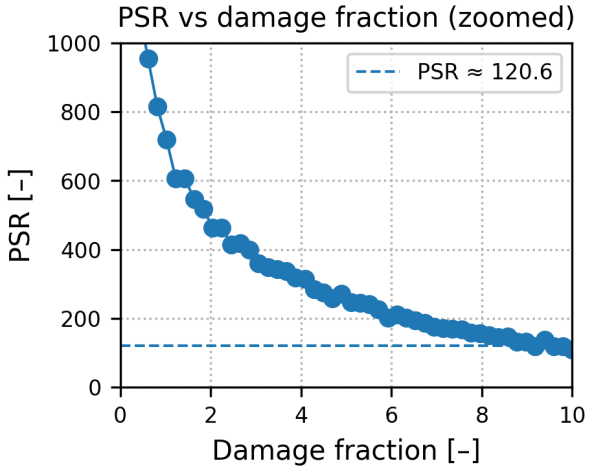


Figure 5.4: PSR as a function of structural damage fraction. PSR remains high across the full range of damage levels, and the localisation error is negligible.

5.2. Fixed-Shift Degradation Experiments

The experiments in section 5.1 varied the amount of drift and the strength of individual degradations independently. In practice, however, the stabiliser is used at a fixed position while imaging conditions vary (for example, due to focus changes, drift or specimen damage). This mirrors common evaluation protocols in video stabilisation and registration [2, 14, 39]. To mimic this scenario more closely, a separate evaluation was carried out in which the *true* translational shift is kept fixed, and only the image degradation is varied.

In this degradation-shift experiment, a set of five representative SEM reference frames of size 712 px×484 px is used. For each reference, a synthetically degraded version is created by applying a fixed translation of

$$(\Delta y_{\text{true}}, \Delta x_{\text{true}}) = (-100, -100)\,\text{px}, \tag{5.4}$$

followed by one of four degradation types: Gaussian blur, additive Gaussian noise, Poisson noise, or structural damage. For every combination of reference frame and degradation parameter, the phase-correlation stabiliser

estimates the shift $(\Delta y_{\text{est}}, \Delta x_{\text{est}})$, and we record

- the localisation error $e = \sqrt{(\Delta y_{\text{est}} - \Delta y_{\text{true}})^2 + (\Delta x_{\text{est}} - \Delta x_{\text{true}})^2}$,
- the peak-to-sidelobe ratio (PSR), and
- additional peak shape metrics (PAR, P2SR, sharpness) for diagnostic use, in line with standard correlation-filter evaluation practice [15].

The complete results are stored in `degradation_shift_results.csv` found in Appendix B.2.1.

5.2.1. Experimental Design and Parameter Ranges

For each of the five reference frames and for each degradation type, the following parameter ranges were explored:

- Gaussian blur: isotropic Gaussian smoothing with radius $r \in [0, 10]$ px in $\Delta r = 0.5$ px steps (21 blur levels, 105 frames in total).
- Additive Gaussian noise: zero-mean Gaussian noise with standard deviation $\sigma \in [0, 100]$ gray levels in $\Delta\sigma = 5$ steps (21 noise levels, 105 frames).
- Poisson noise: Poisson-distributed noise with scale parameter $s \in [10^{-4}, 1.5]$ (25 levels, 125 frames). Small values of s correspond to extremely low-count, heavily noise-dominated images.
- Structural damage: synthetic damage controlled by a parameter $d \in [0, 10]$ (16 levels, 80 frames), where larger values of d correspond to more severe and spatially extended damage patterns.

  Images can be found in Appendix B.3.

In all cases, the ground-truth shift is identical, so any variation in localisation error can be attributed directly to the degradation strength.

5.2.2. Gaussian Blur

Blur has the most pronounced impact on registration accuracy as seen in Subsection 5.1.2. Figure 5.5 plots the mean localisation error as a function of the Gaussian blur radius r; the curve approaches a horizontal asymptote of approximately 130 px, indicating that the stabiliser effectively loses all useful shift information for a very strong blur.

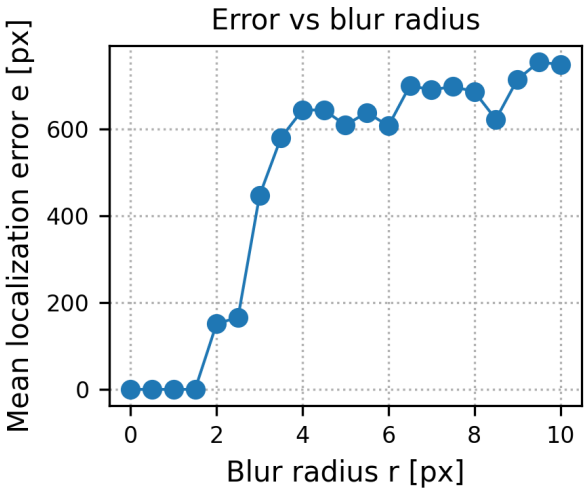


Figure 5.5: Fixed-shift degradation experiment with Gaussian blur: mean localisation error e as a function of blur radius r. The dashed horizontal line indicates the approximate asymptotic error reached for strong blur.

For small blur radii, the stabiliser is essentially unaffected:

- For $0 \leq r \leq 3.0$ px, the mean error is exactly 0 px, and the worst-case error across all references remains 0 px; the PSR decreases from roughly 944 at $r = 0$ to about 53 at $r = 3.0$, but the correlation peak is still very sharp.
- For moderate blur, $3.5 \leq r \leq 6.0$ px, the mean error remains small, between about 0.6 px and 1.8 px, with worst-case errors not exceeding 4 px. Over the same range, the minimum PSR drops from roughly 38 to 15.

As blur becomes stronger, phase correlation gradually breaks down:

- For $r \approx 6.5$-8.0 px, the mean error increases to approximately 2-3 px with occasional outliers of 5 px. The minimum PSR in this range is already close to 10, well below the operational confidence level of PSR ≈ 60 used in the stabiliser logic.
- At $r = 8.5$ px the degradation becomes severe: the mean localisation error jumps to about 13.7 px, and the worst-case error reaches approximately 50.4 px, with a minimum PSR of only ≈ 9.1.
- At the strongest tested blur, $r = 10$ px, the mean error further increases to ≈ 28.7 px and the worst-case error reaches ≈ 124.3 px. The minimum PSR across references is then approximately 7.0.

These results confirm that Gaussian blur acts as a true failure mode for the stabiliser. The method remains highly accurate for mild and moderate blur, but eventually loses enough high-frequency content that the correlation peak becomes broad and ambiguous, as predicted by Fourier-domain analysis of convolution [26, 25, 20]. Importantly, the PSR falls well below the operational threshold precisely in the regime where localisation errors become large, meaning that the stabiliser would correctly classify these cases as low confidence and trigger a beam shift.

5.2.3. Additive Gaussian Noise

For additive Gaussian noise, the behaviour is markedly different. Across the entire tested range $\sigma \in [0, 100]$ gray levels, the stabiliser recovers the shift exactly:

- The mean localisation error is 0 px for all noise levels.
- The worst-case localisation error over all references and σ values is also 0 px.

Figure 5.6a illustrates this flat error curve.

The only effect of increasing σ is a gradual and smooth decay in PSR:

- At $\sigma = 0$ the minimum PSR across references is again ≈ 923, matching the undegraded case.
- At the highest tested noise, $\sigma = 100$, the minimum PSR is still ≈ 60.0, i.e. right at the PSR level adopted as a "bad confidence" threshold in the live system.

In other words, even for very strong Gaussian noise, the phase-correlation stabiliser continues to deliver exact shift estimates, and the correlation peak remains sufficiently sharp that PSR only just enters the low-confidence regime at the very edge of the tested range [13, 21, 14].

5.2.4. Poisson Noise

The Poisson noise experiment explores a wide range of scale parameters $s \in [10^{-4}, 1.5]$. For most of this range, the stabiliser is again extremely robust:

- For all scales $s \geq 0.0626$, the mean localisation error is 0 px, and the worst-case error across all references remains 0 px.
- In this regime, the minimum PSR increases from roughly 208 at $s \approx 0.06$ to about 692 at the largest tested scale $s = 1.5$, reflecting the improved signal-to-noise ratio as the effective exposure increases.

Only at the most extreme, almost unphysical low-count setting, $s = 10^{-4}$, does the method begin to fail:

- At $s = 10^{-4}$ the mean localisation error rises to ≈ 22.8 px and the worst-case error reaches ≈ 113.8 px.
- The corresponding minimum PSR across references is ≈ 7.7, with an average PSR of only ≈ 10.4.

The error behaviour is visualised in Fig. 5.6b. At this extreme point, the image is essentially dominated by pure shot noise, so there is no longer a coherent phase structure to correlate. The behaviour is therefore consistent with the theoretical limitations of phase correlation: when the signal becomes indistinguishable from noise, both PSR and localisation accuracy collapse [25, 20, 14]. For all practically relevant Poisson noise levels, however, the stabiliser remains reliable, and PSR stays far above the ≈ 60 operational threshold.

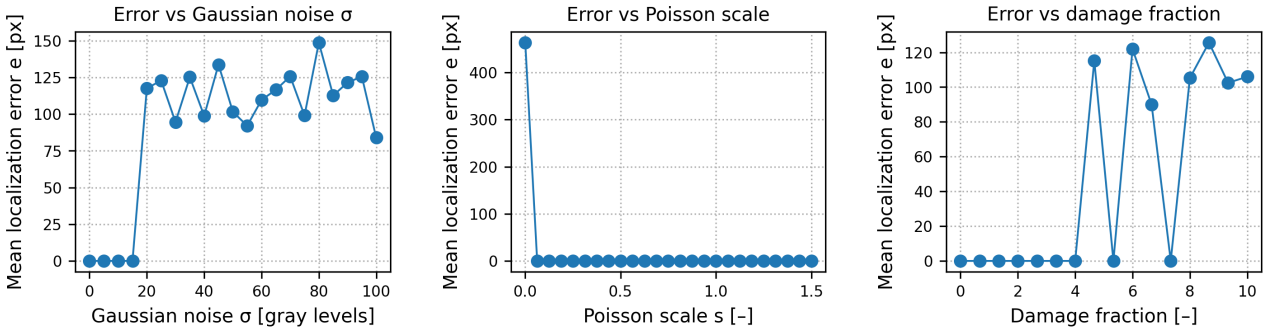
5.2.5. Structural Damage

The `damage` experiments apply progressively more serious synthetic damage patterns to the shifted images. Despite substantial intensity and contrast changes, the stabiliser remains remarkably robust:

- Across the entire damage parameter range $d \in [0, 10]$, the mean localisation error is 0 px and the worst-case error is also 0 px.

- The minimum PSR decreases from the undegraded value (≈ 923) to approximately 81 at the strongest damage, but remains well above the PSR ≈ 60 operational threshold. Typical PSR values lie even higher, with the overall mean PSR around 412.

Figure 5.6c shows that the localisation error remains identically zero over the entire damage range. Even when a large fraction of the frame is damaged, sufficient structure remains unchanged for phase correlation to lock onto [14, 27]. The correlation peak is therefore preserved, and the shift estimate remains accurate.



(a) Gaussian noise **(b)** Poisson noise **(c)** Structural damage

Figure 5.6: Fixed-shift degradation experiments: mean localisation error e as a function of (a) Gaussian noise level σ, (b) Poisson scale s, and (c) damage fraction d. In all three cases, the error remains essentially zero for all practically relevant parameter values.

5.2.6. Summary and Comparison of Degradation Types

Table 5.1 summarises the main findings of the fixed-shift degradation experiment across all four degradation types. Gaussian blur and extreme, almost signal-free Poisson noise are the only conditions that produce substantial localisation errors, and in both cases the PSR drops well into the "bad confidence" regime. Additive Gaussian noise and realistic Poisson and damage levels, by contrast, leave the stabiliser essentially unaffected (Figs. 5.5 and 5.6), in agreement with theoretical expectations for phase-correlation registration [13, 21, 14].

Table 5.1: Summary of the fixed-shift degradation experiment (`degradation_shift_results.csv`). For each degradation type, the table reports the parameter range, the worst-case mean and maximum localisation error over all reference images, and the minimum PSR observed over the entire sweep.

Degradation type	**Parameter range**	**Max mean error** [px]	**Max error** [px]	**Min PSR**
Gaussian blur	$r \in [0, 10]$ px	≈ 28.7	≈ 124.3	≈ 7.0
Additive Gaussian noise	$\sigma \in [0, 100]$ gray levels	0	0	≈ 59.9
Poisson noise	$s \in [10^{-4}, 1.5]$	≈ 22.8 (at $s = 10^{-4}$)	≈ 113.8	≈ 7.7
Structural damage	$d \in [0, 10]$	0	0	≈ 81.0

From the perspective of the image stabilisation software, these results reinforce the conclusions drawn earlier:

- **Blur is the dominant practical failure mode.** Once the blur radius exceeds roughly 6-8 px, both PSR and localisation accuracy deteriorate rapidly, and the PSR-based confidence logic correctly classifies these frames as unreliable.
- **Noise is largely harmless in the SEM regime.** Even very strong additive Gaussian noise and realistic Poisson noise levels do not produce misregistrations. Only at the most extreme, almost signal-free Poisson setting does the method fail, as expected.
- **Structural damage is handled gracefully.** Because a substantial portion of the frame typically remains intact, phase correlation can still lock onto the unchanged region, maintaining both high PSR and zero localisation error [14, 27].

The mathematical basis behind these findings is derived in Appendix E.2

5.3. Summary of Operating Limits

Based on the experimental CSV data (appendix B.2), we derive practical limits for reliable operation:

- **Pure translations.** The localisation error remains essentially zero up to a shift magnitude of $|\Delta| \approx 113$ px. Beyond this point, the mean error grows rapidly with shift magnitude, as can be seen in section 5.1.1. There, it can also be seen that large drifts are classified as "bad confidence" and would trigger a reference update.
- **Maximum tolerable blur.** Small blur radii produce excellent localisation performance with very high PSR. Moderate blur reduces PSR steadily, but localisation error remains relatively small. For larger blur levels, both PSR and localisation accuracy deteriorate sharply, leading to errors. This can be seen in section 5.1.2.
- **Noise tolerance.** Phase correlation is highly robust to both Gaussian and Poisson noise. Even with strong Gaussian noise or typical Poisson noise levels, the PSR remains high, and localisation error stays at zero. Only extreme Poisson noise causes noticeable error. Overall, the method maintains accurate localisation and strong PSR across all realistic noise conditions. As was shown in section 5.1.3.
- **Structural damage tolerance.** In both datasets (main and fixed-shift), no failures occur across the entire tested range of damage parameters. The PSR remains well above classical strong-confidence thresholds [15], and the estimated shifts remain accurate throughout, as can be seen in section 5.1.4.

In the image stabilisation software we use PSR not only as a diagnostic but also as a control signal: once the PSR falls below the values set in equation 4.1, the reference frame is updated to one of the other received images, even though theoretical strong-confidence thresholds from the literature are substantially lower (PSR ≈ 8) [15].

Overall, the phase-correlation stabiliser is highly robust to noise and structural changes, while blur and large drift define the practical performance limits relevant to real-time SEM acquisition [14]. Crucially, the PSR-based confidence logic ensures that conditions corresponding to unreliable registration are correctly classed as "bad confidence" and handled via beam shifts and reference updates in the image stabilisation software.

Table 5.2: Summary of stabiliser robustness under different degradation types, based on the combined main and fixed-shift sweeps. Only blur causes a meaningful reduction in correlation peak sharpness or localisation accuracy; all other degradations remain well within safe operational limits for realistic parameter ranges.

Degradation type	Range tested	PSR behaviour	Localisation error	Conclusion / Failure mechanism
Gaussian blur	$0.1 \leq r \leq 10$ px	Strong monotonic decrease; from $\sim 10^3$ at small r to ~ 7 at $r = 10$; PSR drops below ≈ 60 already around $r \approx 1.5$-2 px	Error ≈ 0 px for $r \lesssim 3$ px; grows to a few pixels for $3 \lesssim r \lesssim 6$ px; for $r \gtrsim 6$ px occasional large errors (tens of pixels, up to $\sim 10^2$ px)	**Only degradation that significantly harms our stabilisation algorithm**; blur suppresses high-frequency edges that generate the sharp correlation impulse [26, 25, 20].
Gaussian noise	$0 \leq \sigma \leq 100$ gray levels	PSR decreases smoothly from very large values at low noise to $\sim$ 50-80 at the highest σ; minimum ≈ 19.5	Error 0 px over entire range (both main and fixed-shift sweeps)	Noise perturbs magnitudes more than phases; CPS normalisation removes most impact, so the peak remains correctly localised [13, 21, 14].
Poisson noise	Sweep: $10^{-4} \leq s \leq 1.5$	In the main sweep, PSR $\gtrsim 25$ at smallest. In the fixed-shift sweep PSR ≈ 10 at $s \approx 10^{-4}$ and rises rapidly above ~ 200 for realistic s	Error 0 px in main sweep; in fixed-shift sweep a mean error of ~ 23 px occurs only at extreme $s \approx 10^{-4}$, otherwise 0 px	Shot noise behaves multiplicatively; for realistic count levels, the phase structure is preserved, and the stabiliser remains reliable [20, 14]. Only unrealistically low-count Poisson noise produces noticeable errors.
Structural damage	Wide range of internal damage parameters (up to near-complete corruption of large regions)	PSR remains between ~ 30 and $\sim 10^3$; decreases with damage but stays well above strong-confidence thresholds for realistic damage levels.	Error $\approx$ 0 px for all tested damage levels (both datasets)	Large intact regions preserve phase information; correlation peak remains sharp and correctly located [14, 27].
Pure shift	$\lvert\Delta\rvert$ = 0-377 px	PSR decreases from ~ 180 at zero shift to ~ 11 at the largest shift; minimum ≈ 5.7 around $\lvert\Delta\rvert \approx 300$ px	Error 0 px up to $\lvert\Delta\rvert \approx 113$ px; beyond that mean error grows rapidly, reaching ~ 260 px at $\lvert\Delta\rvert = 301$ px and ~ 540 px at $\lvert\Delta\rvert = 377$ px; worst-case ~ 712 px	Failure due to decreasing image overlap, not correlation weakness; once overlap becomes too small, the estimator returns high-confidence but geometrically incorrect shifts [11, 21, 14].

In conclusion, the combined benchmarks confirm the theoretical robustness of phase correlation against noise and structural change [13, 21, 14], while highlighting blur and excessive drift as the primary practical limitations of FFT-based SEM stabilisation. The PSR-based confidence logic is well aligned with the underlying failure modes observed in the synthetic tests and with recommendations from the correlation-filter literature [15]. Because the requirements have to hold for the worst case, the focus in this analysis was on SD image,s which correspond to the worst case results, as HD images contain more relevant pixel data. In Appendix E.3, the HD case is examined.

Interpretation and relevance to real SEM operation The experiments in this chapter are *simulation-style* benchmarks: each test frame is created by applying a single, isolated synthetic transformation (shift, blur, noise, or damage) to a reference image. In real SEM acquisition, consecutive frames typically differ by a *combination* of effects at once (small drift, shot noise, minor contrast changes, occasional focus variation, etc.), so the quantitative curves in Sections 5.1.1–5.1.4 should not be interpreted as exact predictions of field performance. Their purpose is instead comparative: they isolate the main degradation mechanisms and show which ones dominate the failure modes of phase correlation in our setting.

Among these benchmarks, the translational-drift sweeps (and especially the fixed-shift degradation experiment in Section 5.2) are the most informative for system design, because drift estimation is the stabilisers primary task and because other real-world variations appear on top of that drift. Importantly, the tested drift magnitudes were chosen to be *much larger* than the frame-to-frame drift expected during normal SEM operation. This deliberately creates a worst-case stress test: if the stabiliser remains accurate and the confidence logic reliably flags failure at these extreme displacements, then it has ample headroom when applied to the substantially smaller inter-frame drifts encountered in practice (the drift velocity can be found in Appendix D). Consequently, even though real frames differ in more ways than our synthetic transforms, the requirements set by (Requirements [B.6] and [B.7]) are taken much more strictly than what would be expected drift-wise between frames in real-world acquisition D. This implies that the system is expected to remain reliable under real operating conditions (provided the SEM is focused), even if this means that we might not technically meet requirements [B.6] and [B.7] in practice.

5.4. Evaluation of Beam-Shift and Software Correction Ranges

To provide an intuitive picture of how the software stabiliser and the hardware beam shift interact, we overlay the corresponding regions directly on SEM images during evaluation of the stabiliser. For each frame, we draw:

- a **blue** rectangle that represents the maximum usable hardware beam shift range projected into the image plane (corresponding to ± 20 μm in both the x- and y-directions, see assumption [1.1]), and
- an **orange** rectangle centred on the image that represents the software-only drift tolerance region determined by the threshold parameter τ.

The pixel inset corresponding to the nominal ± 20 μm beam shift range is computed from the calibrated pixel size p_{corr} as

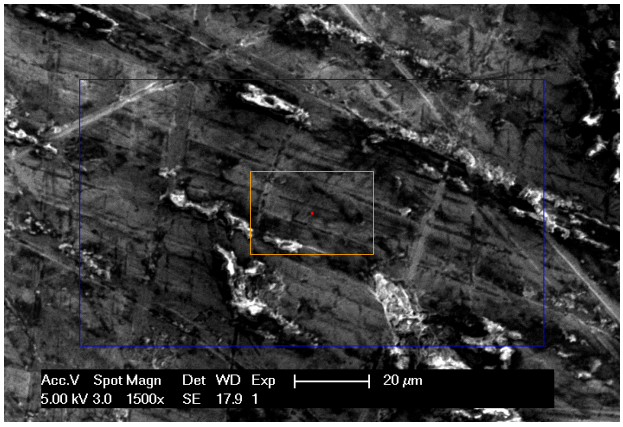
$$m_{\text{px}} = \frac{0.020 \text{ mm}}{p_{\text{corr}}}. \tag{5.5}$$

If $2\,m_{\text{px}}$ is smaller than both the image width and height, the blue beam shift rectangle appears as an inset within the frame. This occurs for the 1500× case: using $p_{\text{corr}} = 1.1494 \times 10^{-4}$ mm/px yields $m_{\text{px}} \approx 174$ px, and therefore a total inset width of $2m_{\text{px}} \approx 348$ px fits comfortably inside the 712×484 image.

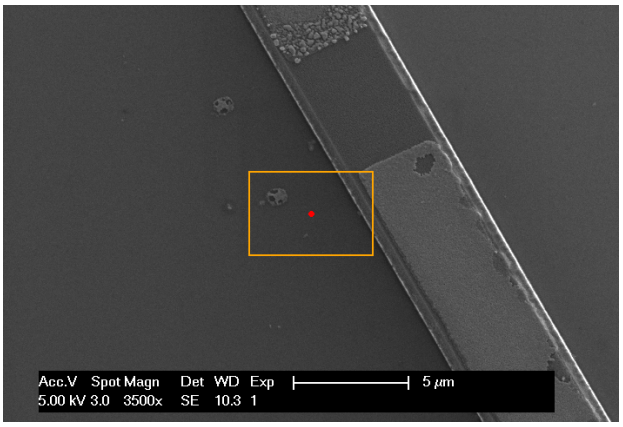
If $2\,m_{\text{px}}$ exceeds either the width or height, the available physical beam shift range is larger than the images field of view. In this situation, the entire frame lies within the hardware-correctable region, and we visualise this by clamping the blue rectangle to the image borders. This occurs for the 3500× case, where $p_{\text{corr}} = 3.7594 \times 10^{-5}$ mm/px gives $m_{\text{px}} \approx 532$ px, which is larger than both half-dimensions of the 712×484 image. In this regime, a single physical beam shift could, in principle, reposition the field of view beyond any point shown in the image.

Figures 5.7a and 5.7b illustrate these two situations. In both images, the orange rectangle (software correction region) is strictly contained within the blue hardware correction region, ensuring that a physical beam shift can always be issued when the software threshold is exceeded.

Both examples demonstrate that the system satisfies all relevant drift-correction requirements. The hardware beam shift range is correctly represented as a rectangular region corresponding to ± 20 μm in each axis, in accordance with assumption [1.1]. For the 1500× case, this beam shift range fits within the field of view, showing that the system can physically correct drifts up to the limits required in [B.4] and [B.5]. At 3500×, the hardware correction region is larger than the entire frame, providing even greater headroom and ensuring that all software-detected drifts remain physically correctable.



(a) 1500× image with beam shift range (blue), software threshold region (orange) and image centre (red). The ±20 μm beam shift range fits inside the field of view and therefore appears as a visible inset rectangle.



(b) 3500× image with beam shift range (blue), software threshold region (orange) and image centre (red). Here, the ±20 μm beam shift range is larger than the field of view, so the blue rectangle is not drawn.

Figure 5.7: Visualisation of the stabiliser geometry at two magnifications. The blue rectangle shows the projection of the hardware beam shift limits, while the orange rectangle shows the software drift tolerance region defined by the threshold parameter τ.

In both magnification regimes, the software threshold region (orange) lies strictly inside the physical correction region (blue), demonstrating that the system can always issue a physical correction when the drift exceeds the software limit, fulfilling requirement [A.6]. This also confirms compliance with the trade-off requirement [4.6], which states that drift must be correctable both digitally and physically.

5.4.1. Beam-Shift Sufficiency as a Function of Magnification

In the visualisation of Fig. 5.7, we use the 1500× image as a conservative reference case. For this medium-quality frame, the calibrated pixel size is

$$p_{\text{corr}} = 0.00011364 \text{ mm/px}, \tag{5.6}$$

and the vertical resolution is $N_y = 968$ pixels. The corresponding vertical field of view is

$$\text{FOV}_y(1500\times) = N_y\, p_{\text{corr}} \approx 0.1100 \text{ mm} = 110.0\ \mu\text{m}. \tag{5.7}$$

A beam shift of magnitude $B = 20\ \mu$m covers the fraction

$$f(1500\times) = \frac{B}{\text{FOV}_y/2} = \frac{2B}{\text{FOV}_y} \approx \frac{40}{110.0} \approx 0.36 \tag{5.8}$$

of the half-field of view along the vertical direction. In other words, at 1500× the available beam shift corresponds to roughly a 36% movement of the image centre from the middle towards the top or bottom border.

In our experiments, we typically use threshold values in the range $\tau \approx 0.05$–0.25. For this range, d_y^{th} corresponds to at most about 25% of the half-field of view, so the hardware beam shift has a comfortable margin to correct such displacements in a single step. This directly supports the requirements on maximum residual shift in the output video (requirements [3.5] and [3.6]) and on the correctable physical shift (requirements [B.4] and [B.5]).

Because the physical field of view $\text{FOV}_y(M)$ scales approximately inversely with magnification M,

$$\text{FOV}_y(M) \propto \frac{1}{M}, \tag{5.9}$$

the fraction of the half-field that can be covered by a fixed beam shift increases linearly with M,

$$f(M) = \frac{2B}{\text{FOV}_y(M)} \propto M. \tag{5.10}$$

Thus, 1500× is effectively a worst-case scenario: for higher magnifications the same ±20 μm range covers a larger and larger fraction of the field of view. Using the calibrated pixel sizes for our test images, we obtain, for example:

- at 1500×, $f \approx 0.36$ (about 36% of the half-FOV),
- at 3500×, $f \approx 0.81$ (the beam shift covers more than 80% of the half-FOV),
- at 8000×, the beam shift range is larger than the entire field of view and easily exceeds any realistic software threshold used in the system.

These results show that the available beam shift range is sufficient to support the image processing requirements across the magnification range of interest. In the case that the user wishes to capture images at an even lower magnification level than 1500×, the output video will be stabilised by just the software. While this technically does not meet requirement [4.6], it does result in a stable output video that meets requirements [3.6], [3.5]. This is because, as the magnification level is decreased, the shift is much less noticeable and therefore easily corrected by the software drift correction algorithm.

5.4.2. Graphical Representation of Threshold Effects

The dependence of the software and hardware capacities on the threshold τ can be visualised with simple parametric plots. For illustration, we consider the same 1500× high-definition configuration that is used in section 5.4.1, with image width $W = 1424$ pixels, half-width $x_{max} = 712$ pixels, calibrated pixel size $p_{\text{corr}} = 0.00011364$ mm/px, and a one-sided beam shift range $B_x = 0.02$ mm.

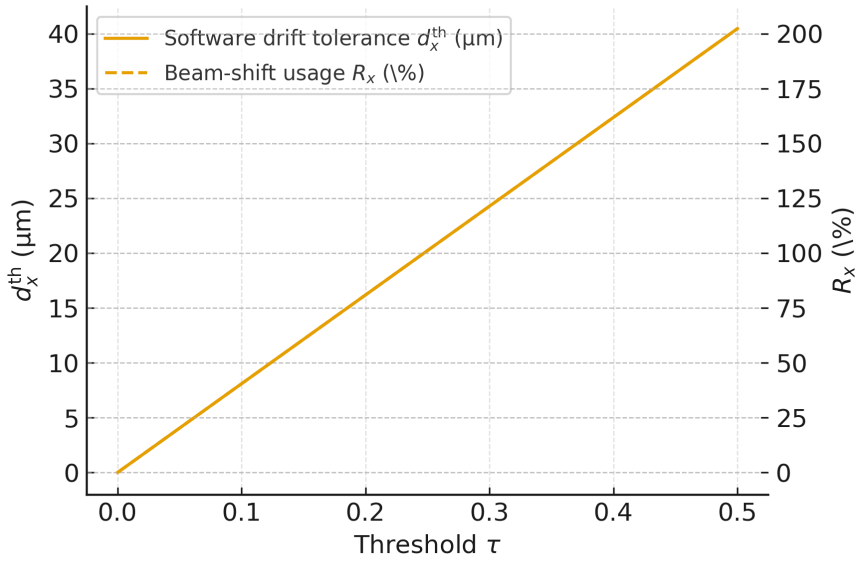


Figure 5.8: Software drift tolerance d_x^{th} and beam shift usage R_x as functions of the threshold τ for the 1500× HD configuration ($W = 1424$ px, $p_{\text{corr}} = 0.00011364$ mm/px, $B_x = 0.02$ mm).

Figure 5.8 shows d_x^{th} (in μm) and R_x (in %) as functions of the threshold τ for this configuration. Such plots make it easy to select operating points where:

- the software correction is large enough to keep the region of interest stable in the output video (requirements [3.5] and [3.6]), and
- the beam shift usage R_x and R_y stay comfortably below 100%, ensuring that physical corrections can still be applied when needed and that the correctable physical shift limits in requirements [B.4] and [B.5] are not exceeded.

Because the pixel size depends directly on the magnification and thus on the field of view (see equation 4.4) and 4.3, the relationship between the threshold τ, the software drift tolerance, and the beam shift usage cannot be predetermined. For this reason, the plot in Fig. 5.8 must be generated after the user acquires the first manual image, allowing the system to compute the actual pixel size and determine the optimal operating point for drift correction. In theory, this procedure would make it possible to guarantee that the system meets trade-off requirement [4.6]. In practice, however, when working at lower magnification levels, the available field of view is sufficiently large that beam shifts are not required to maintain a stable output video. Thus, while the system must support both software-based and hardware-based drift correction to handle large displacements at high magnifications, fulfilling requirement [4.6], it is not necessary for the user to obtain a stable result at lower magnifications. This fact is not surprising, as the measured drift velocity in appendix D is 0.629 μm, which is much smaller in scale than the global scale of images that do not use high magnification. However,

for long experiments at lower magnification levels, the user might want to look at a plot like 5.8 to select a proper threshold value. In the case that no beam shifts are performed at all, the required digital alignment can accumulate over time, increasing the amount of outpainting needed at the image borders and producing a less pleasant viewing experience in the output video.

6

Discussion and Conclusion

6.1. Conclusion

The primary goal of this thesis, building a module to perform image stabilisation on a dataset of images taken with the SEM, and producing a stable video output, was largely met using a modular and empirically tested design.

The design is able to run concurrently with the control software to issue live feedback for stabilisation control. It is also able to run in its stand-alone mode to be able to redo the video generation process. Possibly with different settings.

The core of the image stabilisation module, the phase correlation algorithm, has shown its robust performance in various tests. Once the values in the stabilisation vector cross over a user-specified threshold, the module is able to calculate with high precision the exact millimetre value where the beam should be shifted to realign the sample to the middle of the screen again (see section 4.7.4). When the pixel shift is less than or equal to 113 pixels, SD resolution testing data shows that the shift is always perfectly recoverable. When the shift errors stay below 131 pixels, the error in the shift calculation is approximately equal to the error limit set by requirement [B.6]. For HD images, there is zero error up until a maximum shift of 400 pixels. However, after that, the error quickly grows bigger than the limit in requirement [B.7]. As a fraction of the total image resolution (either height or width), it can be seen that the phase correlation algorithm can handle larger relative shifts for HD images. When using the image stabilisation software, taking pictures in HD mode is thus recommended.

To check whether the stabilisation algorithm is stable at all times, a PSR-based confidence metric was used and verified. In section 4.7.4, it can be seen that this metric is very robust for different types of noise, as long as they are not present in very unreasonable amounts. However, the reliability of this metric is very dependent on how well the user has focused the image initially, as the combination of this measure and the phase correlation means that our confidence is very dependent on high-frequency signals being present in the images. When the image stabilisation software is used in practice, users should thus take extra care to focus the image very well.

At the end of a capture sequence, or after an existing dataset is fully processed, a video is returned. These videos show that the centre of the images stays nicely in the middle of the frame.

6.2. Discussion and Recommendations

Although the primary goal of this thesis has largely been met. There are still some things to consider in the context of this system.

6.2.1. Dataset biases

The product of this thesis, the stabilisation software, results in a wall-stabilised video. This has been verified by a lot of different pictures. Some of these pictures were given to us. Others were taken manually by us. However, all these pictures were taken of the same types of samples: microstructures. This means that there could be a bias in the software for this type of sample. Ideally, the performance of this stabiliser should also be checked on other, preferably more organic, samples such as human tissue. These inherently more three-dimensional shapes may prove more difficult to stabilise, as they are harder to focus well.

6.2.2. Repeating paterns

Since a key algorithm in the stabilisation software is the phase correlation, it is important to think about the limitations of this algorithm. As the name implies, this algorithm needs clear features in the signal's phase. This means that if the input pictures are of a repeating structure (i.e. a chessboard-like image), the phase correlation can return multiple strong peaks. Likely, the current confidence metric is not useful for this type of image. This results in unnecessary reference frame adjustments. Which may come with their own set of

problems (see subsection 6.2.6). One potential addition that could be made is to estimate the drift velocity based on a (sub)set of earlier measurements. Then the expected shift based on the drift velocity can also play a factor in deciding which strong peak to choose. This implementation does have the problem of needing to define a velocity vector reliably. This could be a problem for future work.

6.2.3. Databar

To maximise the available amount of data that the stabilisation software can use, the default databar on the image is not printed. This also has the added benefit that the output video becomes more visually pleasing to watch. However, for individual examination of these pictures, the lack of a databar may be problematic. This could be combated by adding another step in the preprocessing sequence. This step would make a copy of the framebuffer and would then only perform the `generate_annotations` function to add the custom databars. Then the image data of each frame should be stored as a separate image file on the computer.

6.2.4. Increasing maximum relative shift using higher resolutions

During the evaluation of the performance of the stabiliser, it was seen in section E.3 that the stabiliser could recover a larger shift if the image resolution is larger. This could support the argument that the stabilisation software may perform even better when the SEMs `XHD` mode (almost a 4k resolution) is used. However, the XHD feature of the microscope is not readily used, since it maxes out the Windows 2000 PCs' video memory, and is generally considered impractical. For these reasons, this resolution was not part of the scope of this thesis. However, extending the program to also work with this resolution might even result in even better stabilisation performance.

6.2.5. Applicability of the used PSR values

In equation 4.1, the PSR thresholds were determined. However, the testing methodology has one flaw that makes the calculated PSR probably a little on the high side. The affine transformations that were performed on the images to create the full dataset, do correctly apply a shift to these images. However, these images fail to add the data that would be there if the shifts were made on the real SEM. Furthermore, these shifts do not change the random noise that is in the images. Making the calculated PSR values some optimistic values. To account for this, one could consider lowering these factors in the code to 80% of their calculated values.

6.2.6. Reference frame updates

Requirement [B.9] sets the requirement that there should be no cumulative drift. As a primary strategy for this, the code always uses the very first picture that was taken as a reference frame. This way, no cumulative error can build up. However, if the confidence is low because the sample has changed too much, the stabilisation software will change its reference frame. For computing the stabilisation vector of a frame that is not compared with the first frame, first, the stabilisation vector from the new reference frame to the current frame should be calculated. Then the stabilisation vector of the reference frame to the first frame should be added on top of that. This step could potentially add cumulative drift. Especially given the fact that, theoretically, the code could do reference frame updates multiple times during execution.

A way to slightly combat this is to store the stabilisation vector not in integers (which is the logical choice, since the units of this vector are pixels), but in floats. This way, the cumulative error due to rounding of the output values of the stabilisation algorithm can be eliminated. This approach would, however, have an influence on all other functions that use the stabilisation vectors. As all of these will need to perform a rounding step first.

6.2.7. Stand-alone running

Currently, when the stabilisation software is running together with the control software, the user experience of the stabilisation software is relatively seamless. However, when stand-alone running is desired, there is no user interface, and a direct command line call to start the program is required (for instructions on how to run the stand-alone mode, see appendix A.2). This interface is not very user-friendly. In the future, a friendlier UI could be made to improve the ease of stand-alone running.

A

Image stabilisation software manual

A.1. Combined Running

Combined running of the image stabilisation software together with control is relatively easy to do. Enabling the `Drift correction` checkbox in the control software causes the control software to import the main loop of the stabilisation software. This way, in the backend, the stabilisation algorithm is executed on each picture that is taken by the control group.

A.2. Stand-Alone Running

When running the stabilisation software without the control software, there are a few boundary conditions that have to be met. First of all (logically), a folder with `.TIF` files is needed. These files should be named such that sorting this file list alphabetically results in the same order as their order if they are sorted from oldest to newest. In this same folder, a `.TXT` file is needed that contains the settings relevant for that experiment. The name of this file does not matter, as long as it has the `txt` extension. In this file, the 7 parameters from requirement [2.3] need to be placed. A sample settings file is placed in listing A.1.

Listing A.1: A sample settings file

```
output_location: folder/location/for/output
interval: 60
fps: 1.0
max_shift_percentage: 10
Out-painting: Blur
File name: video
Extention: .mp4
# Unfortunately the typo 'extention' is correct. It is a feature, not a bug
# You can use a hashtag at the start of a line to place a comment
```

Finally, make sure there are no subfolders or other files in this folder

Then, once this folder is correctly prepared, the stabilisation software can be started from a command line or terminal window. The software should receive the location of the folder with testing data as a command-line parameter. So starting the stabilisation software is done using:

```
python path/to/main.py path/to/testdata_folder
```

Once started, the command window or terminal window should display values for each image it encounters in the file. Then it should say that it is starting the preprocessing. If the outpainting method blur (or mean_outpaint, both call the same function) is chosen, this step might take a long time, depending on the size of the shifts in your dataset, the number of frames in the dataset and the speed of your computer. For more info about why this method takes a long time to run, see subsection E.1. When the video generator is started, this is also printed in the terminal. Then once the video is generated, the final message should be: `The video generator returned true`

B

More data

B.1. Global Comparison via AUC and a Practical Operating Point

For use in the image stabilisation software, the metric must support a practical decision rule of the form *"flag the registration as unreliable if the confidence drops below a threshold"*. To compare the metrics quantitatively in this setting, we treated each metric as a classifier and evaluated its performance across a range of thresholds.

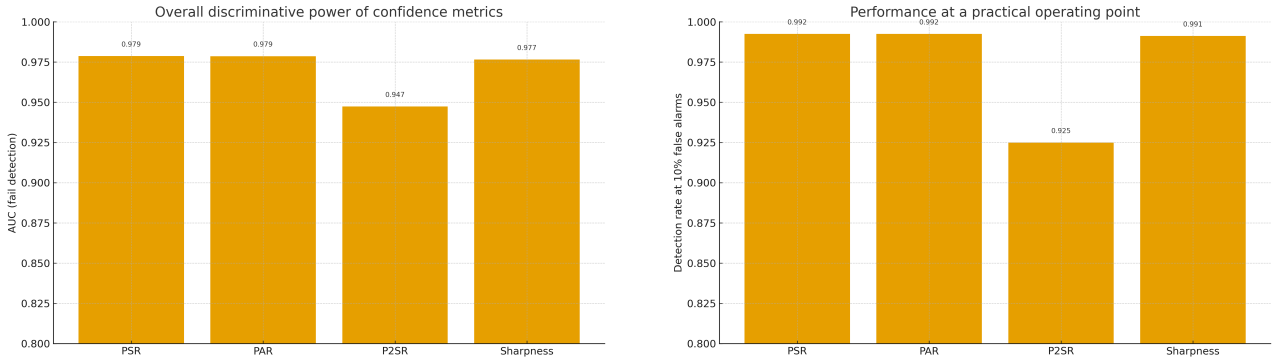
For each metric m, we swept a threshold τ over its value range and, for every τ, computed:

- the *detection rate* (recall) on failed registrations: fraction of $e > 2$ px cases with $m < \tau$,
- the *false-alarm rate* on successful registrations: fraction of $e \leq 2$ px cases with $m < \tau$.

From this sweep we derived:

1. the area under the corresponding detection–false-alarm curve (AUC), summarising the overall discriminative power of the metric;
2. the detection rate at a fixed, practically relevant false-alarm level of 10% ("how many failures do we catch if we accept that 10% of good frames are flagged as low-confidence?"). This level is not acceptable in the system, but for these testing purpose,s it serves to compare the confidence measures.

The resulting summary statistics are shown in Figure B.1.



(a) AUC per metric

(b) Detection rate at 10% false alarms

Figure B.1: Summary comparison of confidence metrics. (a) Overall discriminative power as measured by the area under the detection–false-alarm curve (AUC). (b) Detection rate on failed registrations at a fixed false-alarm rate of 10% on successful registrations.

B.1.1. Choosing a Confidence Measure

Several conclusions emerge:

- All four metrics achieve high AUC values (close to 1), confirming that they all contain strong information about registration quality.
- PSR, PAR, and sharpness attain very similar AUCs (around 0.98–0.99), while P2SR is slightly worse. This is consistent with the histogram view, where P2SR shows somewhat more overlap between successes and failures.
- At a fixed false-alarm rate of 10% (Fig. B.1b), PSR, PAR, and sharpness all detect almost all failed registrations (detection rate close to 0.99). P2SR again performs slightly worse, with a lower detection rate at the same false-alarm level.

Thus, several metrics would be acceptable candidates from a purely statistical perspective. PSR was selected as the primary confidence measure in the stabilisation software for the following reasons:

- It has a clear physical interpretation as the peak-to-sidelobe ratio of the correlation plane, i.e. the correlation peak height normalised by the mean and variance of the surrounding sidelobe region, so that higher PSR corresponds to a sharper and more dominant main peak and a lower risk of false detections [15]. Closely related PSR definitions are widely used as decision metrics in correlation-filter-based recognition and image registration, including applications based on normalised cross-correlation or phase correlation [15, 14].
- It provides one of the best overall discriminative performances (high AUC), matching or exceeding the alternatives B.1.
- At realistic operating points (e.g. around 10% false alarms), PSR detects practically all catastrophic failures, while keeping the rate of unnecessary beam shifts or dropped frames low B.1.

Alternative metrics (PAR, P2SR, and sharpness) are not used as decision variables in the current implementation.

B.2. Attached Data for Plots in Section 5.1

In this section, the CSV files are embedded in the PDF file. To open these, it is recommended to open the PDF with Adobe Acrobat.

B.2.1. Degradations with Fixed Shift:

This is the attached CSV file for the degradation with fixed shift:

B.2.2. All of the Transformations:

This is the attached CSV file with the data of all transformations for the stabiliser benchmark:

B.3. Synthetic SEM Transformations

Four categories of degradations are generated synthetically:

1. Translational drift. The reference image is shifted by integer pixel displacements in horizontal, vertical, and diagonal directions. Shift magnitudes range from 0 px to approximately 377 px, corresponding to more than half of the smallest image dimension, matching the large-drift regimes where phase correlation is known to become overlap-limited [11, 21, 14]. A representative example of a shifted frame is shown in Figure B.2; the image name is “10MIN”.

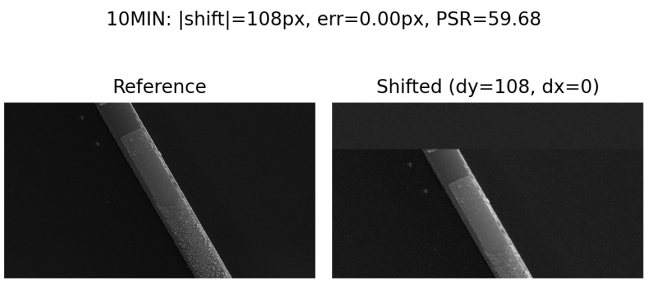


Figure B.2: Example of a synthetically shifted SEM frame (here $\Delta y = 108$ px).

2. Blur (defocus, beam broadening). Gaussian blur of increasing radius $r \in [0.1, 10]$ px is applied. This models loss of high-frequency data due to defocus or beam broadening, which is naturally described in the Fourier domain [25, 26, 20]. Figure B.3 shows two representative blurred images.

3. Noise (Gaussian and Poisson). Additive Gaussian noise with standard deviation up to $\sigma = 100$ grey levels and Poisson noise with scales between roughly 0.01 and 1.5 simulate electronic noise and low-dose shot noise, consistent with common noise models in imaging [25, 20].

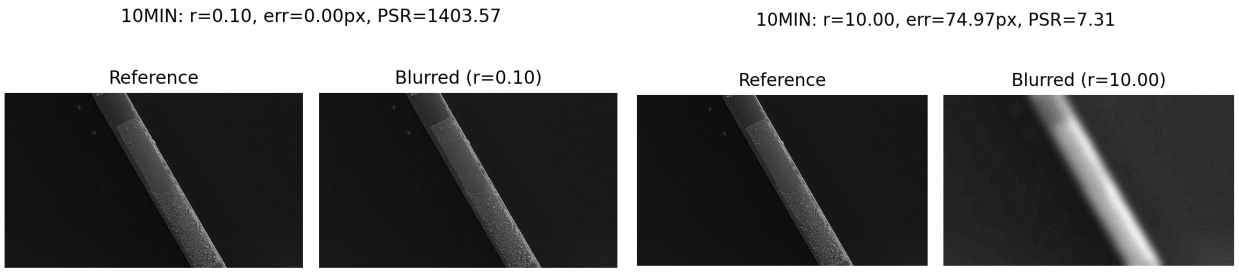


Figure B.3: Representative blurred frames used in the evaluation, generated by a Gaussian blur of radius $r = 0.1$ (left) and $r = 10$ (right).

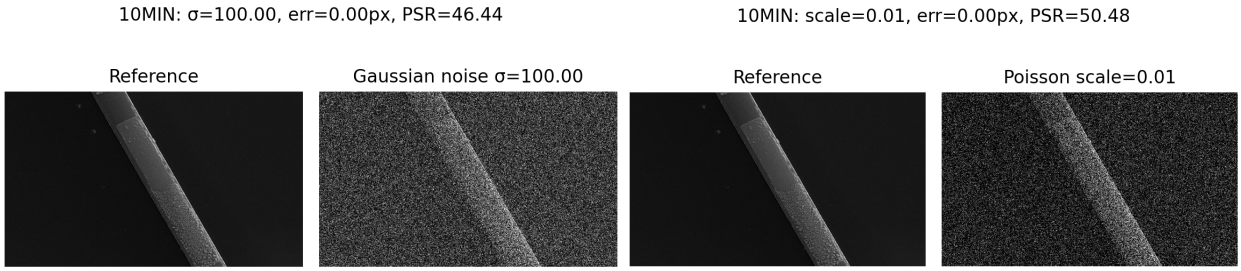


Figure B.4: Synthetic noise degradations: strong Gaussian noise (left) and strong Poisson noise (right).

4. Structural damage/contrast change. Randomised intensity and contrast modifications simulate beam damage, charging, and local specimen changes, similar in spirit to contrast-change scenarios discussed in registration surveys [27]. Damage fractions up to 10 (in the internal scale used by the generator) are applied, corresponding to a large range of affected areas (Figure B.5).

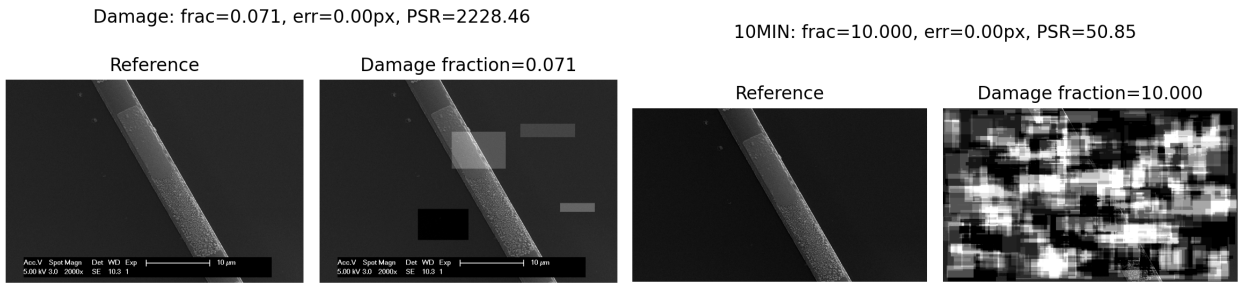


Figure B.5: Examples of synthetic structural damage affecting a small fraction (left) and a large fraction (right) of the frame.

C

Scale bar length calculation algorithm

Given the calibrated pixel size p_{corr} presented in subsection 4.7.4 (in mm/pixel), the algorithm used in this work (with its implementation in appendix F.7) proceeds as follows:

1. First, a visually acceptable range for the bar length in the image is set in pixels, N_{min} is determined as 17% of the frame width. N_{max} is set as either 40% of the frame width, or as 90% of the still available space in the data bar. The smallest value of these two is chosen. The available space in the data bar is dependent on the other values that are printed in there. These values then correspond to physical ranges

$$L_{min} = N_{min} p_{\text{corr}} \tag{C.1}$$
$$L_{max} = N_{max} p_{\text{corr}} \tag{C.2}$$

   expressed in millimetres.

2. Based on L_{max}, an appropriate display unit is chosen: nanometres (nm) when $L_{max} < 1$ μm, micrometres (μm) when $L_{max} < 1$ mm, and otherwise millimetres (mm). This selection avoids labels such as "0.003 mm" that are technically correct but visually awkward.

3. In the chosen unit, the admissible length interval $[L_{min}, L_{max}]$ is mapped to a scalar interval $[\ell_{min}, \ell_{max}]$, e.g. in μm. A "nice" numerical value v is then selected from a fixed set

$$\mathcal{V} = \{1, 2, 5, 10, 20, 50, 100, 200, 500\}, \tag{C.3}$$

   such that $\ell_{min} \leq v \leq \ell_{max}$. When several candidates exist, the smallest such v is chosen; if none fits, the largest element of $\mathcal{V}$ not exceeding ℓ_{max} is used. This step, in combination with the previous step, ensures that the printed label is easy to read and comparable to the scale bars that the SEM puts out by itself.

4. The chosen label value v combined with the selected unit is converted back to a physical length L_{label} in millimetres. The corresponding bar length in pixels is then

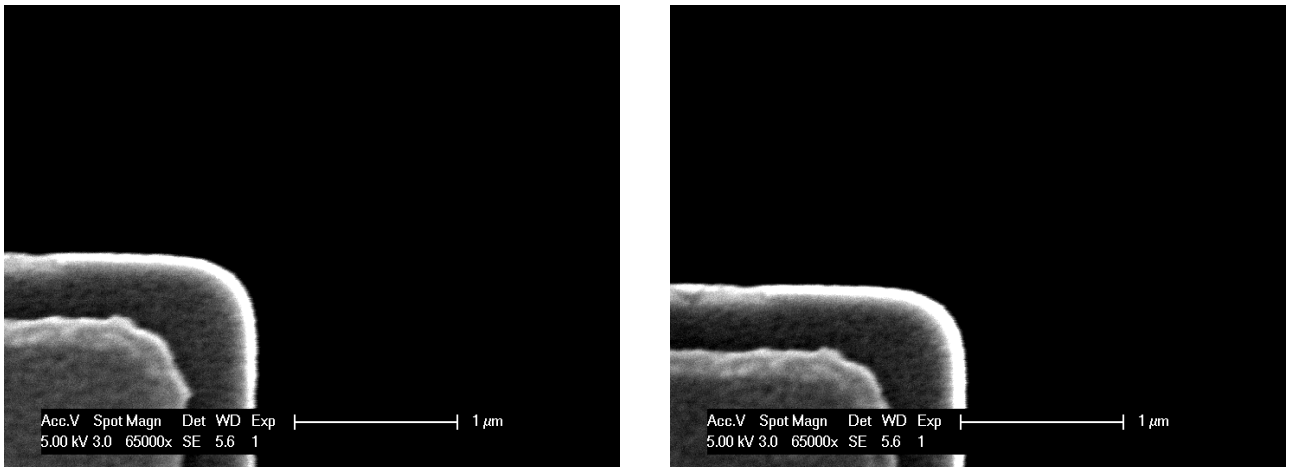
$$N_{\text{bar}} = \text{round}\left(\frac{L_{\text{label}}}{p_{\text{corr}}}\right) \tag{C.4}$$

   and the overlay is drawn using N_{bar} pixels and the text label (e.g. "5 μm").

D

Microscope drift velocity

As a part of the design process in this thesis, the question of how large the drift velocity actually is came up. A quick measurement was performed to check what the drift was. For mounting the samples, one can either use carbon tape or clamps. Clamps hold the sample tighter, so less drift will be expected in that case. To see a bad case of drift, the worse option of carbon tape was chosen. This option was given its best chance, however, by making sure that a fresh piece of tape was used. In figure D.1, the two pictures that were taken are shown.



(a) Sample at $t = 0$

(b) Sample at $t = 30$ minutes

Figure D.1: Effect of shift magnitude on stabiliser performance.

Then our stabiliser software was used to find the pixel shift between the two images and to find the corrected pixel size. The returned pixel shift was `-50, -33` and the reported pixel size was `5.252244089283043e-06` mm/px.

From this, the drift distance is

$$d = \sqrt{x_{\text{drift}}^2 + y_{\text{drift}}^2} = 59.908 \text{ pixels}, \tag{D.1}$$

which corresponds to $0.315\,\mu$m. Since the time between the two images is 30 minutes (0.5 h), this yields a drift rate of

$$\frac{59.908 \text{ pixels}}{0.5 \text{ h}} = 119.8 \text{ pixels/h}, \tag{D.2}$$

or equivalently

$$\frac{59.908 \text{ pixels}}{30 \text{ min}} = 2.00 \text{ pixels/min}. \tag{D.3}$$

In physical units, the drift velocity is

$$v_{\text{drift}} = 0.629 \ \mu\text{m/h}, \tag{D.4}$$

which is equivalent to

$$629 \text{ nm/h} \approx 10.5 \text{ nm/min}. \tag{D.5}$$

E

Additional Technical Details

E.1. Local-Mean Outpainting

This iterative local-mean filling scheme is a discrete derivation of harmonic (Laplace) inpainting, where the unknown region is driven towards the harmonic extension of the known boundary values [43, 20]. The core of the outpainting method is the function `pad_with_local_mean`. Given a single-frame image $I \in \{0, \dots, 255\}^{H \times W}$ and integer padding widths (t, b, ℓ, r), it returns a padded image $\tilde{I} \in \{0, \dots, 255\}^{H_{\text{pad}} \times W_{\text{pad}}}$ in which the original pixels are preserved, and all newly created pixels are filled with local averages of nearby known pixels.

The implementation first converts the image to `float32` and initialises the padded canvas with `NaN` values. In equation E.1 offset accounts for the top, bottom, left and right padding for that given frame.

$$\tilde{I}(p) = \begin{cases} I(p - \text{offset}) & \text{if } p \text{ lies inside the original frame,} \\ \text{NaN} & \text{otherwise,} \end{cases} \tag{E.1}$$

A Boolean mask $M(p) = \mathbb{1}_{\{\tilde{I}(p)=\text{NaN}\}}$ (where NaN values are placed in the places of the pixels that should be filled) in keeps track of pixels that still need to be filled.

Unknown pixels are then filled iteratively by averaging over their valid 3×3 neighbourhood. Let $u^{(k)}$ denote the padded image after k iterations, where $u^{(0)} = \tilde{I}$. For each iteration that is computed, using fast C-based convolutions (`scipy.ndimage.convolve`) [44] the sum and count of valid neighbours are stored:

$$S^{(k)} = K * \big(u^{(k)} \text{ with NaNs replaced by } 0\big), \tag{E.2}$$

$$C^{(k)} = K * \big(\mathbb{1}_{\{\text{pixel is valid in } u^{(k)}\}}\big), \tag{E.3}$$

where K is a 3×3 kernel of ones and $*$ denotes convolution. Pixels that are still missing but have at least one valid neighbour are updated according to

$$u^{(k+1)}(p) = \begin{cases} S^{(k)}(p)/C^{(k)}(p) & \text{if } M(p) \wedge C^{(k)}(p) > 0, \\ u^{(k)}(p) & \text{otherwise} \end{cases} \tag{E.4}$$

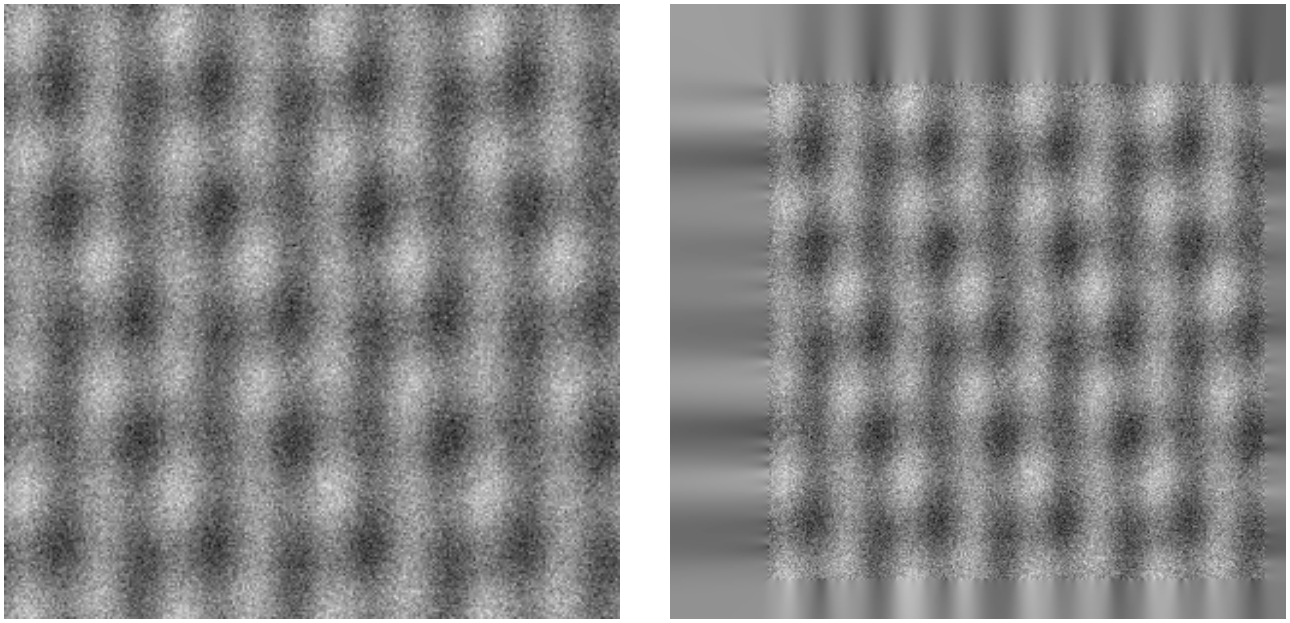
The mask M that keeps track of pixels still to be filled in is then updated accordingly, and the process is repeated until there are no missing pixels left, or a maximum number of iterations has been reached. In practice, choosing `max_iter` $\approx \max(t + b, \ell + r) + 2$ is sufficient, because each iteration propagates known values roughly one pixel further into the unknown band.

Equation (E.4) can be interpreted as a discrete diffusion or (approximate) Laplace inpainting step: the unknown region is pushed towards the harmonic extension of the known boundary values. The resulting filled margins blend smoothly with the original frame and avoid sharp artificial edges.

After the iterative filling step, any remaining `NaN` values (which should not occur under normal circumstances) are replaced by zero as a fallback. The image is then clipped to the valid grey range and cast back to 8-bit unsigned integers for storage:

$$\tilde{I}_{\text{out}} = \text{clip}(u^{(k_{\text{final}})}, 0, 255) \in \{0, \dots, 255\}^{H_{\text{pad}} \times W_{\text{pad}}}. \tag{E.5}$$

Figure E.1 shows an example frame before and after outpainting-based drift correction. The newly filled margins fade smoothly to a background.



(a) Unprocessed frame (resolution: 256x256) **(b)** Processed frame (resolution: 316x316

Figure E.1: Example of software drift correction using outpainting. The margins are filled by iterative local-mean interpolation.

E.2. Mathematical Interpretation of the Stabiliser Results

Let F and G denote the noisy spectra of the reference and current frame, and write each as a sum of a signal term and a noise term, $F = S_F + N_F$, $G = S_G + N_G$. The (unnormalised) cross-spectrum is

$$F\overline{G} = (S_F + N_F)(\overline{S_G} + \overline{N_G}) = S_F\overline{S_G} + S_F\overline{N_G} + N_F\overline{S_G} + N_F\overline{N_G} = S + E, \tag{E.6}$$

with $S := S_F\overline{S_G}$ and E collecting all noise-dependent terms. The cross-power spectrum used in phase correlation is

$$C_{\text{noisy}} = \frac{S + E}{|S + E| + \epsilon}, \tag{E.7}$$

where $\epsilon > 0$ is a small numerical stabiliser that suppresses frequency components with vanishing magnitude, which carry no reliable phase information.

When the signal term dominates, $|E| \ll |S|$, we have $S + E \approx S$ and $|S + E| \approx |S|$, so

$$C_{\text{noisy}} \approx \frac{S}{|S| + \epsilon} = \frac{S_F\overline{S_G}}{|S_F||S_G| + \epsilon}. \tag{E.8}$$

In this regime, the phase of the cross-power spectrum (and thus the location of the correlation peak) is essentially unchanged by the noise, even if the Fourier magnitudes are strongly perturbed. This explains why additive Gaussian noise and all practical Poisson noise levels leave the localisation error at zero, and mainly manifests as a gradual PSR reduction, as widely discussed in Fourier-based signal processing [26, 25, 20].

The synthetic structural damage modifies intensities only inside random rectangles; a substantial fraction of the frame remains unchanged. Phase correlation effectively locks onto this intact region, so the dominant signal term S is preserved, and the phase structure remains coherent. Consequently, the PSR stays high, and the estimated shifts remain accurate, as seen in Fig. 5.6c and in agreement with general registration theory [14, 27].

Blur behaves fundamentally differently. Gaussian blur corresponds in the Fourier domain to multiplication by

$$H(k_x, k_y) = e^{-\sigma^2(k_x^2 + k_y^2)} \tag{E.9}$$

which strongly suppresses high-frequency components. These components are responsible for producing a narrow, unambiguous peak in the correlation surface. As H attenuates them, the spectrum of both frames collapses to its low-frequency content; the correlation peak broadens, sidelobes become more ambiguous, and the localisation error grows quickly. This is exactly the behaviour observed in Fig. 5.5, where PSR and accuracy

deteriorate rapidly once the blur radius exceeds a few pixels and approach an error plateau when almost all useful high-frequency information has been removed [26, 25, 20].

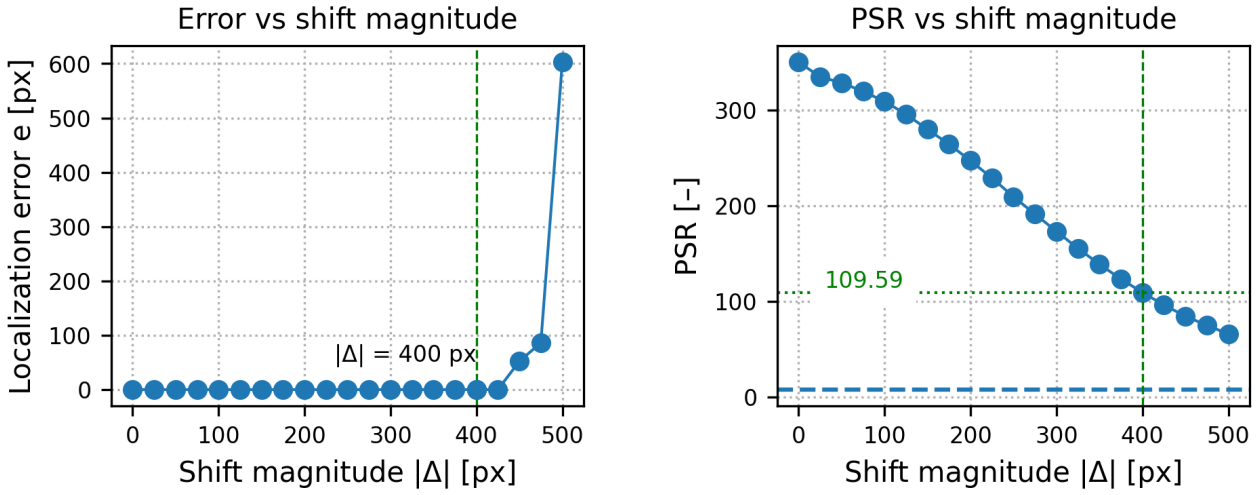
Overall, the fixed-shift degradation experiment confirms that the phase-correlation stabiliser is robust to noise and structural changes under SEM imaging conditions, while blur and extreme noise define the practical limits of its reliability. Crucially, the PSR-based confidence measure remains a reliable indicator of failure: in all regimes where the estimated shifts become inaccurate, the PSR drops well below the operational threshold used by the live system, consistent with correlation-filter practice [15].

E.3. High-definition (HD) Images vs. SD: Robustness and Thresholds

The fixed-shift degradation experiments in section 5.2 were carried out on standard-definition (SD) frames of size 712×484 px, using a constant ground-truth translation $(\Delta y_{\text{true}}, \Delta x_{\text{true}}) = (-100, -100)$ px. To verify that the conclusions also hold for high-definition (HD) acquisition—and to determine resolution-dependent confidence thresholdswe repeated the entire experiment on HD frames ($\approx 2\times$ width and height), but increased the imposed drift fourfold to

$$(\Delta y_{\text{true}}, \Delta x_{\text{true}}) = (-400, -400)\,\text{px}. \tag{E.10}$$

Figure E.2 summarises the behaviour of the stabiliser as a function of shift magnitude for HD images. Up to $|\Delta| \approx 400$ px, the localisation error remains identically zero, even though the absolute shift is four times larger than in the SD experiment. Only for still larger displacements does the error rise steeply. The corresponding PSR values (Fig. E.2b) remain high in the zero-error regime (around PSR ≈ 110 at $|\Delta| = 400$ px) and drop towards PSR ≈ 60 only once the shift becomes so large that failures appear. This shows that, for an equal *percentage* of field-of-view (FOV) drift, HD images are substantially easier to register: the stabiliser can tolerate about four times the SD pixel shift before exhibiting comparable failure behaviour, in agreement with general scaling arguments for registration on higher-resolution grids [14]. As can be seen in Appendix D, for an image with magnification level 65000×, the pixel shift is approximately 2 pixels per minute. At operational interval timings, the acquisition time between two frames is about 1 minute, so we expect a 2 pixel drift between consecutive frames for this magnification level. As seen in Figure E.2, this is well within the error-free correction zone of the system.



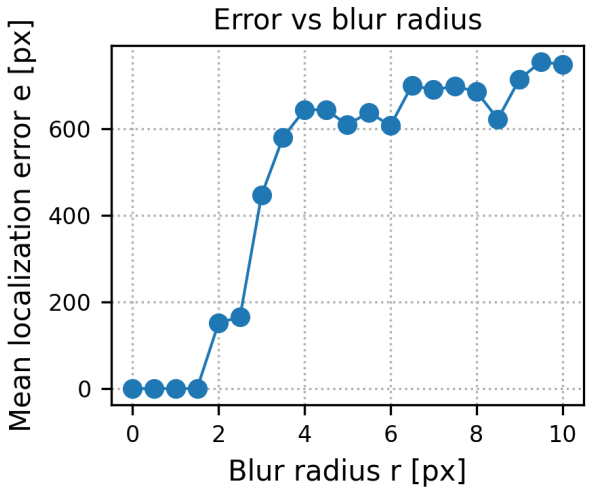
(a) Mean localisation error vs. shift magnitude.

(b) PSR vs. shift magnitude.

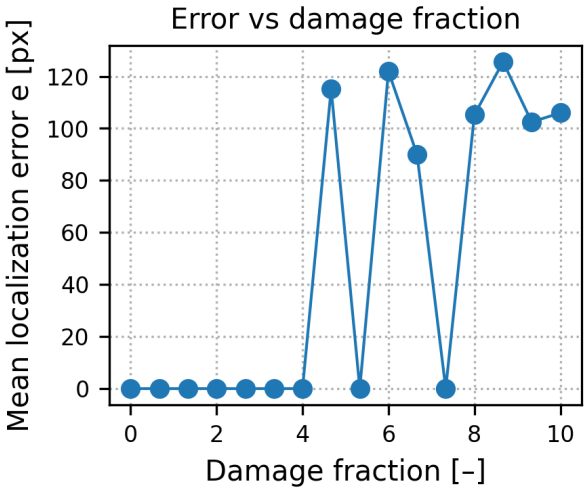
Figure E.2: HD fixed-shift experiment: effect of shift magnitude on localisation error and PSR. The vertical line marks $|\Delta| = 400$ px, where the error is still zero and PSR is ≈ 110.

The degradation sweeps for blur, Gaussian noise, Poisson noise and structural damage at this HD shift (Fig. E.3) confirm the same qualitative picture, as in the SD case: Gaussian blur is the dominant failure mode, whereas realistic noise levels and structural changes leave the registration essentially intact. Crucially, this holds even though the underlying drift is four times larger in pixels, which is consistent with Fourier-based analyses of registration robustness [14].

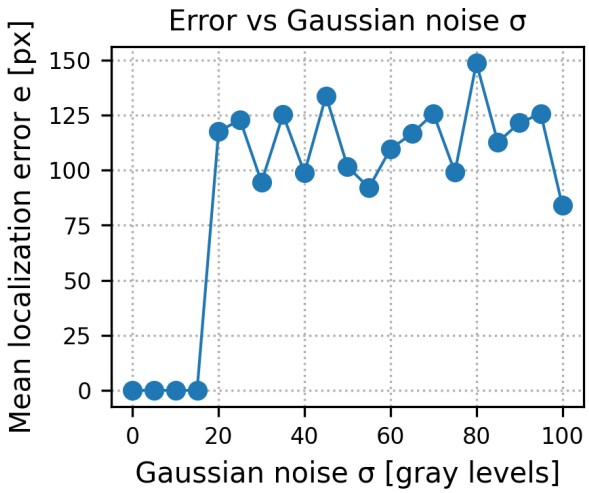
For system design, we are primarily interested in the worst-case conditions. The HD experiments show that, at the same relative FOV shift, the failure boundary is reached only at much larger absolute pixel displacements



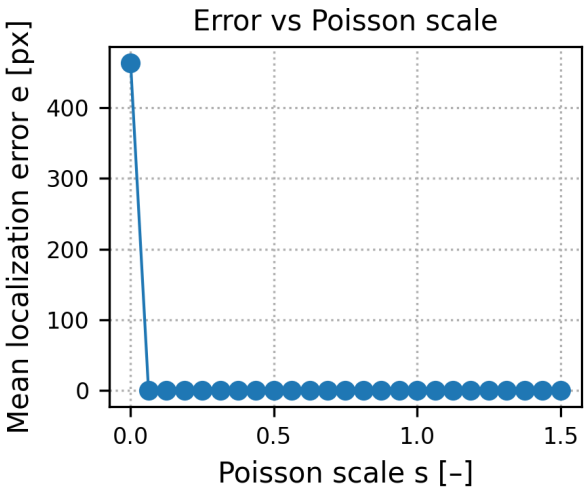
(a) Blur radius r.



(b) Damage fraction.



(c) Gaussian noise σ.



(d) Poisson scale s.

Figure E.3: HD fixed-shift degradation experiments at $(\Delta y_{\text{true}}, \Delta x_{\text{true}}) = (-400, -400)$ px. As in the SD case, blur is the only degradation that leads to large localisation errors; noise and structural damage are mostly acceptable.

than in SD. In other words, operating the stabiliser at a given percentage of FOV is *more conservative* for HD than for SD.

Based on these results, we implement *resolution-specific* confidence limits in the controller. In addition to the relative PSR rule of section 4.5.2, the software uses two fixed, empirically safe absolute PSR thresholds: one for SD acquisition and one for HD acquisition, each chosen from the corresponding SD and HD degradation curves at the last PSR value for which no failures were observed. This ensures that the stabiliser runs with a comfortable safety margin in both modes while fully exploiting the extra robustness available at HD resolution.

F

Python Code

The following sections contain the complete Python source code used in this project. All files are included exactly as they appear in the final software implementation. In writing the Python code, attention was paid to following the PEP 8 style guidelines and to providing consistent type hints in accordance with PEP 484, ensuring readability, maintainability, and clarity throughout the implementation.

F.1. communication.py

```
"""
Communication implementations for the analysis system.

This module defines a generic communication protocol interface and concrete
implementations for different communication strategies:

- ``CommunicationProtocol``: Abstract base class that specifies the required
  methods for any communication backend (connect, send, receive, etc.).
- ``FolderCommunication``: Uses a filesystem folder as a pseudo-communication
  channel by turning files into messages (e.g. manager info from a .txt file
  and image locations from image files).
- ``CombinedCommunication``: In-memory communication channel that stores
  messages in a local queue, for integration with the control software.

It also provides ``read_manager_data`` to parse configuration parameters from
the ManagerInfo file into a dictionary.
"""


from abc import ABC, abstractmethod
import os
from typing import Dict, List

try:
    from Analysis.messages import Message, message_types
except ModuleNotFoundError as e:
    if e.name in ("Analysis", "Analysis.messages"):
        from messages import Message, message_types
    else:
        raise


class CommunicationProtocol(ABC):
    """This base class enforces what functions a comms method should have."""

    @abstractmethod
    def connect(self, initializer: str) -> bool:
        pass

    @abstractmethod
    def send(self, message: Message) -> None:
        pass

    @abstractmethod
    def receive(self, receiver: str) -> Message:
```

```
        pass

    @abstractmethod
    def message_available(self, receiver: str) -> bool:
        pass

    @abstractmethod
    def check_connected(self, entity: str) -> bool:
        pass

    @abstractmethod
    def close(self, entity: str) -> None:
        pass


class FolderCommunication(CommunicationProtocol):
    """The communication implementation for using just a folder.

    Protocol description
    This protocol acts like a communication class, except that it uses pre-existing
    data in a given folder. The messages are created based on the files that exist
    in that given folder.

    First the protocol searches for a .txt file with the manager data. This txt file
    should be given like this:

        PARAMETER1_NAME: PARAMETER1_VALUE
        PARAMETER2_NAME: PARAMETER2_VALUE
        PARAMETER3_NAME: PARAMETER3_VALUE
        etc

    Then it goes through the rest of the filenames. These are all assumed to be
    images, and will generate an ImageLocation messagetype.

    The sending functions are implemented but do not contain any logic.
    """

    def __init__(self, folder_location: str) -> None:
        self.folder_location: str = folder_location
        self.queue: List[Message] = []
        self.connect_status: List[str] = []
        self.connect_status.append("CONTROL")  # Fake that control exists

    def connect(self, entity: str) -> bool:
        self.connect_status.append(entity)
        self.queue = []
        file_list = [
            f
            for f in os.listdir(self.folder_location)
            if os.path.isfile(os.path.join(self.folder_location, f))
        ]
        file_list = [f"{self.folder_location}\\{f}" for f in file_list]
        not_img_msg: List[str] = []  # list to store the messages that are not img locations

        # The order matters here. Because we search for a txt first, the first
        # message will always be managerdata.
        for f in file_list:
            if f.endswith(".txt"):
                with open(f, "r") as file:
                    data = [line for line in file]
                    # data now is a list of lines
                msg = Message(message_types.ManagerInfo, data, "ANALYSIS")
                self.queue.append(msg)
                not_img_msg.append(f)

        # Remove all non-img locations from file_list
```

```
        for f in not_img_msg:
            file_list.remove(f)

        for f in file_list:
            filename = f.split(".")[0]
            if filename.endswith("_M"):
                self.queue.append(
                    Message(message_types.ControlMoved, True, "ANALYSIS")
                )
            msg = Message(message_types.ImageLocation, f, "ANALYSIS")
            self.queue.append(msg)

        if len(self.queue) > 0:
            # Add a done message to signal start of vid generation
            self.queue.append(
                Message(message_types.Done, True, "ANALYSIS")
            )
        return len(self.queue) > 0

    def send(self, message: Message) -> None:
        # The file communication thing does not do sending
        pass

    def receive(self, receiver: str) -> Message:
        # Assumes that a message is available
        queue = [msg for msg in self.queue if msg.receiver == receiver]
        msg = queue.pop(0)
        self.queue.remove(msg)
        return msg

    def message_available(self, receiver: str) -> bool:
        queue = [msg for msg in self.queue if msg.receiver == receiver]
        return len(queue) > 0

    def check_connected(self, entity: str) -> bool:
        return entity in self.connect_status

    def close(self, entity: str) -> None:
        # Nothing to close for folder-based communication
        pass


class CombinedCommunication(CommunicationProtocol):
    """
    Our communication link with the control software.

    This implementation keeps all messages in a queue.

    Attributes:
        queue: A list of messages waiting to be delivered to their receivers.
        connect_status: A list of entity identifiers that are marked as
            connected to this communication channel.
    """

    def __init__(self, initial_message: Message) -> None:
        self.queue: List[Message] = []
        self.queue.append(initial_message)
        self.connect_status: List[str] = []

    def connect(self, entity: str) -> bool:
        self.connect_status.append(entity)
        # If this returns False then wow
        return entity in self.connect_status

    def send(self, message: Message) -> None:
        self.queue.append(message)
```

```
    def receive(self, receiver: str) -> Message:
        # Assumes that a message is available
        queue = [msg for msg in self.queue if msg.receiver == receiver]
        msg = queue.pop(0)
        self.queue.remove(msg)
        return msg

    def message_available(self, receiver: str) -> bool:
        queue = [msg for msg in self.queue if msg.receiver == receiver]
        return len(queue) > 0

    def check_connected(self, entity: str) -> bool:
        return entity in self.connect_status

    def close(self, entity: str) -> None:
        self.connect_status.remove(entity)


def read_manager_data(message: Message) -> Dict[str, str]:
    """
    Parse manager configuration data from a ManagerInfo message.

    The message_content is expected to contain lines in the format
    ``KEY: VALUE``. Lines starting with ``#`` are treated as comments and
    ignored. The resulting dictionary must contain at least the following
    keys: ``"max_shift_percentage"``, ``"output_location"``, ``"fps"``,
    and ``"interval"``.

    Args:
        message: The message containing manager configuration lines in its
            message_content. The message_type must be ManagerInfo.

    Returns:
        A dictionary mapping configuration parameter names to their
        corresponding string values.

    Raises:
        ValueError: If the message does not have type ManagerInfo.
        AttributeError: If one or more required parameters are missing
            from the message_content.
    """
    manager_data: Dict[str, str] = dict()
    if message.message_type != message_types.ManagerInfo:
        raise ValueError("Message does not contain manager info")

    delimiter = ":"
    for line in message.message_content:
        if line.startswith("#"):
            continue
        # Split at the first delimiter, all before that is the key, all after is val
        key, val = line.split(delimiter, 1)
        # Use strip to remove whitespace and newlines after the delimiter
        manager_data[key.strip()] = val.strip(" \n")

    required = ["output_location", "interval", "fps", "max_shift_percentage", "Out-painting", "File
    ↪  name", "Extention"] # Note the typo in extention/extension
    missing = set(required) - manager_data.keys()
    if missing:
        raise AttributeError(
            "Not all required parameters have been received. "
            "Be sure to send them in one message.\n"
            f"Missing parameter(s): {', '.join(missing)}"
        )
    return manager_data
```

F.2. frame.py

```
"""
Frame handling utilities for the SEM images.

This module defines the `Frame` class, which represents a single microscope
image with associated metadata, stabilization information, pixel-size
calibration, and FEI_SFEG microscope metadata extraction.

Features:
- Load and store grayscale image data (numpy array).
- Automatically extract FEI_SFEG metadata from TIFF files using `tifffile`.
- Compute real-world pixel size in millimeters based on microscope metadata.
- Store and convert stabilization offsets from pixels to millimeters.
- Expose convenience helpers for beam-shift computation.

This module is designed for use in microscope image stabilization workflows.
"""

from typing import Optional, Tuple

from PIL import Image, ImageOps
import numpy as np
from tifffile import TiffFile

try:
    from Analysis.messages import Message
except ModuleNotFoundError as e:
    if e.name in ("Analysis", "Analysis.messages"):
        from messages import Message
    else:
        raise


microscope_scale_correction = 0.8931653453


class Frame:
    """
    Represents a single SEM image frame, including image data, FEI metadata,
    stabilization results, physical calibration information, and beam-shift
    computations.

    A `Frame` can:
    - Load its image and original dimensions.
    - Read FEI microscope metadata for calibration.
    - Track stabilization shifts in both pixels and millimeters.
    - Compute necessary beam-shift commands for the SEM.
    """

    def __init__(self) -> None:
        self.image: Optional[np.ndarray] = None
        self._stabilization_data: Optional[Tuple[float, float]] = None
        self.stabilization_data_mm: Optional[Tuple[float, float]] = None

        self.path: Optional[str] = None
        self.size: Optional[Tuple[int, int]] = None
        self.size_original: Optional[Tuple[int, int]] = None

        self.fei_info: Optional[dict] = None
        self.beam_shift_initial_mm: Optional[Tuple[float, float]] = None
        self.beam_shift_target_mm: Optional[Tuple[float, float]] = None

    # ------------------------------------------------------------------
    # Public API
    # ------------------------------------------------------------------
```

```
    def load_image(self, file_path: str) -> None:
        """
        Load an image file, convert to grayscale, store as numpy array, and
        attempt FEI metadata extraction.

        Args:
            file_path: Path to the image file to load.

        Raises:
            NameError: If the file extension is not supported by Pillow.
        """
        self.path = file_path

        ext = "." + file_path.split(".")[-1].lower()
        if ext not in Image.registered_extensions():
            raise NameError(
                f"File {file_path} does not have a supported extension\n"
                f"{ext}"
            )

        img = Image.open(file_path)
        img = ImageOps.autocontrast(img, cutoff=0, ignore=None)
        img = ImageOps.grayscale(img)

        self.image = np.asarray(img)
        self.__update_size()
        self.__update_size_original()

        # Load FEI metadata if available (TIFF only)
        self.__load_fei_metadata()

    def from_message(self, msg: Message) -> None:
        """
        Load the image referenced in a Message.

        Args:
            msg: A Message whose content is a file path string.
        """
        self.load_image(msg.message_content)

    def update_image(self, img_data: np.ndarray) -> None:
        """
        Replace the image data (e.g., after stabilization) but preserve metadata.

        Args:
            img_data: A numpy array representing the updated image.
        """
        self.image = img_data
        self.__update_size()

    def get_image(self) -> Optional[np.ndarray]:
        """Return the image numpy array."""
        return self.image

    def compute_pixel_size_mm(self) -> Optional[float]:
        """
        Compute the pixel size in millimeters using FEI metadata.

        Uses FEI tag fields:
            ImageDevice.SizeY:        physical image height in mm
            Vector.Magnification:     microscope magnification

        Formula:
            fov_mm = SizeY_mm / Magnification
            pixel_size_mm = fov_mm / image_height_px
```

```
        Returns:
            Pixel size in mm, corrected with microscope_scale_correction,
            or None if the metadata is unavailable or incomplete.
        """
        if self.fei_info is None:
            self.__load_fei_metadata()
        if self.fei_info is None or self.size_original is None:
            return None

        try:
            image_device = self.fei_info.get("ImageDevice", {})
            vector = self.fei_info.get("Vector", {})

            size_y_mm = float(image_device["SizeY "])  # FEI trailing space
            magnification = float(vector["Magnification "])

            height_px = self.size_original[0]
            fov_y_mm = size_y_mm / magnification
            pixel_size_mm = fov_y_mm / height_px

            return pixel_size_mm * microscope_scale_correction
        except FileNotFoundError("FEI metadata file not found"):
            return None

    # ------------------------------------------------------------------------
    # Properties
    # ------------------------------------------------------------------------

    @property
    def stabilization_data(self) -> Optional[Tuple[float, float]]:
        """Return (shift_x_px, shift_y_px) stabilization tuple in pixels."""
        return self._stabilization_data

    @stabilization_data.setter
    def stabilization_data(self, value: Optional[Tuple[float, float]]) -> None:
        """
        When stabilization data is updated, automatically compute the physical
        shift in millimeters and update beam-shift target position.

        Args:
            value: Tuple of (shift_x_px, shift_y_px), or None.
        """
        self._stabilization_data = value
        self.__update_stabilization_data_mm()

    # ------------------------------------------------------------------------
    # Internal helpers
    # ------------------------------------------------------------------------

    def __update_size_original(self) -> None:
        """Store the original image array shape."""
        self.size_original = None if self.image is None else self.image.shape

    def __update_size(self) -> None:
        """Store the current image array shape."""
        self.size = None if self.image is None else self.image.shape

    def __load_fei_metadata(self) -> None:
        """
        Attempt to load FEI_SFEG metadata from TIFF tags.

        Populates:
            fei_info
            beam_shift_initial_mm

        If anything fails, metadata values are set to None.
```

```
        """
        if self.path is None:
            return

        try:
            with TiffFile(self.path) as tif:
                page = tif.pages[0]
                tags = page.tags

                if "FEI_SFEG" in tags:
                    fei_tag = tags["FEI_SFEG"]
                elif 34680 in tags:
                    fei_tag = tags[34680]
                else:
                    self.fei_info = None
                    self.beam_shift_initial_mm = None
                    return

                self.fei_info = fei_tag.value

                # Try to extract beam shift from metadata
                self.beam_shift_initial_mm = None
                try:
                    if isinstance(self.fei_info, dict):
                        vector = self.fei_info.get("Vector", {})
                        bx = vector.get("BeamShiftX ", vector.get("BeamShiftX"))
                        by = vector.get("BeamShiftY ", vector.get("BeamShiftY"))

                        if bx is not None and by is not None:
                            self.beam_shift_initial_mm = (float(bx), float(by))
                except FileNotFoundError("FEI metadata file not found"):
                    self.beam_shift_initial_mm = None

        except TypeError("Only use TIF file, we load metadata using the tifffile package"):
            self.fei_info = None

    def __update_stabilization_data_mm(self) -> None:
        """
        Convert pixel-based stabilization offset into millimeter-scale
        displacement using microscope calibration, and compute the final
        beam-shift target position if initial shift is known.
        """
        if self._stabilization_data is None:
            self.stabilization_data_mm = None
            self.beam_shift_target_mm = None
            return

        pixel_size_mm = self.compute_pixel_size_mm()
        if pixel_size_mm is None:
            self.stabilization_data_mm = None
            self.beam_shift_target_mm = None
            return

        shift_x_px, shift_y_px = self._stabilization_data

        # Convert pixel shift to millimeters (SEM coordinate system)
        shift_x_mm = shift_x_px * pixel_size_mm
        shift_y_mm = shift_y_px * pixel_size_mm

        # SEM coordinate system: invert Y
        self.stabilization_data_mm = (shift_x_mm, -shift_y_mm)

        if self.beam_shift_initial_mm is not None:
            bx0, by0 = self.beam_shift_initial_mm
            dx, dy = self.stabilization_data_mm
            self.beam_shift_target_mm = (bx0 + dx, by0 + dy)
```

```
        else:
            self.beam_shift_target_mm = None
```

F.3. main.py

```
"""
Main analysis pipeline for SEM frame stabilization and video generation.

This module wires together the communication layer, frame loading,
stabilization, preprocessing, and final video generation.

High-level flow:
    1. Connect to a CommunicationProtocol implementation.
    2. Receive manager configuration and image locations via messages.
    3. For each image:
        - Load as a Frame
        - Stabilize against a reference frame
        - Optionally request beam-shift corrections from CONTROL
    4. After a Done message:
        - Validate stabilization data
        - Pad frames and add annotations
        - Render the output video to disk.
"""

from __future__ import annotations

import sys
from time import sleep
from pathlib import Path
from typing import List, Any, Dict, Optional

import numpy as np

try:
    from Analysis.communication import (
        FolderCommunication,
        read_manager_data,
        CommunicationProtocol,
    )
    from Analysis.frame import Frame
    from Analysis.messages import message_types as mt, Message
    from Analysis.preprocessing import (
        pad_frames,
        check_stabilization_data,
        check_same_size,
        generate_annotation,
    )
    from Analysis.stabilization import stabilize, beamshift_needed
    from Analysis.video import extract_codec, frames_to_video
except ModuleNotFoundError as e:
    if e.name in (
        "Analysis",
        "Analysis.communication",
        "Analysis.messages",
        "Analysis.preprocessing",
        "Analysis.stabilization",
        "Analysis.video",
    ):
        from communication import (
            FolderCommunication,
            read_manager_data,
            CommunicationProtocol,
        )
        from frame import Frame
        from messages import message_types as mt, Message
        from preprocessing import (
```

```
            pad_frames,
            check_stabilization_data,
            check_same_size,
            generate_annotation,
        )
        from stabilization import stabilize, beamshift_needed
        from video import extract_codec, frames_to_video
    else:
        raise


def analysis_main(comm: CommunicationProtocol) -> None:
    """
    Main processing loop for the analysis pipeline.

    This function:
    - Connects to the communication backend.
    - Receives configuration and image-location messages.
    - Performs frame stabilization and optionally requests beam-shift moves.
    - After receiving a Done message, preprocesses frames and generates a video.

    Args:
        comm: An implementation of CommunicationProtocol used to exchange
            messages with CONTROL and other components.
    """
    connection_success = comm.connect("ANALYSIS")
    if not connection_success:
        raise RuntimeError("Could not connect to communication method")

    control_ready = comm.check_connected("CONTROL")
    tries = 5
    while not control_ready:
        print(
            "Control not connected to communication protocol.\n"
            f"{tries} retries left\n"
            f"Retrying in 2 seconds"
        )
        if tries <= 0:
            raise RuntimeError("Control was not connected to the communication")
        tries -= 1
        sleep(1)
        control_ready = comm.check_connected("CONTROL")

    # Now there is communication. We can continue
    framebuffer: List[Frame] = []
    manager_data: Dict[str, Any] = {}

    base_location: Optional[int] = None
    confidence: Optional[float] = None
    additional_shift_x, additional_shift_y = 0, 0

    while True:
        # inf loop to check for messages
        if not comm.message_available("ANALYSIS"):
            continue  # go back to start of loop, this way we limit our indents

        msg = comm.receive("ANALYSIS")
        if msg.message_type == mt.ManagerInfo:
            manager_data = read_manager_data(msg)

        elif msg.message_type == mt.ControlMoved:
            # insert logic for how to handle if a move was performed
            pass

        elif msg.message_type == mt.ImageLocation:
            frame = Frame()
```

```
            try:
                frame.from_message(msg)
            except NameError as e:
                print(e)
                continue  # if it is an unsupported file type, then just continue,

            if base_location is None:
                # set the reference frame to the first one in the buffer
                base_location = 0
                dx, dy = 0, 0
                frame.stabilization_data = (dx, dy)
                confidence = None
            else:
                ref_frame = framebuffer[base_location]
                dx, dy, confidence = stabilize(frame, ref_frame, True)
                frame.stabilization_data = (
                    dx + additional_shift_x,
                    dy + additional_shift_y,
                )
                print(f"Current confidence: {confidence}")

            framebuffer.append(frame)
            print(dx, dy)  # kinda want to know the shift of each image

            # Now the frame is processed. Time to do some other checks
            if beamshift_needed(frame, int(manager_data["max_shift_percentage"])):
                # shift is too large, so we need to send a shift request to control
                # Clamp the required shift to the allowed values
                clamped_shift = (
                    max(-0.02, min(frame.stabilization_data_mm[0], 0.02)),
                    max(-0.02, min(frame.stabilization_data_mm[1], 0.02)),
                )
                message = Message(mt.MoveCommand, clamped_shift, "CONTROL")
                comm.send(message)

            # Beam might now have corrected
            # But are we confident?
            if confidence is None:
                # catch the first run, when we have not yet defined a confidence
                pass
            else:
                # new absolute threshold based on frame height
                threshold = np.floor(frame.size[0] * 0.115)

                if confidence < threshold:  # low confidence compared to absolute threshold
                    # retry stabilization with the previous frames to see if the confidence improves
                    print(
                        f"Confidence too low (conf={confidence:.3f}, "
                        f"threshold={threshold:.3f}). Trying to fix:"
                    )
                    for ref_frame in reversed(framebuffer[:-1]):
                        x, y, conf = stabilize(frame, ref_frame, True)
                        print(
                            f"\tPrevious confidence: {confidence:.3f} "
                            f"\tNew confidence: {conf:.3f}"
                        )
                        if conf > 1.5 * threshold:
                            base_location = framebuffer.index(ref_frame)
                            additional_shift_x = x
                            additional_shift_y = y
                            print(
                                f"\tNew reference set at index {base_location} "
                                f"with confidence {conf:.3f}"
                            )
                            break
                    else:
```

```
                        # if the loop finished without breaking
                        print("Confidence was very low. Could not recover")

        elif msg.message_type == mt.Done:
            break  # break out of loop to finalize video

        else:
            raise ValueError(f"Message type {msg.message_type} is not implemented")
        # ============================== END OF LOOP ==============================

    # Start preprocessing
    print("DONE received, starting preprocessing steps")
    if not check_stabilization_data(framebuffer):
        print(
            "WARNING: Not all contained stabilization data. defaulted those frames to (0, 0).\n"
            "Video generator will still run, although one or more frames could look off"
        )

    if not check_same_size(framebuffer):
        raise RuntimeError("Before frame padding not all frames are the same size")

    framebuffer = pad_frames(framebuffer, mode=manager_data["Out-painting"])

    if not check_same_size(framebuffer):
        raise RuntimeError("After frame padding not all frames are the same size")

    speedup_factor = float(manager_data["fps"]) * int(manager_data["interval"])
    generate_annotation(framebuffer, speedup_factor)

    # make video
    print("Preprocessing done. Starting video generation")
    video_name = manager_data["File name"] + "." + manager_data["Extention"]
    out_path = Path(manager_data["output_location"]) / video_name
    # out_path = Path("out") / "demo.mp4"  # .mkv voor lossless, .mp4 voor compressed
    codec = extract_codec(out_path)
    out_path.parent.mkdir(parents=True, exist_ok=True)
    ok = frames_to_video(
        framebuffer,
        str(out_path),
        float(manager_data["fps"]),
        codec,
    )
    print(f"Video generator returned {ok}")


# This is used for the standalone functionality. You can load the images into the main loop.
if __name__ == "__main__":
    if not len(sys.argv) > 1:
        raise RuntimeError("In stand-alone mode, ensure to call the function with exactly one
        ↪  parameter that contains" +
                           " a string to a folder with data")
    image_folder = sys.argv[1]
    commclass = FolderCommunication(image_folder)
    analysis_main(commclass)
```

F.4. maintestfromfolder.py

```
"""
Demo and test script for framebuffer generation, padding, annotation, and video output.

This module provides:
- A simple `__main__` block that:
    * Generates a framebuffer from a test dataset.
    * Pads frames.
    * Adds annotations.
    * Writes the result to a demo video file.
```

```
- An `old_main` function that demonstrates how to:
    * Use `FolderCommunication` to iterate over messages.
    * Load frames from image paths.
    * Inspect and display them using matplotlib.
"""

import sys
from pathlib import Path

import matplotlib.pyplot as plt

from communication import FolderCommunication, read_manager_data
from frame import Frame
from messages import message_types
from preprocessing import (
    framebuffer_generator_map,
    pad_frames,
    DEBUG_framebuffer_generator,
    generate_annotation,
)
from video import frames_to_video, extract_codec


# test
if __name__ == "__main__":
    FB = framebuffer_generator_map("testing_data/200nm")
    FB = pad_frames(FB, mode="mean_outpaint")
    generate_annotation(FB, 30.0)
    out_path = Path("out") / "demo_200nm.mp4"  # .mkv voor lossless, .mp4 voor compressed
    codec = extract_codec(out_path)  # Dit zoekt de bijpassende codec voor de file extensie die je
    ↪  hebt gegeven.
    out_path.parent.mkdir(parents=True, exist_ok=True)
    ok = frames_to_video(FB, str(out_path), 1, codec)
    print("Wrote:", out_path, "OK:", ok)


def old_main() -> None:
    """
    Legacy test entry point for manual debugging.

    This function:
    - Checks command-line arguments for a (future) communication-method selector.
    - Demonstrates use of `FolderCommunication` to read messages from a folder.
    - Loads images into `Frame` objects and prints or displays them.
    - Finally, generates a debug framebuffer and displays each frame with matplotlib.

    Note:
        This function is not used in the standard execution path and primarily
        exists for interactive testing and development.
    """
    if len(sys.argv) > 1:
        # Here you could check which communication method should be started
        pass
    else:
        exit("Please enter arguments for selecting communication method")

    # Testing for frame class (example code, currently commented out):
    # frame = Frame()
    # file_loc = "20240630_201008.jpg"
    # frame.load_image(file_loc)
    # plt.imshow(frame.image, cmap='gray')
    # plt.show()
    # img = frame.get_image()
    # img = np.rot90(img, k=3)
    # frame.update_image(img)
    # plt.imshow(frame.image, cmap='gray')
```

```
    # plt.show()
    # print(frame.size)

    # Testing for FolderCommunication class
    comm = FolderCommunication("temp_pics")
    if not comm.connect():
        # if comm.connect returns true, connection was successful
        raise FileNotFoundError(
            f"Failed to connect to folder {comm.folder_location}. Folder is empty"
        )

    # get the filenames, and display them one by one
    while comm.message_available("ANALYSIS"):
        message = comm.receive("ANALYSIS")
        if message.message_type == message_types.ImageLocation:
            file_loc = message.message_content
            frame = Frame()
            frame.load_image(file_loc)
            # plt.imshow(frame.image, cmap='gray')
            print(file_loc)
            # plt.show()
        elif message.message_type == message_types.ManagerInfo:
            manager_data = read_manager_data(message)
            print(manager_data)
        elif message.message_type == message_types.ControlRefocused:
            print("Refocused")
        elif message.message_type == message_types.ControlMoved:
            print("Moved")
        else:
            raise RuntimeError(f"Unhandled message type: {message.message_type}")

    framebuffer = pad_frames(DEBUG_framebuffer_generator(), "same")
    for frame in framebuffer:
        plt.imshow(frame.image, cmap="gray")
        plt.show()
```

F.5. messages.py

```
"""
Message types and Message class used for communication between CONTROL and ANALYSIS.

This module defines:
- `message_types`: an Enum describing the supported message kinds.
- `Message`: a small container class that validates message payloads based
  on their type and records the intended receiver.
"""

from __future__ import annotations

from enum import Enum, EnumMeta
from typing import Union, List, Tuple, Any


class MessageType(Enum):
    ManagerInfo = 0
    ControlMoved = 1
    ImageLocation = 2
    MoveCommand = 3
    Cancel = 4
    Done = 5


# optional alias to mimic your old name
message_types = MessageType
```

```python
MessageData = Union[List[str], bool, str, Tuple[float, float]]


class Message:
    """
    Encapsulates a typed message passed between components.

    Each message has:
    - a `message_type` from `message_types`,
    - a `message_content` payload whose type depends on `message_type`,
    - a `receiver` indicating which subsystem it is intended for
      (currently 'CONTROL' or 'ANALYSIS').

    Data type expectations per message type:
        ManagerInfo   -> list[str]
        ControlMoved  -> bool
        ImageLocation -> str (filepath)
        MoveCommand   -> tuple[float, float]
        Cancel        -> bool
        Done          -> bool
    """

    def __init__(self, message_type: MessageType, data: MessageData, receiver: str) -> None:
        # subfunction for this function's errors
        def error(
            the_message_type: MessageType,
            requested_type: Union[type, str],
            received_type: type,
        ) -> None:
            raise TypeError(
                f"{the_message_type} should receive data as {requested_type}. "
                f"Got {received_type} instead"
            )

        if message_type in message_types:
            self.message_type: MessageType = message_type
        else:
            raise ValueError(f"Message type {message_type} is not a valid message type")

        # Check if data is of the correct (defined) type. The definitions are:
        if message_type == message_types.ManagerInfo:
            if not (
                isinstance(data, list)
                and all(isinstance(line, str) for line in data)
            ):
                error(message_type, list, type(data))

        elif message_type == message_types.ControlMoved:
            if not isinstance(data, bool):
                error(message_type, bool, type(data))

        elif message_type == message_types.ImageLocation:
            if not isinstance(data, str):
                error(message_type, str, type(data))

        elif message_type == message_types.MoveCommand:
            if not (
                isinstance(data, tuple)
                and len(data) == 2
                and all(isinstance(i, float) for i in data)
            ):
                error(message_type, "a tuple of two floats", type(data))

        elif message_type == message_types.Cancel:
            if not isinstance(data, bool):
                error(message_type, bool, type(data))
```

```
        elif message_type == message_types.Done:
            if not isinstance(data, bool):
                error(message_type, bool, type(data))

        else:  # Catch all, should not happen
            # as the ValueError should have been raised if the message type was wrong.
            # Can only happen when you have defined the message type, but not the corresponding data
            ↪  type.
            raise RuntimeError(
                "The message type check was valid, but the message type is not "
                "implemented in the data type check"
            )

        # if all checks were good, then we get here and we can assign the data.
        self.message_content: MessageData = data

        if receiver in ["CONTROL", "ANALYSIS"]:
            self.receiver: str = receiver
        else:
            raise RuntimeError(f"Receiver {receiver} is not an allowed receiver")

    def __str__(self) -> str:
        return f"{self.message_type}: {self.message_content}"

    def __repr__(self) -> str:
        return str(self)
```

F.6. outpainter.py

```
"""
Padding utilities for images.

This module provides a function to pad the borders of a 2D image using
local mean values of neighboring pixels, intended for use on shifted or
stabilized microscopy images.
"""

from typing import Optional

import numpy as np
from scipy.ndimage import convolve  # snelle convolutie in C


def pad_with_local_mean(
    image: np.ndarray,
    top: int,
    bottom: int,
    left: int,
    right: int,
    max_iter: Optional[int] = None,
) -> np.ndarray:
    """
    Pad the borders of a 2D image using local mean values of neighboring pixels.

    Functie om de randen van de geshifte images te vullen met locale mean
    values van pixels.

    Assumptions:
        - `image` is 2D (grayscale), e.g. from a .TIFF file.
        - Padding is specified in pixels for top, bottom, left, and right.
        - Iterative diffusion from known pixels into unknown regions is used,
          averaging over 3x3 neighborhoods until borders are filled or
          `max_iter` is reached.

    Args:
```

```
        image:
            Input 2D numpy array (grayscale image).
        top:
            Number of pixels to pad at the top.
        bottom:
            Number of pixels to pad at the bottom.
        left:
            Number of pixels to pad on the left.
        right:
            Number of pixels to pad on the right.
        max_iter:
            Optional maximum number of diffusion iterations. If None, a
            reasonable upper bound is computed based on padding size.

    Returns:
        A new 2D numpy array (uint8) with padded borders filled using
        local mean values.
    """
    # Werk in float32 voor NaN-waarden en gemiddelde berekeningen.
    img = image.astype(np.float32)

    h, w = img.shape
    h_pad = h + top + bottom
    w_pad = w + left + right

    # Maak een nieuwe array met NaN: hiermee markeren we onbekende (in te vullen) pixels.
    padded = np.full((h_pad, w_pad), np.nan, dtype=np.float32)

    # Plaats de originele image in het midden van de nieuwe array.
    padded[top: top + h, left: left + w] = img

    # Boolean mask: True op plekken die nog gevuld moeten worden.
    missing = np.isnan(padded)

    # 3x3 kernel om de som / count van buurpixels te berekenen (8-connected neighbors).
    kernel = np.ones((3, 3), dtype=np.float32)

    # Bepaal maximaal aantal iteraties: genoeg om de buitenste rand te bereiken.
    # Elke iteratie kan de bekende regio ~1 pixel naar buiten uitbreiden.
    if max_iter is None:
        max_band = max(top + bottom, left + right)
        max_iter = max_band + 2  # klein beetje marge

    for _ in range(max_iter):
        # Als er geen missende pixels meer zijn, kunnen we stoppen.
        if not np.any(missing):
            break

        # Vervang NaN door 0 om de som van buurwaarden te kunnen pakken.
        padded_zero = np.nan_to_num(padded, nan=0.0)

        # Som van alle 3x3 buurpixels.
        neigh_sum = convolve(padded_zero, kernel, mode="constant", cval=0.0)

        # Aantal geldige buurpixels (niet-NaN).
        valid_mask = ~np.isnan(padded)
        neigh_count = convolve(
            valid_mask.astype(np.float32),
            kernel,
            mode="constant",
            cval=0.0,
        )

        # Pixels die nog missend zijn maar moeten een geldige neighbour pixel (niet NaN) hebben.
        to_update = missing & (neigh_count > 0)
```

```
        if not np.any(to_update):
            continue

        # Vul missende pixels met het gemiddelde van de bekende neighbours.
        padded[to_update] = neigh_sum[to_update] / neigh_count[to_update]

        # Deze pixels zijn nu ingevuld.
        missing[to_update] = False

    # Op dit punt zou alles gevuld moeten zijn met lokale gemiddelden. Deze code fungeert als backup.
    if np.any(np.isnan(padded)):
        # In principe zou dit niet moeten gebeuren, maar we doen een simpele fallback:
        # vul overgebleven NaNs met de dichtstbijzijnde bekende waarde (nearest).
        # Dit is nog steeds lokaal, geen globale image-mean.
        known = ~np.isnan(padded)
        # Waar niets bekend is, vul gewoon met 0.
        if not np.any(known):
            padded[:] = 0
        else:
            # Als er toch nog NaNs zijn, vul ze met 0.
            padded[np.isnan(padded)] = 0.0

    # Cast terug naar uint8 image (0-255).
    padded = np.clip(padded, 0, 255).astype(np.uint8)

    return padded
```

F.7. preprocessing.py

```
"""
Preprocessing utilities for SEM frames.

This module provides:
- Helpers to build a framebuffer from a directory of images.
- Padding logic to align frames based on stabilization shifts.
- Sanity checks for stabilization data and frame sizes.
- Annotation generation (text + scale bar) rendered into the frames.
"""

from pathlib import Path
from typing import List, Tuple

from math import floor, log10

import numpy as np
from numpy.ma.extras import average
from PIL import Image, ImageDraw, ImageFont

try:
    from Analysis.frame import Frame
    from Analysis.stabilization import stabilize
    from Analysis.outpainter import pad_with_local_mean
except ModuleNotFoundError as e:
    if e.name in (
        "Analysis",
        "Analysis.frame",
        "Analysis.stabilization",
        "Analysis.outpainter",
    ):
        from frame import Frame
        from stabilization import stabilize
        from outpainter import pad_with_local_mean
    else:
        raise
```

```python
def framebuffer_generator_map(directory: str = "testing_data/200nm") -> List[Frame]:
    """
    THIS WAS USED FOR TESTING PURPOSES.
    Load all .TIF images from the given directory into a framebuffer.

    Stabilization is performed against the previous frame using `stabilize`.

    Args:
        directory: Path to the directory containing .TIF images.

    Returns:
        List of Frame objects with `image` and `stabilization_data` filled.

    Raises:
        FileNotFoundError: If no .TIF images are found in the directory.
    """
    dir_path = Path(directory)
    # simple: only .TIF, can be extended to other extensions if needed
    paths = sorted(dir_path.glob("*.TIF"))

    if not paths:
        raise FileNotFoundError(f"No .TIF images found in: {dir_path}")

    framebuffer: List[Frame] = []
    first = paths[0]
    prev = Frame()
    prev.load_image(str(first))
    prev.stabilization_data = (0, 0)

    for p in paths:
        f = Frame()
        f.load_image(str(p))
        if len(framebuffer) == 0:
            f.stabilization_data = (0, 0)
        else:
            # default; already stabilized/preprocessed. Here the stabilization
            # module should be invoked.
            x, y, _ = stabilize(f, prev, True)
            f.stabilization_data = (x, y)
        framebuffer.append(f)

    return framebuffer


def DEBUG_framebuffer_generator() -> List[Frame]:
    """
    THIS WAS USED FOR TESTING PURPOSES.
    Generate a small debug framebuffer with hardcoded image paths and
    artificial stabilization patterns (circle around center).
    """
    frame1 = Frame()
    frame2 = Frame()
    frame3 = Frame()
    frame4 = Frame()
    frame5 = Frame()
    frame6 = Frame()
    frame7 = Frame()

    frame1.load_image("testing_data/base/1.TIF")
    frame2.load_image("testing_data/base/2.TIF")
    frame3.load_image("testing_data/base/3.TIF")
    frame4.load_image("testing_data/base/4.TIF")
    frame5.load_image("testing_data/base/5.TIF")
    frame6.load_image("testing_data/base/6.TIF")
    frame7.load_image("testing_data/base/7.TIF")
```

```
    frame1.stabilization_data = (1, 0)
    frame2.stabilization_data = (1, 1)
    frame3.stabilization_data = (0, 1)
    frame4.stabilization_data = (-1, 1)
    frame5.stabilization_data = (-1, 0)
    frame6.stabilization_data = (-1, -1)
    frame7.stabilization_data = (0, -1)

    fb = [frame1, frame2, frame3, frame4, frame5, frame6, frame7]
    return fb


def pad_frames(framebuffer: List[Frame], mode: str) -> List[Frame]:
    """
    Pad all frames so that they align based on their stabilization_data.

    The padding amounts are computed from the min/max of the x/y shifts in
    the framebuffer, and different padding modes are supported.

    Args:
        framebuffer: List of Frame objects, each with stabilization_data set.
        mode: Padding mode, one of:
            'black'        -> pad with 0, simple
            'white'        -> pad with 255, simple.
            'same'         -> extend edge pixels, not great..
            'mean'         -> numpy mean padding, not great..
            'mean_outpaint'-> custom local-mean outpainting, in most cases this will give the best
            ↪  result.
            'blur'         -> A different name for mean_outpaint that was adopted by the Control
            ↪  software.
                              This function supports both names

    Returns:
        The same framebuffer list, with each Frame updated in-place.

    Raises:
        ValueError: If mode is invalid.
    """
    modes = ["black", "white", "same", "mean", "mean_outpaint", "blur"]
    mode = mode.lower()
    if mode not in modes:
        raise ValueError(f"{mode} is not a valid input value, choose from: {modes}")

    x_shifts = [frame.stabilization_data[0] for frame in framebuffer]
    y_shifts = [frame.stabilization_data[1] for frame in framebuffer]
    x_min = min(x_shifts)
    x_max = max(x_shifts)
    y_min = min(y_shifts)
    y_max = max(y_shifts)

    for frame in framebuffer:
        top = int(frame.stabilization_data[1] - y_min)
        bottom = int(y_max - frame.stabilization_data[1])
        left = int(frame.stabilization_data[0] - x_min)
        right = int(x_max - frame.stabilization_data[0])

        img = frame.get_image()
        pad_width = ((top, bottom), (left, right))

        if mode == "black":
            padded_frame = np.pad(
                img,
                pad_width,
                mode="constant",
                constant_values=0,
            )
```

```
        elif mode == "white":
            padded_frame = np.pad(
                img,
                pad_width,
                mode="constant",
                constant_values=255,
            )
        elif mode == "same":  # 'extend' could be an alternative name
            padded_frame = np.pad(
                img,
                pad_width,
                mode="edge",
            )
        elif mode == "mean":
            padded_frame = np.pad(
                img,
                pad_width,
                mode="mean",
            )
        elif mode == "mean_outpaint" or mode == "blur":
            # Custom: fills padded bands using local neighbor mean
            padded_frame = pad_with_local_mean(img, top, bottom, left, right)
        else:
            raise ValueError(f"No logic is implemented for mode {mode}")

        # padded_frame might be referenced before assignment: this warning is not
        # actually true, since invalid modes cause a ValueError before here.
        frame.update_image(padded_frame)

    # now all frames are updated in place, we can return the framebuffer
    return framebuffer


def check_stabilization_data(fb: List[Frame], fill_nones: bool = True) -> bool:
    """
    Check if all frames contain stabilization data.

    Args:
        fb: List of Frame objects.
        fill_nones: If True, any missing stabilization_data will be set to
            (0, 0). The function will still return False in that case to mark
            that there were frames without data.

    Returns:
        True if all frames had stabilization_data initially, False otherwise.
    """
    return_bool = True  # assume correct, will be set false if not
    for frame in fb:
        if frame.stabilization_data is None:
            return_bool = False
            if fill_nones:
                frame.stabilization_data = (0, 0)
    return return_bool


def check_same_size(fb: List[Frame]) -> bool:
    """
    Check whether all frames have the same size.

    Args:
        fb: List of Frame objects.

    Returns:
        True if all frames have identical (height, width), otherwise False.

    Raises:
```

```
        ValueError: If the framebuffer list is empty.
    """
    if not fb:
        raise ValueError("Framebuffer is empty")

    h0, w0 = fb[0].size[:2] if hasattr(fb[0], "size") else fb[0].image.shape[:2]
    # Check from the second element (first is the basis)
    for f in fb[1:]:
        h, w = f.size[:2] if hasattr(f, "size") else f.image.shape[:2]
        if (h, w) != (h0, w0):
            return False
    return True


def generate_annotation(fb: List[Frame], speedup_factor: float) -> None:
    """
    Render textual annotations and scale bars into each frame.

    The annotation shows:
        - Speed (speedup_factor)
        - Acceleration voltage (kV)
        - Spot size
        - Magnification
        - Working distance
        - Scale bar with human-friendly units

    The overlay is appended to the bottom of each frame as extra rows.
    """
    # Use SD height as reference to scale everything else
    BASE_FRAME_HEIGHT = 484.0  # reference: SD height
    BASE_FONT_SIZE = 14
    BASE_COL_PADDING = 6
    BASE_ROW_PADDING = 6

    def round_first_two_digits(val: float) -> int:
        if val == 0:
            return 0

        sign = 1 if val > 0 else -1
        v = abs(val)

        order = floor(log10(v))
        factor = 10 ** (order - 1)
        rounded = round(v / factor)
        return int(sign * rounded * factor)

    def text_size(txt: str) -> Tuple[int, int]:
        """Measure text width/height for the current font."""
        _img = Image.new("L", (200, 200))  # create large enough img
        _draw = ImageDraw.Draw(_img)
        bbox = _draw.textbbox((0, 0), txt, font=font)
        _w = bbox[2] - bbox[0]
        _h = bbox[3] - bbox[1]
        return _w, _h

    for frame in fb:
        frame_height = frame.size[0]
        frame_width = frame.size[1]

        # Scale factor relative to SD height; clamp to reasonable bounds
        scale = frame_height / BASE_FRAME_HEIGHT if BASE_FRAME_HEIGHT else 1.0
        scale = max(0.75, min(scale, 3.0))

        # Scaled typography and padding
        font_size = max(10, int(round(BASE_FONT_SIZE * scale)))
        col_padding = max(4, int(round(BASE_COL_PADDING * scale)))
```

```
    row_padding = max(4, int(round(BASE_ROW_PADDING * scale)))

    vector = frame.fei_info.get("Vector")
    databar_data = frame.fei_info.get("DatabarData")
    voltage = format(vector["HighTension "] / 1000, ".1f") + " kV"
    spot = format(databar_data["flSpot "], "0.1f")
    zoom = str(round_first_two_digits(vector["Magnification "])) + "x"
    wd = format(databar_data["flWD "], "0.1f")
    speedup = str(speedup_factor) + "x"

    parameter_template = [
        ["Speed", "AccV", "Spot", "Magn", "WD"],
        [speedup, voltage, spot, zoom, wd],
    ]

    # Use scaled font
    try:
        font = ImageFont.truetype(
            "Analysis/fonts/arial-geo-webfont.ttf",
            font_size,
        )
    except OSError:
        font = ImageFont.truetype(
            "fonts/arial-geo-webfont.ttf",
            font_size,
        )

    # determine column width
    num_cols = max(len(row) for row in parameter_template)
    col_widths = [0] * num_cols
    row_heights = [0] * len(parameter_template)

    for r, row in enumerate(parameter_template):
        h_max = 0
        for c, text in enumerate(row):
            w, h = text_size(text)
            col_widths[c] = max(col_widths[c], w)
            h_max = max(h_max, h)
        row_heights[r] = max(row_heights[r], h_max)

    # pad columns further (scaled)
    col_widths = [w + col_padding for w in col_widths]
    row_heights = [h + row_padding for h in row_heights]

    # Overlay spans full image width; height from rows + padding
    width = frame_width
    height = sum(row_heights) + 2 * row_padding
    img = Image.new("L", (width, height))
    draw = ImageDraw.Draw(img)

    y = row_padding  # Starting y coord of annotation
    y_scalebar = y
    x_scalebar = 0

    for r, row in enumerate(parameter_template):
        x = col_padding  # Starting X coord of annotation
        for c, text in enumerate(row):
            draw.text((x, y), text, fill="white", font=font)
            x += col_widths[c]
            x_scalebar = max(x, x_scalebar)
        y += row_heights[r]

    # Scale bar thickness scales with text height (already scaled)
    bar_height = max(1, int(round(average(row_heights) / 2)))

    # Scale bar length limits relative to frame width
```

```
        scalebar_space = frame_width - x_scalebar
        scalebar_max_l = min(0.4 * frame_width, scalebar_space * 0.9)
        scalebar_min_l = 0.17 * frame_width

        pixel_size_mm = frame.compute_pixel_size_mm()
        if pixel_size_mm is None:
            pixel_size_mm = float(1)  # fallback

        scalebar_length, scale_label = compute_scale_bar(
            pixel_size_mm,
            int(scalebar_min_l),
            int(scalebar_max_l),
        )

        x_scalebar_unit = x_scalebar + scalebar_length + col_padding
        y_scalebar_unit = y_scalebar
        draw.text(
            (x_scalebar_unit, y_scalebar_unit),
            scale_label,
            fill="white",
            font=font,
        )

        img_overlay = np.array(img)

        # Draw the scale bar (vertical ticks + horizontal bar)
        img_overlay[y_scalebar: y_scalebar + 2 * bar_height, x_scalebar] = 255
        img_overlay[
            y_scalebar: y_scalebar + 2 * bar_height,
            x_scalebar + scalebar_length,
        ] = 255
        img_overlay[
            y_scalebar + bar_height,
            x_scalebar: x_scalebar + scalebar_length,
        ] = 255

        img_frame = frame.get_image()
        new_img = np.vstack((img_frame, img_overlay))
        frame.update_image(new_img)


def compute_scale_bar(
    pixel_length_mm: float,
    min_pixels: int = 50,
    max_pixels: int = 400,
) -> Tuple[int, str]:
    """
    Compute a 'nice' scale bar length in pixels and human-readable unit.

    Args:
        pixel_length_mm: Size of one pixel in mm.
        min_pixels: Minimum pixel length for the scale bar.
        max_pixels: Maximum pixel length for the scale bar.

    Returns:
        A tuple (scale_length_px, scale_label) where:
            scale_length_px: number of pixels for the scale bar.
            scale_label: human-readable label, e.g. '5 µm'.
    """
    # min and max bar sizes in mm
    min_mm = pixel_length_mm * min_pixels
    max_mm = pixel_length_mm * max_pixels

    # Select best unit
    if max_mm < 1e-3:  # < 1 µm
        unit = "nm"
```

```
        factor = 1e6  # mm  nm
    elif max_mm < 1:  # < 1 mm
        unit = "ţm"
        factor = 1e3  # mm  ţm
    else:
        unit = "mm"
        factor = 1.0

    # set min and max val in the chosen unit
    min_val = min_mm * factor
    max_val = max_mm * factor

    # choose a scale value
    nice_numbers = [1, 2, 5, 10, 20, 50, 100, 200, 500]
    candidates = [n for n in nice_numbers if min_val <= n <= max_val]

    if candidates:
        chosen_value = candidates[0]  # smallest "nice" number that fits
    else:
        # fallback: pick largest nice number below max_val
        candidates = [n for n in nice_numbers if n <= max_val]
        chosen_value = candidates[-1] if candidates else nice_numbers[0]

    # back to mm
    chosen_mm = chosen_value / factor

    # calculate pixel length
    scale_length_px = int(round(chosen_mm / pixel_length_mm))

    # Generate label
    scale_label = f"{chosen_value:g} {unit}"

    return scale_length_px, scale_label
```

F.8. stabilization.py

```
"""
Frame stabilization and correlation-confidence utilities.

This module provides:
- Phase-correlation based shift estimation between two images.
- A `stabilize` helper that computes relative shifts between frames.
- A `beamshift_needed` helper to decide when to request SEM beam shifts.
- Several confidence metrics (PSR, PAR, P2SR, sharpness) for correlation peaks.
"""

import numpy as np
from typing import Optional, Callable, Union, Tuple, cast

try:
    from Analysis.frame import Frame
except ModuleNotFoundError as e:
    if e.name in ("Analysis", "Analysis.frame"):
        from frame import Frame
    else:
        raise


def _center_crop_to_same_size(a: np.ndarray, b: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    """
    (This function should actually do nothing but its here as a failsafe).
    Center-crop two arrays to the same minimum (height, width).

    This is needed so that numpy's correlation / FFT functions can operate
    on arrays with identical shapes.
    """
```

```
    ha, wa = a.shape[:2]
    hb, wb = b.shape[:2]
    h, w = min(ha, hb), min(wa, wb)

    def crop(img: np.ndarray, h_: int, w_: int) -> np.ndarray:
        H, W = img.shape[:2]
        y0 = (H - h_) // 2
        x0 = (W - w_) // 2
        return img[y0:y0 + h_, x0:x0 + w_]

    return crop(a, h, w), crop(b, h, w)


def _hanning_window(h: int, w: int) -> np.ndarray:
    """
    Create a 2D Hanning window of size (h, w).

    This reduces the impact of pixels at the image borders on the correlation.
    """
    wy = np.hanning(h)
    wx = np.hanning(w)
    return np.outer(wy, wx)


def _phase_correlation_shift(
    ref: np.ndarray,
    img: np.ndarray,
    confidence_func: Optional[Callable[[np.ndarray, Tuple[int, int]], float]] = None,
) -> Tuple[float, float, Optional[float]]:
    """
    Determine (dx, dy) shift of `img` with respect to `ref` using phase correlation.

    Positive dy = downward shift, positive dx = rightward shift.

    Args:
        ref: Reference image array.
        img: Image array to be aligned to `ref`.
        confidence_func: Optional callable to compute a confidence metric from
            the correlation surface and peak location.

    Returns:
        Tuple (dx, dy, confidence) where dx and dy are floats. The confidence
        is None if no confidence_func was given.
    """
    # Ensure images have the same size via center crop.
    ref, img = _center_crop_to_same_size(ref, img)
    ref = ref.astype(np.float32)
    img = img.astype(np.float32)

    # DC-suppression and windowing to reduce edge effects.
    ref = ref - ref.mean()
    img = img - img.mean()
    win = _hanning_window(*ref.shape)
    ref *= win
    img *= win

    # FFTs
    F = np.fft.rfft2(ref)
    G = np.fft.rfft2(img)

    # Cross-power spectrum with protection against division by zero
    R = F * np.conj(G)
    denom = np.abs(R)
    denom[denom == 0] = 1e-12
    CPS = R / denom
```

```
    # Inverse FFT is the normalized cross-correlation.
    corr = np.fft.irfft2(CPS, s=ref.shape)

    # Find peak position
    max_pos = np.unravel_index(np.argmax(corr), corr.shape)
    peak_y, peak_x = int(max_pos[0]), int(max_pos[1])

    h, w = ref.shape
    # Wrap-around to signed shift.
    dy = peak_y if peak_y <= h // 2 else peak_y - h
    dx = peak_x if peak_x <= w // 2 else peak_x - w

    # Optional confidence computation
    if confidence_func is not None:
        confidence = confidence_func(corr, (peak_x, peak_y))
    else:
        confidence = None

    # Return the shift needed to align img to ref.
    return float(dx), float(dy), confidence


def stabilize(
    frame: Frame,
    ref_frame: Frame,
    round: bool,  # noqa: A002  (kept name to avoid API change)
) -> Tuple[Union[int, float], Union[int, float], Optional[float]]:
    """
    Compute the shift between `frame` and `ref_frame` using phase correlation.

    Args:
        frame: Frame to be stabilized.
        ref_frame: Reference Frame to align to.
        round: If True, shifts are rounded to nearest integers.

    Returns:
        Tuple (shift_x, shift_y, confidence) where shift_x and shift_y are
        either ints (if round=True) or floats, and confidence is the value
        returned by the confidence metric used (PSR here).
    """
    if frame.image is None or ref_frame.image is None:
        raise ValueError("Beide frames moeten geladen zijn (frame.image is None).")

    # measure displacement (dx, dy) of current frame w.r.t. reference
    dx, dy, conf = _phase_correlation_shift(
        ref_frame.get_image(),
        frame.get_image(),
        confidence_psr,
    )

    if round:
        shift_x: Union[int, float] = int(np.round(dx))
        shift_y: Union[int, float] = int(np.round(dy))
    else:
        shift_x, shift_y = dx, dy

    # Note: we no longer directly write stabilization_data here so that
    # confidence logic can safely inspect the data first.
    if conf is not None:
        print(conf)

    return shift_x, shift_y, conf


def beamshift_needed(frame: Frame, threshold_percentage: int) -> bool:
    """
```

```
    Decide whether a beam shift is required based on stabilization data.

    Args:
        frame: Frame with stabilization_data set.
        threshold_percentage: Threshold of allowed shift as a percentage of
            half the frame size (100% = half the image).
            We define it like this because the max the center pixel can shift is half the image.

    Returns:
        True if either x or y shift exceeds its respective threshold, False otherwise.
    """
    percentage = threshold_percentage / 100.0

    # cast for type checkers; behavior unchanged if attributes are None
    size = cast(Tuple[int, int], frame.size)
    x_shift, y_shift = cast(Tuple[float, float], frame.stabilization_data)

    x_shift_max = int(0.5 * size[1] * percentage)  # 100% is half the image
    y_shift_max = int(0.5 * size[0] * percentage)

    # if one of these shifts goes over these thresholds, return True
    return x_shift > x_shift_max or y_shift > y_shift_max


def confidence_psr(corr: np.ndarray, peak: Tuple[int, int], window: int = 5) -> float:
    """
    Peak-to-Sidelobe Ratio (PSR) confidence metric.

    Args:
        corr: Correlation surface.
        peak: (x, y) coordinates of the peak.
        window: Half-size of the window to zero around the peak when
            computing sidelobe statistics.

    Returns:
        PSR value: (peak_value - mean_sidelobes) / std_sidelobes
    """
    px, py = peak
    corr_copy = corr.copy()
    corr_copy[
        max(py - window, 0): py + window + 1,
        max(px - window, 0): px + window + 1,
    ] = 0  # set the window around the peak to 0

    mean = corr_copy.mean()
    std = corr_copy.std() + 1e-12  # std in denominator to avoid division by zero.
    peak_val = corr[py, px]
    confidence = (peak_val - mean) / std

    return confidence


def confidence_par(corr: np.ndarray, peak: Tuple[int, int]) -> float:
    """
    Peak-to-Average Ratio (PAR) confidence metric.

    Args:
        corr: Correlation surface.
        peak: (x, y) peak coordinates.

    Returns:
        PAR = peak_value / mean(|corr|).
    """
    px, py = peak
    peak_val = corr[py, px]
    mean = abs(corr.mean())
```

```
    confidence = peak_val / (mean + 1e-12)

    return confidence


def confidence_p2spr(
    corr: np.ndarray,
    peak: Tuple[int, int],
    window: int = 5,
) -> float:
    """
    Peak-to-Second-Peak Ratio (P2SR) confidence metric.

    Args:
        corr: Correlation surface.
        peak: (x, y) peak coordinates.
        window: Half-size of the exclusion window around the main peak.

    Returns:
        P2SR = peak1 / peak2, where peak2 is the max outside the excluded window.
    """
    px, py = peak
    peak_val = corr[py, px]
    corr_copy = corr.copy()
    corr_copy[
        max(py - window, 0): py + window + 1,
        max(px - window, 0): px + window + 1,
    ] = 0
    second_peak = corr_copy.max()
    confidence = peak_val / (second_peak + 1e-12)

    return confidence


def confidence_sharpness(
    corr: np.ndarray,
    peak: Tuple[int, int],
    window: int = 3,
) -> float:
    """
    Peak sharpness metric based on local vs. total correlation energy.

    Args:
        corr: Correlation surface.
        peak: (x, y) peak coordinates.
        window: Half-size for the local window around the peak.

    Returns:
        Local energy ratio: sum(local^2) / sum(corr^2).
    """
    px, py = peak
    h, w = corr.shape
    y0 = max(py - window, 0)
    y1 = min(py + window + 1, h)
    x0 = max(px - window, 0)
    x1 = min(px + window + 1, w)
    local = corr[y0:y1, x0:x1]

    total_energy = float(np.sum(corr * corr)) + 1e-12
    local_energy = float(np.sum(local * local))
    confidence = local_energy / total_energy

    return confidence
```

F.9. stabilization_test.py

```
"""
Demo script for validating the phase-correlationbased stabilizer.

This script:
1. Loads an SEM image.
2. Applies a known synthetic shift using `scipy.ndimage.affine_transform`.
3. Wraps the original and shifted images in Frame objects.
4. Runs the stabilization function to estimate the shift.
5. Prints and visually verifies whether the estimated shift matches the true shift.
"""

from __future__ import annotations
import numpy as np
from PIL import Image, ImageOps
from scipy.ndimage import affine_transform  # for synthetic shifting

from frame import Frame
from stabilization import stabilize


def to_pil(arr: np.ndarray) -> Image.Image:
    """
    Convert a numpy array to a grayscale PIL image.

    If the array is not uint8, it is normalized to the 0255 range.

    Args:
        arr: Numpy array representing an image.

    Returns:
        A grayscale PIL Image.
    """
    a = np.asarray(arr)
    if a.dtype != np.uint8:
        amin = float(np.min(a))
        rng = float(np.ptp(a))
        if rng == 0:
            a8 = np.zeros_like(a, dtype=np.uint8)
        else:
            a8 = ((a - amin) * (255.0 / rng)).astype(np.uint8)
    else:
        a8 = a
    return Image.fromarray(a8)


img = Image.open("testing_data/base/1.TIF")
img = ImageOps.grayscale(img)
base = np.asarray(img, dtype=np.float32)


true_dx, true_dy = 40, -30  # shift: x right, y down

# Identity matrix: affine transform with only translation
M = np.array([[1, 0], [0, 1]], dtype=float)

# Note: affine_transform uses (row, col)  (y, x) ordering.
shifted = affine_transform(
    base,
    M,
    offset=(-true_dy, -true_dx),  # negative because affine_transform applies reverse mapping
    order=1,
    mode="nearest",
)

to_pil(base).show(title="Base")
```

```
to_pil(shifted).show(title="Shifted")

f_ref = Frame()
f_ref.update_image(base.astype(np.uint8))

f_cur = Frame()
f_cur.update_image(shifted.astype(np.uint8))

dx, dy, _ = stabilize(f_cur, f_ref, round=True)

print("True shift:      ", true_dx, true_dy)
print("Estimated shift: ", dx, dy)
```

F.10. video.py

```
"""
Video writing utilities for SEM frame sequences.

This module provides helpers to:
- Normalize frame data to uint8, 3-channel images for OpenCV support.
- Resolve output file paths for videos.
- Choose a codec based on file extension.
- Convert a list of Frame objects into a video file using OpenCV.
"""

from __future__ import annotations

import os
from typing import List, Optional, Tuple

import numpy as np
import cv2  # Prefer OpenCV as backend over imageio.

try:
    from Analysis.frame import Frame
except ModuleNotFoundError as e:
    if e.name in ("Analysis", "Analysis.frame"):
        from frame import Frame
    else:
        raise

_HAS_CV2 = True


def _to_uint8(img: np.ndarray) -> np.ndarray:
    """
    Ensure the given image is in uint8 format (0255).

    If the input is not uint8, it is normalized to the full 0255 range.
    """
    if img.dtype == np.uint8:
        return img

    a = np.asarray(img)
    amin = np.min(a)
    rng = np.ptp(a)
    if rng == 0:
        return np.zeros_like(a, dtype=np.uint8)
    return ((a - amin) * (255.0 / rng)).astype(np.uint8)


def _ensure_3ch(img: np.ndarray) -> np.ndarray:
    """
    Ensure the image has 3 channels (BGR) for OpenCV.

    If input is 2D (grayscale) or single-channel, it is replicated into 3
```

```
    identical channels. Otherwise the image is returned unchanged.
    """
    if img.ndim == 2:
        return np.repeat(img[..., None], 3, axis=2)
    if img.ndim == 3 and img.shape[2] == 1:
        return np.repeat(img, 3, axis=2)
    return img


def _get_frame_size(frames: List[Frame]) -> Tuple[int, int]:
    """
    Get (height, width) of the first frame in a framebuffer.

    This assumes all frames are the same size.
    """
    h0, w0 = (
        frames[0].size[:2]
        if hasattr(frames[0], "size")
        else frames[0].image.shape[:2]
    )
    return h0, w0


def _resolve_out_path(out_path: Optional[str], default_name: str = "output.mp4") -> str:
    """
    Resolve a usable output file path from a possibly None or directory-like string.

    Args:
        out_path: None, a directory path, or a file path.
        default_name: Default filename to use if out_path is None or looks like a directory.

    Returns:
        An absolute file path string where the video should be written.
    """
    if not out_path:
        return os.path.abspath(os.path.join(".", default_name))

    norm_path = os.path.normpath(out_path)

    # Directory heuristics
    looks_like_dir = (
        norm_path.endswith(os.sep)
        or os.path.isdir(norm_path)
        or os.path.splitext(os.path.basename(norm_path))[1] == ""
    )

    if looks_like_dir:
        # ensure directory exists later; just return full file path now
        return os.path.abspath(os.path.join(norm_path, default_name))

    # Explicit file path
    return os.path.abspath(norm_path)


def extract_codec(out_path: "os.PathLike[str] | str") -> Optional[str]:
    """
    Choose a codec string based on the output path's file extension.

    For now:
        - '.mkv' -> 'FFV1' (lossless)
        - '.mp4' -> 'mp4v' (MPEG-4 part 2)

    Args:
        out_path: Path-like or string with a filename.

    Returns:
```

```
        Codec string understood by cv2.VideoWriter_fourcc, or None if the
        extension is not recognized.
    """
    # handle both Path-like and str:
    name = os.fspath(out_path)
    ext = name.split(".")[-1].lower()
    if ext == "mkv":
        return "FFV1"
    if ext == "mp4":
        return "mp4v"
    return None


def frames_to_video(
    framebuffer: List[Frame],
    out_path: Optional[str] = None,
    fps: float = 30.0,
    codec: str = "mp4v",
) -> bool:
    """
    Write a video file from preprocessed frames (same size, stabilized).

    Args:
        framebuffer: List of Frame objects with `.image` as np.ndarray
            (grayscale or 3-channel).
        out_path: File path OR directory OR None (defaults to ./output.mp4).
        fps: Frames per second for the output video.
        codec: Codec for OpenCV path (e.g., "mp4v", "FFV1", "XVID").

    Returns:
        True on success, False otherwise.
    """
    try:
        # Resolve output path and ensure directory exists
        out_file = _resolve_out_path(out_path, default_name="output.mp4")
        os.makedirs(os.path.dirname(out_file), exist_ok=True)

        # Validate sizes
        H, W = _get_frame_size(framebuffer)

        if _HAS_CV2:
            writer = cv2.VideoWriter(
                out_file,
                cv2.VideoWriter_fourcc(*codec),
                fps,
                (W, H),
                isColor=True,  # we expand grayscale to 3ch
            )
            if not writer.isOpened():
                raise RuntimeError(f"Failed to open VideoWriter for '{out_file}'.")

            for idx, f in enumerate(framebuffer):
                if getattr(f, "image", None) is None:
                    raise ValueError(f"Frame {idx} has no image data")
                frame8 = _to_uint8(f.image)
                frame3 = _ensure_3ch(frame8)
                writer.write(frame3)
            writer.release()
        else:
            raise NotImplementedError("imageio not supported yet")

        if not os.path.exists(out_file) or os.path.getsize(out_file) == 0:
            raise RuntimeError(f"Output '{out_file}' was not created or is empty.")

        return True
```

```
    except Exception as e:
        print(f"[frames_to_video] ERROR: {e}")
        return False
```

Bibliography

[1] Maria Cristina Tanzi et al. "Chapter 7 - Techniques of Analysis". In: *Foundations of Biomaterials Engineering*. Ed. by Maria Cristina Tanzi et al. Academic Press, 2019, pp. 393–469. DOI: https://doi.org/10.1016/B978-0-08-101034-1.00007-4. URL: https://www.sciencedirect.com/science/article/pii/B9780081010341000074.

[2] Yiming Wang et al. "Video stabilization: A comprehensive survey". In: *Neurocomputing* (2023), pp. 205–230. DOI: https://doi.org/10.1016/j.neucom.2022.10.008. URL: https://www.sciencedirect.com/science/article/pii/S092523122201270X.

[3] G. Wolberg et al. "Robust image registration using log-polar transform". In: *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*. 2000, 493–496 vol.1. DOI: 10.1109/ICIP.2000.901003.

[4] G. Balachandran et al. "A Review of Video Stabilization Algorithms". In: *2022 International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*. Nov. 2022, pp. 643–648. DOI: 10.1109/ICAISS55157.2022.10011015.

[5] *FEI XL30 SFEG Scanning Electron Microscope: Technical Specification*. Specification Sheet. Eindhoven, The Netherlands: FEI Company, n.d. URL: https://media-moov-co.s3.us-west-1.amazonaws.com/user_media/listingFile/n4ek4IIz0TSXElVmsIyzHGwp-x9L1vJhNBOC4smXZcw/mflz9UVhuPi0qivIwGlgplmHPyEaFd_hgmqR49oyqwk/ehSGVBqDgRSKSd1vO4tZmuBbYDdfSXa9_uJvNtgAGsU_XL30_SFEG.pdf.

[6] Adobe Developers Association. *TIFF Revision 6.0: Tagged Image File Format Specification*. Defines the TIFF file structure, metadata tags, and supported image data types. Adobe Systems Incorporated. 1992. URL: https://www.adobe.io/content/dam/udp/en/open/standards/tiff/TIFF6.pdf.

[7] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[8] Wissal Hassen et al. "Block Matching Algorithms for motion estimation". In: *2013 7th IEEE International Conference on e-Learning in Industrial Electronics (ICELIE)*. 2013, pp. 136–139. DOI: 10.1109/ICELIE.2013.6701287.

[9] Andrea Alfarano et al. "Estimating optical flow: A comprehensive review of the state of the art". In: *Computer Vision and Image Understanding* 249 (2024), p. 104160. DOI: https://doi.org/10.1016/j.cviu.2024.104160. URL: https://www.sciencedirect.com/science/article/pii/S1077314224002418.

[10] Barbara Zitová et al. "Feature Point Detection in Multiframe Images". In: (June 2000).

[11] C. D. Kuglin et al. "The Phase Correlation Image Alignment Method." In: *Proceeding of IEEE International Conference on Cybernetics and Society* (1975), pp. 163–165.

[12] Tanisha Bhatia et al. "Practical Approach for Implementation of Phase Correlation Based Image Registration in FPGA". In: *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*. 2018, pp. 1–4. DOI: 10.1109/ICCTCT.2018.8551117.

[13] Bharat B. Reddy et al. "An FFT-Based Technique for Translation, Rotation, and Scale-Invariant Image Registration". In: *IEEE Transactions on Image Processing* 5.8 (1996), pp. 1266–1271. DOI: 10.1109/83.506761.

[14] Xiaohua Tong et al. "Image Registration With Fourier-Based Image Correlation: A Comprehensive Review of Developments and Applications". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12.10 (2019), pp. 4062–4081. DOI: 10.1109/JSTARS.2019.2937690. URL: https://doi.org/10.1109/JSTARS.2019.2937690.

[15] B. V. K. Vijaya Kumar et al. "Performance Measures for Correlation Filters". In: *Applied Optics* 29.20 (1990), pp. 2997–3006. DOI: 10.1364/AO.29.002997. URL: https://doi.org/10.1364/AO.29.002997.

[16] Sharif Naser Makhadmeh et al. "Optimization methods for power scheduling problems in smart home: Survey". In: *Renewable and Sustainable Energy Reviews* 115 (2019), p. 5. DOI: https://doi.org/10.1016/j.rser.2019.109362. URL: https://www.sciencedirect.com/science/article/pii/S1364032119305702.

[17] Alex Clark et al. *Pillow (PIL Fork) Documentation*. https://pillow.readthedocs.io/. Accessed: 09-12-2025. 2015.

[18] D. M. S. Pinto et al. "Python-Microscope: High-performance control of arbitrarily complex and scalable bespoke microscopes". In: *Journal of Cell Science* (2021). DOI: 10.1242/jcs.258955. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC8520730/.

[19] Python Software Foundation. *Python Language Reference, version 3.8*. https://docs.python.org/3.8/. 2020.

[20] Rafael C. Gonzalez et al. *Digital Image Processing*. 4th ed. New York, NY: Pearson, 2018.

[21] Hassan Foroosh et al. "Extension of Phase Correlation to Subpixel Registration". In: *IEEE Transactions on Image Processing* 11.3 (2002), pp. 188–200. DOI: 10.1109/83.988953. URL: https://doi.org/10.1109/83.988953.

[22] Kazuya Takita et al. "High-Accuracy Subpixel Image Registration Based on Phase-Only Correlation". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A.8 (2003), pp. 1925–1934. URL: https://search.ieice.org/bin/summary.php?id=e86-a_8_1925.

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

[24] Douglas A. Lyon. "The Discrete Fourier Transform, Part 4: Spectral Leakage". In: *Journal of Object Technology* 8.7 (2009), pp. 23–34. URL: https://www.jot.fm/issues/issue_2009_09/article2.

[25] Ronald N. Bracewell. *The Fourier Transform and Its Applications*. 3rd ed. Boston, MA: McGraw-Hill, 2000.

[26] Alan V. Oppenheim et al. *Discrete-Time Signal Processing*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1999.

[27] Barbara Zitová et al. "Image Registration Methods: A Survey". In: *Image and Vision Computing* 21.11 (2003), pp. 977–1000. DOI: 10.1016/S0262-8856(03)00137-9. URL: https://doi.org/10.1016/S0262-8856(03)00137-9.

[28] Michael T. Snella. "Drift Correction for Scanning-Electron Microscopy". MA thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, 2010. URL: https://dspace.mit.edu/handle/1721.1/59703.

[29] P. Jin et al. "Correction of image drift and distortion in a scanning electron microscopy". In: *Journal of Microscopy* (2015), pp. 268–280. DOI: 10.1111/jmi.12293. URL: https://onlinelibrary.wiley.com/doi/10.1111/jmi.12293.

[30] S. Maraghechi et al. "Correction of scanning electron microscope imaging artifacts in a novel digital image correlation framework". In: *Experimental Mechanics* (Apr. 2019), pp. 489–516. DOI: 10.1007/s11340-018-00469-w.

[31] Mike Hayles. *The XL FEG/SFEG/SIRION Scanning Electron Microscope Operating Manual*. FEI Company. Mar. 2002.

[32] Philips Electron Optics and FEI. "Control and communication with external computers and programs". In: *XL Interfacing Handbook*. 2002.

[33] Microscopy Australia. *Scanning Electron Microscopy*. URL: https://myscope.training/pdf/MyScope_SEM.pdf.

[34] P. Cizmar et al. "Real-Time Scanning Charged-Particle Microscope Image Composition with Correction of Drift". In: *arXiv e-prints* (2009). eprint: 0910.0213. URL: https://arxiv.org/abs/0910.0213.

[35] INSEE Statistical Glossary. *Coefficient of Variation*. https://www.insee.fr/en/metadonnees/definition/c1463. Accessed: 2025-01-15.

[36] Barbara Illowsky et al. *Introductory Statistics*. Sections used: Central Limit Theorem; Coefficient of variation. OpenStax, 2018. URL: https://openstax.org/details/books/introductory-statistics.

[37] Lingen Liu et al. "Image drift compensation in scanning electron microscopy facilitated by an external scanning and imaging system". In: *Micron* (2025). DOI: https://doi.org/10.1016/j.micron.2025.103848. URL: https://www.sciencedirect.com/science/article/pii/S0968432825000666.

[38] Naresh Marturi et al. “Fast image drift compensation in scanning electron microscope using image registration”. In: *2013 IEEE International Conference on Automation Science and Engineering (CASE)*. 2013, pp. 807–812. DOI: 10.1109/CoASE.2013.6653936.

[39] Lei Zhang et al. “A Global Approach to Fast Video Stabilization”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.2 (2017), pp. 225–235. DOI: 10.1109/TCSVT.2015.2501941.

[40] Weiyue Zhao, Xin Li, Zhan Peng, Xianrui Luo, Xinyi Ye, Hao Lu and Zhiguo Cao. “Fast Full-frame Video Stabilization with Iterative Optimization”. In: *Computer Vision Foundation* (2023). DOI: 10.1109/ICCV51070.2023.02151. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=10378088.

[41] Marcos Roberto e Souza et al. “Rethinking two-dimensional camera motion estimation assessment for digital video stabilization: A camera motion field-based metric”. In: *Neurocomputing* (2023), p. 126768. DOI: https://doi.org/10.1016/j.neucom.2023.126768. URL: https://www.sciencedirect.com/science/article/pii/S0925231223008913.

[42] Steven W. Smith. “The Scientist and Engineer’s Guide to Digital Signal Processing”. In: *California Technical Publishing* 2.1999 (1997). Available at http://www.dspguide.com.

[43] Tony F. Chan et al. “Mathematical Models for Local Nontexture Inpaintings”. In: *SIAM Journal on Applied Mathematics* 62.3 (2002), pp. 1019–1043. DOI: 10.1137/S0036139900368844.

[44] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.