***Abstract*—**

I. Introduction

The scanning electron microscope (SEM) is a type of microscope that produces images using signals generated from the interaction between electrons and the surface under observation. Higher resolution can be achieved compared to the traditional optical microscope, since electrons have much lower wavelength than light. An SEM can have resolution lower than one nanometre, whereas that of an optical microscope is often limited to a few hundred nanometres. This has benefited a variety of fields. For example, scientists have been using the SEM to analyse the doping density in semiconductors [1] and to view changes in bacterial cells [2].

Fig. 1 shows the basic construction of an SEM. The electron gun generates an electron beam, which is transformed into an electron probe after passing through the condenser lens and objective lens. It is scanned across the specimen under the effect of the scanning coil. As a result of the interaction between the incident electrons and the specimen, some electrons are emitted from the specimen. These are called secondary electrons and are collected by the detector, which generates signals whose magnitude depend on the strength of the secondary electrons. The display unit produces one frame of image after each complete scan of the specimen.

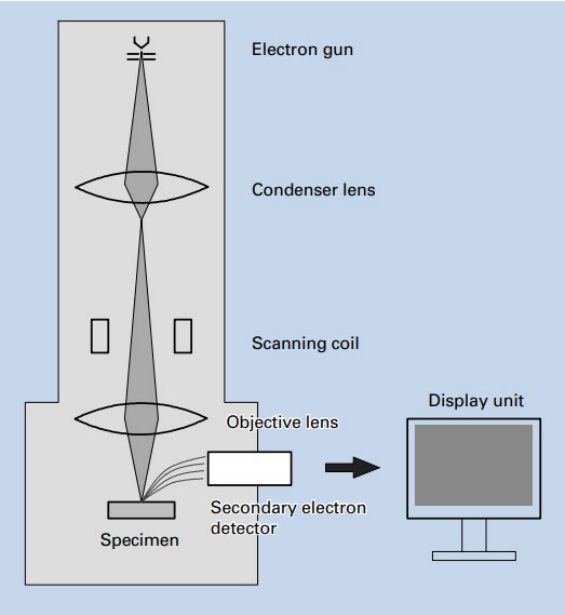


Fig. 1: Basic construction of an SEM [3].

Digital image analysis methods have been widely used in the field of SEM. For example, the fibre orientation distribution of nonwoven fabrics can be determined using fast Fourier transform (FFT) and Hough transform (HT) [4]. The FFT is an especially popular algorithm and many papers have been published on the use of it. However, due to its complexity and a lack of fast hardware, real-time analysis had largely been impossible or impractical in the past, i.e. it was only feasible to perform FFT on images off-line. In 1997, with an advanced central processing unit (CPU) — the Pentium Pro, it was only possible to achieve a refresh rate of about 0.6

frames per second for an 8-bit 1024 × 1024 image [5]. The enhancement of CPUs has enabled faster frame rates and more recently, the development of graphics processing unit (GPU) has taken image analysis to a different level.

The GPU was originally designed to efficiently manipulate data in order to accelerate the rendering of 3D images on displays. Depending on the position, pixels in a 3D image may require different processing to achieve effects such as lighting, blurring and fogging. This is achieved by breaking down the image into a massive number of fragments and processing each fragment individually. The processes happen independently but may share the same logical sequence of control, and this pattern is named single instruction multiple data (SIMD). A GPU consists of a large array of processing cores, with each of them using SIMD to process a block of fragments.

The characteristic of the GPU allows it to outperform central processing units (CPUs) when performing algorithms that manipulate data in parallel. Take the dot product between two vectors of size 1000 as an example, the processing cores on the GPU may be 100 times slower than the CPU, but if 1000 of them work in parallel and each computes the multiplication of one element from each vector, the overall speed will be 10 times faster. This has allowed computer vision programs that were previously impossible or computationally too expensive to build. For example, a wearable mediated reality device that make computer-generated information appear to the user as though it was anchored in the real world [6].

The goal of the project is to develop software tools based on fast computation provided by the GPU, to support interactive real-time diagnosis of SEM images for the purpose of assisting the operator or automating procedures, with a focus on the use of FFT.

II. Software aspects of the tools

A. Design principle

Considering that the work of the project may serve as the foundation stone for many further developments, the design of the software has a strong emphasis on readability and maintainability.

B. The selection of programming language

The SEM used for the project is made by Carl Zeiss AG, which provides an application programming interface (API) that can be used for controlling the SEM and grabbing images from it. The API supports C++ and thus makes it a possible programming language for the project. Being a relatively low-level compiled language makes

C++ extremely fast and useful for speed-critical applications. However, it also means that C++ has a complex syntax, which could significantly slow down development if the user does not have enough experience with it.

Considering the time scale of the project and to make development easier for people who will continue the work, Python is selected instead as the programming language, which has a much simpler syntax and is widely supported. With careful design, it has proved to be able to produce useful results despite the slower speed (see later sections).

C. Modules of the software

The software is highly modularised, which maximises readability and maintainability. This also makes it easier to translate the programs into C++ later if better performance is needed, since C++ is mostly used in an object-oriented manner. Fig. 2 shows the six main classes of the software.

SEM_API is a Python wrapper for the native SEM API written by Luyang Han, which allows the user to directly control the SEM in Python.

SemImage encapsulates variables and methods that are directly relevant to an image taken by the SEM, and *SemImageGrabber* is a helper class that helps obtain image data from the SEM and create instances of *SemImage* from them. Images can be obtained in two ways:

- From the SEM.
- From a local folder, which is helpful when the SEM is not available.

SemImageGrabber detects if the SEM is available and decides on which way to use.

SemTool handles the creation and rendering of the graphical user interface (GUI). Fig. 3 shows a screenshot of the control panel. The push buttons open a window for the corresponding plot and the radio buttons select algorithms (see section III and IV for detailed descriptions) to be performed on the image. The plots are shown on different windows, and can be opened and closed individually. This improves framerate by allowing the user to close unneeded windows, and also makes

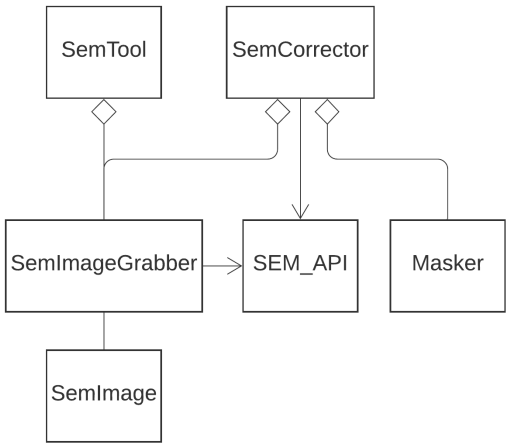


Fig. 2: Class diagram of the software in unified modelling language (UML).

it easier to add other plots later. Table I shows that the improvement is rather significant, as rendering windows is a major time-consuming process of the software. When only one window is opened, a refresh rate of about 16 frames per second is possible.

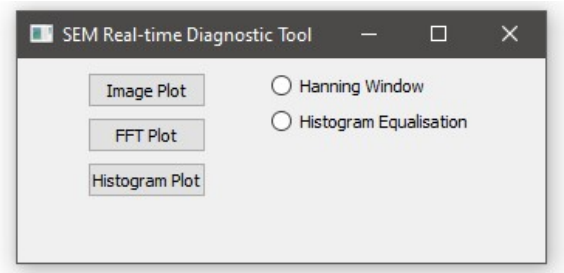


Fig. 3: Screenshot of the GUI.

TABLE I: Results of software framerate test

Number of windows opened	**Time taken to update each frame**
zero	15 ms
one	60 ms
three	160 ms

[a]The numbers are approximate.

[b]With 8-bit grey-scale 1024 × 768 input images.

[c]With Intel Core i7 and NVIDIA GeForce GTX 1060.

SemCorrector implements the automatic focusing and astigmatism correction algorithm and uses a helper class *Masker* to achieve fast computation (see section V for detailed descriptions). Due to the impact of the COVID-19 pandemic, the algorithm has not been fully tested, and is therefore not integrated to the GUI yet.

III. Real-time histogram equalisation for improving image contrast

There are many types of histograms in image processing. The grey level (brightness level) pixel intensity histogram, which plots the number of pixels of each grey value in the image, is the most relevant one for SEM images. This is because an SEM translates the energy of the secondary electrons directly into a grey level, colours do not exist in SEM images.

The algorithm for obtaining the histogram of an SEM image is given by

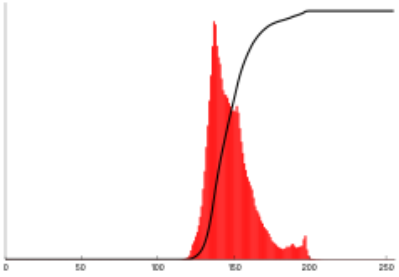
$$n_l = \frac{1}{P} \sum_p I(l_p = l), \tag{1}$$

where n_l is the normalised number of pixels of grey level l, P is the total number of pixels in the image and l_p is the grey level of the p_{th} pixel.

Fig. 4a shows an 8-bit grey-scale image, i.e. its depth of digitisation is 8-bit and it has 256 grey levels. Fig. 4b shows the histogram of the image and its integral. As can be seen in the histogram, most pixels are concentrated in the middle of the grey scale. This is reflected by the fact that the image is missing its highlights and shadows.

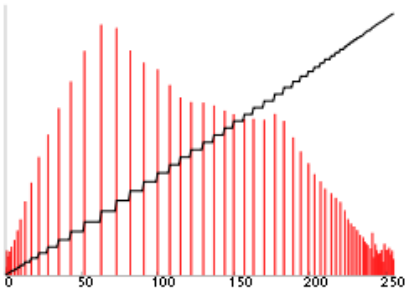
Histogram equalisation is a method for adjusting the distribution of pixel intensities of an image, in order to improve its overall

(a) Original image.



(b) Histogram of the original image.

(c) Histogram-equalised image.



(d) Histogram of the histogram-equalised image.

Fig. 4: Histogram and histogram equalisation.

contrast. Effectively, it is achieved by spreading out the more frequent intensity values. To perform histogram equalisation, first obtain the integral of the histogram using

$$s_l = \sum_{i=1}^{l} n_i,$$

where s_l is the value of the integral at grey level l. The integral spans from 0 to 1 as the histogram is normalised. Scale the integral by the maximum grey level and preform rounding to create the transform function

$$f_l = \lfloor s_l L \rfloor, \tag{2}$$

If a pixel in the original image has grey level l, it will have grey level f_l in the transformed image. For example, if all pixels in the original image are concentrated between grey level 100 to 200, the pixels of level 100 will become 0 in the new image while the pixels of level 200 will become 255 (assuming 8-bit image).

The result is more noticeable when the image has a low contrast, such as the one shown in Fig. 4a. The histogram-equalised version of it is given in Fig. 4c and the new histogram in Fig. 4d. More details are now visible as the contrast has been enhanced.

The most time-consuming part of histogram equalisation is to map all pixels in the original image to their new values using the transform function (2). This can be implemented using CPU code in one line:

```
newImage =
```

```
map(lambda x: f[x], image)
```

`f` is an array that serves as the transformation function, if the original pixel value is `x`, the new value should be `f[x]`. The `lambda` operator is a way to create small anonymous functions, which are throw-away functions and are only needed where they are created. It improves conciseness and readability by reducing code bloat. `lambda x: f[x]` creates an anonymous function that returns `f[x]` when given `x`, and `map()` uses this function to transform all pixels in `image`. This gives a time complexity of $O(N^2)$ which is not optimal. A smaller time complexity can be achieved using the GPU code:

```
map = cupy.ElementwiseKernel(
  'T x, raw T f', 'T xNew',
  'xNew = f[x]',
  'map'
)
newImage = map(image, f)
```

`cupy.ElementwiseKernel()` defines a kernel that the GPU uses to process the input data in parallel. Tests have shown that the use of GPU can increase the speed by 2 times, as summarised in Table II. In theory, the factor of speed improvement should be similar as that for calculating the histogram, since it also requires iterating through all pixels in the image. However, the code implementations introduce overheads which limit the improvement in speed.

TABLE II: Speed comparison between CPU and GPU

Algorithm	**Time taken using CPU**	**Time taken using GPU**
Apply Hanning window	16 ms	9 ms
Calculating histogram	30 ms	2 ms
Histogram equalisation	9 ms	5 ms
Fast Fourier transform	60 ms	8 ms

[b]For an 8-bit grey-scale 1024 × 768 input image.

[c]With Intel Core i7 and NVIDIA GeForce GTX 1060.

IV. Real-time fast Fourier transform for evaluating image focusing and astigmatism

V. Automatic focusing and astigmatism correction algorithm

VI. Conclusions

References

[1] T. Agemura and T. Sekiguchi, "Secondary electron spectroscopy for imaging semiconductor materials," 2018 International Symposium on Semiconductor Manufacturing (ISSM), Tokyo, Japan, 2018, pp. 1-3, doi: 10.1109/ISSM.2018.8651171.

[2] T. Cushnie, N. O'Driscoll and A. Lamb, "Morphological and ultrastructural changes in bacterial cells as an indicator of antibacterial mechanism of action," Cellular and Molecular Life Sciences, 2016, vol. 73, no. 23, pp. 4471-4492, doi: 10.1007/s00018-016-2302-2.

[3] "SEM A to Z", Jeol.co.jp, [Online], available: https://www.jeol.co.jp/en/applications/pdf/sm/sem_atoz_all.pdf. [Accessed: 18 May 2020].

[4] E. Ghassemieh, M. Acar and H. Versteeg, "Microstructural analysis of non-woven fabrics using scanning electron microscopy and image processing. Part 1: Development and verification of the methods," Proceedings of the Institution of Mechanical Engineers, Part L: Journal of Materials: Design and Applications, 2002, vol. 216, no. 3, pp. 199-207, doi: 10.1177/146442070221600305.

[5] A. Vladár, M. Postek and M. Davidson, "Image sharpness measurement in scanning electron microscopy-part II," Scanning, 2006, vol. 20, no. 1, pp. 24-34, doi: 10.1002/sca.1998.4950200104.

[6] J. Fung, F. Tang and S. Mann, "Mediated reality using computer graphics hardware for computer vision," Proceedings. Sixth International Symposium on Wearable Computers, Seattle, WA, USA, 2002, pp. 83-89, doi: 10.1109/ISWC.2002.1167222.