

Kristian Tveitstøl

Using EELS to measure the local conductivity in Aluminium

Master's thesis in Nanotechnology

Supervisor: Randi Holmestad

Co-supervisor: Ragnvald Mathiesen and Emil Frang Christiansen

June 2023

Kristian Tveitstøl

Using EELS to measure the local conductivity in Aluminium

Master's thesis in Nanotechnology

Supervisor: Randi Holmestad

Co-supervisor: Ragnvald Mathiesen and Emil Frang Christiansen

June 2023

Norwegian University of Science and Technology

Faculty of Natural Sciences

Department of Physics



Norwegian University of
Science and Technology

Abstract

The plasmonic properties of pure aluminium have been investigated using electron energy loss spectroscopy in order to reproduce its macroscopic electrical conductivity. Temperatures varying from room temperature to 500°C have been investigated with an energy resolution of $\lesssim 100$ meV. The plasmon energy was found to be ≈ 15 eV at room temperature, decreasing with approximately 0.5 meV K $^{-1}$ and had an asymmetric linewidth of ≈ 500 meV and 700 meV resulting in an underestimate of the conductivity with one order of magnitude. The findings suggest that the full width of half maximum broadens with increasing temperature, though no conclusive relation has been outlined.

Sammendrag

De plasmoniske egenskapene til ren aluminium har blitt undersøkt ved hjelp av elektronenergitapsspektroskopi med hensikt til å reprodusere den makroskopiske elektriske ledningsevnen til materialet. Dette har blitt gjennomført for temperaturer fra romtemperatur til 500 °C med en energiopløsning på $\lesssim 100$ meV. Plasmontoppen ble målt til å være sentrert ved omtrent 15 eV ved romtemperatur, sank med 0.5 meV K^{-1} , og hadde en asymmetrisk halvverdibredde på 500 og 700 meV, som medførte at metoden underestimerte ledningsevnen med en størrelsesorden. Funnene antyder at det er en temperatureavhengighet for halvverdibredden, men ingen konkluderende relasjon har blitt foreslått. Drude-modellen og den dielektriske formalismen innen mange-partikkel systemer har blitt brukt som utgangspunkt for det teoretiske grunnlaget for at elektronenergitapsspektroskopi kan bli brukt for å finne ledningsevnen, dog har det blitt antatt at de eksperimentelle resultatene kan forklares ut ifra plasmonspreddning i langbølgegrensen og kan direkte knyttes opp mot Drude-modellen. Det har videre blitt spekulert i at dette ikke har vært tilfellet for forsøkene som har blitt gjennomført, og dermed at bidrag fra plasmonspreddning som ikke er i langbølgegrensen i tillegg til interbåndsoverganger har hatt en utbredende effekt. For å begrense den påfølgende asymmetrien fra bevegelsesmengdeoverganger med kortere bølgelengde i eventuelle fremtidige forsøk har det blitt foreslått at akselerasjonsspenningen i transmissjonselektronmikroskopiskolonnen og oppsamlinksvinkelen inn til spektrometeret reduseres.

Preface

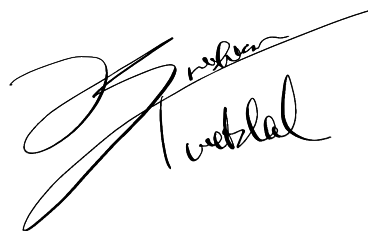
This master thesis completes a five year master program in Nanotechnology at the Norwegian University of Science and Technology (NTNU), and is a continuation of a preliminary project thesis during the autumn 2022. The low-loss regime of aluminium and platinum was investigated during that project. The work concluded that the data quality was inadequate for the purpose due to lack of energy resolution and carbon contamination, respectively.

I would like to thank my supervisors Randi Holmestad, Ragnvald Mathiesen and Emil Frang Christiansen for their guidance and patience during both this work and the project thesis. Not many people can say that they have had the pleasure to visit both England and Japan during their master thesis. Thank you Randi for fixing this and all the additional thoughts and experts you have gained valuable considerations from. Emil, thank you for all your support with sample preparation, TEM training, EELS acquisition, and tips and tricks for especially plotting (and `Hyperspy` last semester). I still remember Friday 16.12.2022 when we spent 12 h to acquire the EELS data for my preliminary project thesis and apologized for “stealing” my Friday afternoon. *I* am sorry for stealing *your* Friday afternoon. Thank you Ragnvald for all your insights in the physical aspects of the plasmon peak. I don’t know how this would have been without your knowledge and ability and patience to explain the concepts to enhance my understanding. Additionally, I would like to thank Per Erik Vullum for his contribution on the FIB, making sure that everything was in order prior to EELS analysis.

I would also like to thank Simon Fairclough and John Walmsley from the University of Cambridge for conducting the experiments leading to all the EELS data presented in this work. Unless otherwise specified, all data presented in this thesis has been acquired by Simon Fairclough.

When referring to a *plasmon* it will throughout the entire text be implicit that we are talking about the *bulk*-plasmon and not the *surface*-plasmon as the latter has not been investigated.

Til slutt vil jeg rette en takk til alle som har bidratt til 5 uforglemmelig år i tigerstaden. Takk for dere har gjort studenthverdagen til å ikke bare omhandle studier. En spesiell takk rettes til medlemmene av Timini kull-18 (wwwiiiiiiiiiiiiiiiiii) og NTNUI Samba (Samba-toget ruller videre!). Jeg er glad i dere < 3

A handwritten signature in black ink, appearing to read 'Kristian Tveitstøl', written in a cursive style.

Kristian Tveitstøl
June 23, 2023

List of Aberrations

ADF angular dark field. 20, 21

EELS electron energy loss spectroscopy. 2, 9

FIB focused ion beam. 15, 20

FWHM full width at half maximum. 8, 9, 18

GIF Gatan imaging filter. 15

SNR signal-to-noise-ratio. 12, 17, 20

STEM scanning transmission electron microscope/microscopy. 2

TEM transmission electron microscope/microscopy. 4, 5, 15

ZLP zero-loss peak. 12

Selected Experimental Values for Aluminium

Table 1: selected physical properties for aluminium. The conductivity is taken at room temperature. The electron density is taken from Ref. [1], while the other properties are taken from Ref. [2].

Quantity	Symbol	Value
Conductivity	σ	$37.7 \cdot 10^6 \text{ S m}^{-1}$
Temperature coefficient	α	0.0053
Coefficient of linear expansion	β	$23 \cdot 10^{-6} \text{ K}^{-1}$
Electron density	n	$18.1 \cdot 10^{22} \text{ cm}^{-3}$

Contents

Abstract	i
Sammendrag	ii
Preface	iii
List of Aberrations	v
Selected Experimental Values for Aluminium	vi
1 Introduction	1
2 Theory	4
2.1 Scattering	4
2.2 Drude Model	5
2.3 Dielectric formulation	6
2.4 Plasmons	9
2.5 Metals and thermal dependencies	9
2.6 Electron Energy Loss Spectroscopy	10
2.7 Deconvolution techniques	11
2.8 Lindhart Extension to higher k -values	13
3 Method	15
3.1 Sample preparation	15
3.2 EELS acquisition	15
3.3 Data handling	17
3.4 Note	18
4 Results and Discussion	20

5 Conclusion	29
5.1 Conclusion	29
5.2 Future work	29
A Temperature plot	35
B Resolution	36
C Effect of Deconvolution for SplitLorentzian	37
D Code	38

Chapter 1

Introduction

Nanotechnology and nanomaterials is an emerging field of research attempting to tailor the (macroscopic) behaviour of materials and devices to meet certain needs for physical properties. Mechanical strength, electrical conductivity, thermal properties and also functional properties such as energy conversion between photons and electrical energy (solar cells) and electrochemical properties (carbon capture) are all examples of material properties one wants to control, and in order to control these aspects, both understanding and manipulation at the micro-/nano scale are required [3]. Being able to manipulate materials down to the nano-scale would contribute to the different challenges the modern society are facing. Nanomaterials and -technology may offer more efficient and economically sustainable materials in the search for environmentally sustainable energy sources, more efficient transistors, nanoparticles that can help removing contaminants in water where potable water is lacking, as well as both medical treatment and -imaging [4, 5].

Aluminium is one of the largest exports in Norway, only behind oil/gas and fish, and due to its versatility and manipulable mechanical properties, price and weight, aluminium and aluminium alloys are a big field of research. With an increasing interest in using aluminium for electrification purposes in e.g. automotive industry and wires, there are great incentives of finding alloys with a high electrical conductivity and good mechanical performance. Whilst pure and undeformed, aluminium has a high electrical conductivity and can suitably serve as an electrical conductor. However, the mechanical properties of pure aluminium are less suited for practical use. It is easily bendable and fails to reach the performance required to meet the needs in e.g. wires and chassis (cars, airplanes etc.). Aluminium alloys however, are often used in all the aforementioned [6]. Here the strength is improved by

alloying with e.g. magnesium, lithium, silicon and zinc, depending on the required properties [7]. This even holds at elevated temperatures, making the alloys suitable as an all-round material for many applications. One of the drawbacks of such alloying is that the electrical conductivity is lowered. Today, copper is most widespread as an electrical conductor. It is fairly cheap, environmentally sustainable and has a high electrical conductivity relative to other candidates. On a world basis however, there is a supply shortage resulting in an increasing price [8]. It is not enough copper, taking the massive electrification of today's society into account. This creates a further incentive of finding new materials in the future to meet the climate goals regarding electrification. Tailoring aluminium in such a way that it can replace copper as an electrical conductor would then be beneficial regarding price, recyclability and weight. With an industrial demand of both high conductivity and improved mechanical properties, a suitable trade off between the two must then be found. As alloying with microconstituents with a size typically ranging down to the nano-scale, improved understanding of the electrical effects they have at precisely the nano-scale is then required to perform this tailoring [9].

Scanning transmission electron microscope/microscopy (STEM) is a technique offering sub-nanometer resolution, and utilizing this resolution for the aforementioned analysis would be a great technique to measure a *local*- or a (sub-)nanometer electrical conductivity. This would require additional spectroscopy methods and in this project, it will be investigated whether electron energy loss spectroscopy (EELS) can be utilized for this purpose. As the electrical properties of pure aluminium are well documented, the applicability of this method will be tested for pure aluminium. A temperature range spanning from room temperature to 500 °C has been considered, affecting both the macroscopic conductivity as well as the expected signal. It has been investigated whether or not the technique can reproduce the macroscopic electrical conductivity as function of temperature. The Drude model and the dielectric formulation within solid state physics have been utilized as the physical foundation relating the conductivity to the EELS signal [10, 11].

Similar experiments have been carried out by J.-L. Verger-Gaugry and P.Guyot on crystals and quasicrystals in AlMn and AlMnSi in the mid. 1980's [12]. However, their results had uncertainties at the same order of magnitude as their calculated mean values, and as far as the author can see from the literature, there has not been conducted many studies since then. It is thus assumed that advances in TEM instrumentation and EEL spectroscopy could allow further investigation of such measurements.

The report will firstly introduce the physical and instrumental aspects of the project as well as a short introduction to some deconvolution techniques. Secondly, the methods that has been used during sample preparation, EELS acquisition and post-processing will be presented. Lastly, the results will be presented and discussed.

Chapter 2

Theory

This chapter will firstly introduce the underlying physics justifying the attempt to use EELS as means for calculating the conductivity, being a short introduction of the differential cross section within quantum mechanics, the Drude model followed by the dielectric formulation of many-particle systems. This will be followed by the theory of a plasma and *plasmons*, being a quantum of plasma oscillation. Secondly, the chapter introduces shortly some theory behind the working principle of EELS, and lastly, some deconvolution techniques will be presented. The reader is expected to be acquainted with transmission electron microscope/microscopy (TEM) and to have basic knowledge in solid state physics. If so is not the case, several textbooks are available covering these topics, such as Refs. [1, 11, 13] and Refs. [10, 14] for the physics and instrumentation, respectively. Unless otherwise specified, the theory presented in this chapter rests on these textbooks.

2.1 Scattering

As the electron beam hits the sample, multiple scattering events may occur. These are mainly divided into *elastic*- and *inelastic* scattering. Elastic scattering originates from interactions between the incoming electron and the electrostatic field from the nuclei and bound electrons in the sample. Although *some* energy gets lost in the process in terms of excitation of phonons, the energy loss in these interactions are typically not resolved in a TEM [10]. Inelastic scattering on the other hand, results in a detectable and measurable energy loss and originates from electron-electron interactions. These scattering events include excitation of the electrons present in the sample, i.e. absorption. Additionally, and more relevant for our purposes, the beam

electrons in a TEM may interact with the free electron gas as a collective. This will be further explained in section 2.4 when discussing plasmas and plasmons.

Central to the scattering mechanisms is the differential cross section

$$\frac{d\sigma}{d\Omega} \tag{2.1}$$

representing the probability of an incident electron being scattered per unit solid angle by an atom. Here, σ is the cross section of scattered electrons at a solid angle Ω . Finding this cross section is itself a large field of study within quantum mechanics and will in the following be limited to the work of Ritchie [15] and Lindhart [16].

2.2 Drude Model

The Drude model was developed by Paul Drude in 1900 and is an application of the kinetic theory of gases. Originally, the Drude theory coarsely assumed that the electrons behaved in a classical manner (i.e. billiard balls), colliding with positively charged and immobile spheres [1, 17]. In later work, the theory has been refined due to new discoveries such as the atomic model, phonons and other scattering events. The Drude model is now considering free electrons in a lattice of positive ions. The only scattering event that is considered is the interactions between the free electrons and the ion lattice, thus neglecting electron-electron interactions. If the free electrons have a momentum per electron $\mathbf{p}(t)$ at a time t and a momentum $\mathbf{p}(t + dt)$ at some later time dt , the fraction of electrons having “collided” with the ion lattice will be dt/τ , where τ is the *relaxation time* of the system. With no external forces, the electron gas will (if not at rest) come to rest, and the net momentum of the free electrons will become zero. If an external force is present however, the electrons will gain net momentum between the collisions. The resulting equation of motion for the electron gas then becomes

$$\frac{d\mathbf{p}}{dt} = \frac{-\mathbf{p}}{\tau} - e\mathcal{E}, \tag{2.2}$$

where the external force has been taken to originate from an external electrical field \mathcal{E} , and e is the elementary charge. The effect of single-electron-lattice interactions therefore acts as a dampening force to the free electron gas as a collective. For a many particle electron gas, the conductivity, σ (not to be

confused with the cross section in section 2.1), is defined through

$$\mathbf{j} = -ne\langle\mathbf{v}\rangle = \sigma\mathcal{E} \quad (2.3)$$

where \mathbf{j} is the current density, n is the electron density and $\langle\mathbf{v}\rangle$ is the mean velocity of the electrons. Taking $x, \mathcal{E} \propto \exp(-i\omega t)$ where ω is the frequency, gives

$$\sigma(\omega) = \sigma_0 \frac{1 + i\tau\omega}{1 + \omega^2\tau^2} \quad (2.4)$$

where σ_0 is defined through the relationship

$$\sigma_0 = \frac{ne^2\tau}{m}, \quad (2.5)$$

whereas m is the electron mass. The current σ_0 is also known as the *Drude conductivity*, and represents the DC conductivity of the material.

2.3 Dielectric formulation

The dielectric function, $\epsilon_r(\mathbf{k}, \omega)$, describes the dielectric response of the material to an external electromagnetic field and is therefore also known as the *dielectric response function*. Its dependency on both frequency and wave vector, \mathbf{k} , is strong, though the long wavelength limit $\epsilon_r(\mathbf{k} \rightarrow 0, \omega)$ will be assumed sufficient for our purposes describing the collective excitation of the electron gas [11].

By representing a transmitting electron as a point charge with coordinate \mathbf{r} and velocity ν , satisfying the Poisson equation,

$$\epsilon_0\epsilon_r(\mathbf{k}, \omega)\nabla^2\phi(\mathbf{r}, t) = e\delta(\mathbf{r}, t), \quad (2.6)$$

where ϵ_0 is the vacuum permittivity, $\phi(\mathbf{r}, t)$ is the electrostatic potential and $\delta(\mathbf{r}, t)$ is the Dirac delta function, R. H. Ritchie showed that the differential cross section in Equation 2.1 for small angles can be written as

$$\frac{d^2\sigma}{d\Omega dE} \propto \frac{1}{\nu^2} \Im\left(\frac{-1}{\epsilon_r(k, E)}\right) \left(\frac{1}{\Theta^2 + \Theta_E^2}\right), \quad (2.7)$$

where $\Theta_E = E/(\gamma m\nu^2)$ is a characteristic angle, γ is a relativistic factor, ν is the velocity of the incident electrons, E is the energy, Θ is the scattering angle and $\Im(-1/\epsilon(k, E))$ is the *energy-loss function*, providing a complete description of the medium the transversing electron is going through [10, 15].

Two of Maxwell's equations read

$$\nabla \cdot \mathbf{D} = \rho_{\text{ext}} \quad ; \quad \nabla \cdot \mathbf{E} = \rho_{\text{tot}}/\epsilon_0 \quad (2.8\text{a,b})$$

where \mathbf{D} and \mathbf{E} are the electric displacement and electric field, respectively, whilst ρ_{ext} and ρ_{tot} is the external charge density and the total charge density, respectively. The dielectric function is defined through

$$\mathbf{D} = \epsilon_0 \mathbf{E} + \mathbf{P} = \epsilon_r \epsilon_0 \mathbf{E}, \quad (2.9)$$

where \mathbf{P} is the polarization density, and will in the long wavelength limit take the form

$$\epsilon_r(\omega) \approx \epsilon_{\omega \rightarrow \infty} \left[1 - \frac{\bar{\omega}_p^2}{\omega^2} \right]. \quad (2.10)$$

Here $\bar{\omega}_p$ is the *plasma frequency* accounting for the ion core distribution at high frequency:

$$\bar{\omega}_p^2 = \omega_p^2 / \epsilon_{\omega \rightarrow \infty} = \frac{1}{\epsilon_{\omega \rightarrow \infty}} \frac{ne^2}{\epsilon_0 m}, \quad (2.11)$$

n and m are the electron density and the electron mass, respectively, whilst e is the elementary charge and $\epsilon_{\omega \rightarrow \infty}$ is the dielectric contribution from the ion background at high frequency. Inserting plane waves into the wave equation for a non-magnetic medium

$$\mu_0 \ddot{\mathbf{D}} = \nabla^2 \mathbf{E}, \quad (2.12)$$

μ_0 being the vacuum permeability, and using Equation 2.9 provides the dispersion relation, being a function of \mathbf{k} and ω

$$\epsilon_r(\mathbf{k}, \omega) \frac{\omega^2}{c^2} = \|\mathbf{k}\|^2. \quad (2.13)$$

From here, it is clear that for $\epsilon_r < 0$, $\|\mathbf{k}\|$ becomes imaginary and the waves are attenuated exponentially. The plasma frequency therefore acts as a lower cutoff for wave propagation in a plasma, in which $\mathbf{P} = -\epsilon_0 \mathbf{E}$. At this frequency, the dielectric function equates to 0 and a collective longitudinally polarized wave mode of the system gets excited, whereas transversal modes are excited at higher frequencies. This is schematically illustrated in Figure 2.1 showing both the dispersion relation in Equation 2.13, the optical limit as well as a shaded region indicating that waves cannot propagate at frequencies lower than ω_p . Not considering spin, the equation of motion of

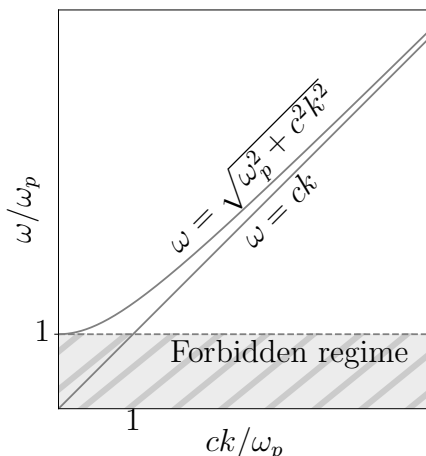


Figure 2.1: the dispersion relation for the plasma waves as well as the optical limit. No propagation occurs in the shaded region.

the plasma is given by

$$\begin{aligned} \mathbf{P} = -ne\langle\mathbf{x}\rangle &= \frac{-ne^2\tau}{m(\omega^2\tau - i\omega)} [\boldsymbol{\mathcal{E}} + \langle\mathbf{v}\rangle \times \mu\mathbf{H}] \\ &\approx \frac{-ne^2\tau}{m(\omega^2\tau - i\omega)} \boldsymbol{\mathcal{E}} \end{aligned} \quad (2.14)$$

where \mathbf{H} is the magnetic field strength, and the latter holds true in the long wavelength limit $\langle\mathbf{v}\rangle = \hbar\langle\mathbf{k}\rangle/m \rightarrow 0$, being the same form as for the Drude theory. Conclusively, the dielectric formalism in the long wavelength limit predicts that the Drude conductivity defined in Equation 2.4 can be investigated thorough the properties of the collective electron motion at the plasma frequency.

Jumping back to the differential cross section derived by Ritchie and introduced in Equation 2.7, the energy-loss function in this long wavelength limit is given by

$$-\Im\mathfrak{m}\left(\frac{1}{\epsilon_r(k=0, E)}\right) = \frac{E(\Delta E)E_p^2}{(E^2 - E_p^2)^2 + (E\Delta E)^2}, \quad (2.15)$$

known as the *inverted Drude-Sellmeier function* [13]. In this equation, $E_p = \hbar\omega_p$ is the plasma energy, $\Delta E = \hbar/\tau$ is the full width at half maximum

(FWHM) and \hbar is Planck's reduced constant, thus connecting the EELS measurements to the conductivity given in Equation 2.18, and will for $E_p/\Delta E \gg 1$ take the shape of a Lorentzian function [18] (see Equation 2.17).

2.4 Plasmons

As briefly mentioned, at the plasma frequency, the material sets up a depolarizing field to the electric field $\mathbf{P} = -\epsilon_0 \mathcal{E}$. The total dielectric displacement then equates to 0, and Equation 2.2 and Equation 2.14 takes the form of a damped harmonic oscillator, and a longitudinal collective travelling wave of the conduction electron gas is excited. Similarly to a phonon being a quantum of lattice oscillation, a *plasmon* is a quantum of electron density oscillation with energy $E_p = \hbar\omega_p$. A plasmon can be excited in a metallic thin film by a transversing electron as presented in Equation 2.6. The negative charge of the electron couples to the electron gas, and the transversing electron will then have lost energy equal to an integer multiple of the plasmon energy $\hbar\omega_p$. Equation 2.14 then takes the form

$$n\ddot{\mathbf{x}} = -\frac{n}{\tau}\dot{\mathbf{x}} - \frac{n^2 e^2}{\epsilon_0 m}\mathbf{x}. \quad (2.16)$$

For $\omega_p\tau \gg 1$, the normalized solution to the differential equation takes the form of a Lorentzian in reciprocal space:

$$L(\omega) = \frac{1}{2\pi} \frac{\Gamma}{(\omega - \omega_p)^2 + \Gamma^2/4}, \quad (2.17)$$

where $\Gamma = 1/\tau$ is the FWHM [18]. The quantities in Equation 2.5 can then be found in an EELS experiment through Equation 2.11, such that

$$\sigma_0 = \frac{\epsilon_0}{\hbar} \frac{E_p^2}{\Delta E}. \quad (2.18)$$

2.5 Metals and thermal dependencies

With increasing temperature, the electron-phonon interactions increase, thus decreasing the conductivity in a metal. According to Matthiesen's rule, the relationship between the total relaxation time in the system is related to the

individual scattering events through

$$\begin{aligned} \frac{1}{\tau_{\text{tot}}} &= \sum_i \frac{1}{\tau_i} \\ &= \frac{1}{\tau_{\text{im}}} + \frac{1}{\tau_{\text{ph}}} \end{aligned} \tag{2.19}$$

where the sum is taken over all scattering events [11]. In this equation, τ_{im} and τ_{ph} are the relaxation times related to impurities and phonon excitation, respectively. For a metal at room temperature, $\tau_{\text{im}} \ll \tau_{\text{ph}}$, and thus Equation 2.19 can be approximated as

$$\tau_{\text{tot}} \approx \tau_{\text{ph}}. \tag{2.20}$$

The collision rate with phonons is proportional to the concentration of phonons, which increases linearly above the Debye temperature Θ_D . With this temperature dependency, the Drude conductivity in Equation 2.5 is expected to decrease as $1/T$ for $T > \Theta_D$ [11], commonly written on the form:

$$\sigma_0(T)^{-1} = [\sigma_0(T_0)(1 - \alpha\Delta T)]^{-1}, \tag{2.21}$$

where α is the temperature coefficient, T_0 is a reference temperature with known conductivity and $\Delta T = T - T_0$ is the temperature difference between the measured temperature and the reference temperature.

A general property of metals is that they expand when the temperature increases. This happens in a linear manner for our temperature ranges, and it can readily be shown from linear expansion of solids and Equation 2.11 that the plasmon energy is expected to decrease linearly with increasing temperature [19]:

$$E_p(T) = E_p(T_0) \left[1 - \frac{3}{2}\beta\Delta T \right]. \tag{2.22}$$

In this equation, β is the coefficient of linear expansion and is taken here as a material constant as enlisted in Table 1.

2.6 Electron Energy Loss Spectroscopy

Electron energy loss spectroscopy measures the kinetic energy of initially monoenergetic electrons after specimen interaction. The beam electrons hits the sample at what is known as the *convergence angle*, α . After the beam

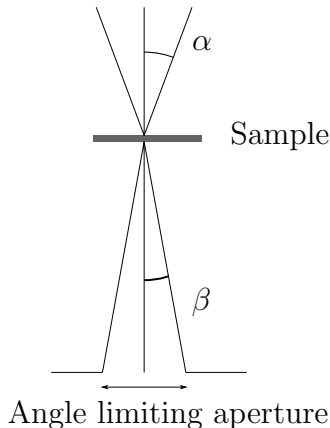


Figure 2.2: schematic of convergence angle, α , collection angle, β , and the angle limiting aperture in EELS experiments.

electrons have transmitted and interacted with the sample, they travel further into the EEL spectrometer through an angle limiting aperture. As the word suggests, this aperture filters out all electrons having scattered more than the *collection angle*, β , see Figure 2.2. The electrons are then bent by magnetic forces in a magnetic prism. The curvature of the electrons are then depending on their velocity as they enter the magnetic field:

$$R = \frac{\gamma m \nu}{eB}. \quad (2.23)$$

In Equation 2.23, m is the electron mass, e the electric charge, ν the electron velocity after specimen interaction, B the strength of the magnetic field, and γ a relativistic factor. The EELS detector therefore detects the energy loss of the electrons after the magnetic prism based this curvature. For our purposes the energy loss regime around the plasmon peak in aluminium is of interest, and will be the main focus.

2.7 Deconvolution techniques

As for all experimental techniques, measurable features are limited by the resolution of the instrument. For our purposes, the term *resolution* can be two different things. The *spatial* resolution is how small features in real space that is detectable. That is, if microconstituents are present in the material, it is the spatial resolution that determines how small clusters/clouds/precipitates one is able to distinguish. The *energy* resolution is a measure of

how well the instrument can distinguish different energy loss peaks. If two physically distinct peaks are closer to each other than the energy resolution, the two peaks will appear as one, wider peak. In general, the resolution affects the detected signal by a convolution between the actual signal and the *instrumental response function*, $R(E)$, determining the resolution. Further on, the response function will be referred to as the energy resolution and how the instrumental setup responds to no inelastic scattering, i.e. the zero-loss peak (ZLP). Setting the response function to have unit area, the ZLP will take the form

$$Z(E) = I_0 R(E), \quad (2.24)$$

where I_0 is the incoming intensity, $Z(E)$ is the acquired signal of the ZLP and E is the energy loss of an electron. The energy resolution of the system is then given by the FWHM of $Z(E)$, in which for our purposes will be approximated by curve-fitting with a Gaussian and a SplitGaussian (see section 3.4). All other features, such as the plasmon peak, will also be affected by the response function through a convolution [10]:

$$J(E) = Z(E) * S(E). \quad (2.25)$$

Here $J(E)$ is the signal recorded by the instrument and $S(E)$ is the signal from the physical processes. Directly solving this equation for $S(E)$ by Fourier transforms would greatly amplify noise, and a *reconvolution function* is required to maintain a sufficiently high signal-to-noise-ratio (SNR). The *Fourier-ratio* method solves this by reconvolving Equation 2.25 with a zero-centered unit area Gaussian, $G(E)$, as a reconvolution function:

$$G(E) * S(E) = \mathcal{F}^{-1} \{g(\nu)j(\nu)/z(\nu)\}. \quad (2.26)$$

Here, $g(\nu)$, $r(\nu)$ and $j(\nu)$ are the Fourier transforms of $G(E)$, $R(E)$ and $Z(E)$, respectively. This gives an additional free parameter, as the FWHM of the Gaussian reconvolution function can be chosen.

Fourier-log deconvolution attempts not only to remove the effect of a finite resolution, but also to eliminate multiple scattering. Assuming that the scattering events follow Poisson statistics, the Fourier transform of the single scattering signal, s , takes the form [10]

$$s(\nu) = I_0 \log(j(\nu)/z(\nu)). \quad (2.27)$$

Once again, solving this by taking the inverse Fourier transform is very prone to noise. The Fourier-log deconvolution technique handles this issue by using $Z(E)$ or $G(E)$ as reconvolution functions, termed *zero-loss modifier* and

Gaussian modifier, respectively, and Equation 2.27 then takes the form:

$$z(\nu)s(\nu) = I_0 z(\nu) \log \left(\frac{j(\nu)}{z(\nu)} \right) \quad ; \quad g(\nu)s(\nu) = I_0 g(\nu) \log \left(\frac{j(\nu)}{z(\nu)} \right). \quad (2.28\text{a,b})$$

The zero-loss modifier will not alter the energy resolution, but remove effects from multiple scattering, whilst the Gaussian modifier will remove multiple scattering effect and can both compensate for a potential asymmetry of $Z(E)$ and improve the resolution by setting the FWHM lower than that of $Z(E)$. This can also introduce other artefacts, especially for noisy data.

2.8 Lindhart Extension to higher k -values

This section is heavily based on Ref [13]. The derivations can be found in this reference and will thus not be repeated here.

Lindhart extended the inverted Drude-Sellmeier function (Equation 2.15) to not only yield in the optical limit, but also for small scattering vectors. He found that the plasmon energy E_p is itself depending on the momentum transfer through

$$E_p(k) = E_p(0) + \frac{\alpha' \hbar^2}{m} k^2 \quad (2.29)$$

where

$$\alpha' = \frac{3}{5} \frac{E_F}{E_p(0)} \quad (2.30)$$

and E_F is the Fermi energy [10]. He further considered both the real and imaginary part of the dielectric function: $\epsilon = \epsilon_1 + i\epsilon_2$. For the real part, a similar expression as in Equation 2.10 is found, but the imaginary part divides the dielectric function into four parts. For our purposes, one of these parts are of interest, and restricts collective excitations of plasmons to frequencies lower than the parabola

$$\hbar\omega = \frac{\hbar^2}{2m}(k^2 + 2kk_F), \quad (2.31)$$

where k_F is the Fermi wave vector. This is shown in Figure 2.3 along with Equation 2.29. The parabola marks the boundary of where ϵ_2 equates to 0, whereas for $\hbar\omega$ greater than this parabola, no electron-hole excitations of wave vector k is possible. This creates a curve in the (k, ω) -plane where ϵ_1 and ϵ_2 are simultaneously zero, up to a cut-off wave vector k_c where they intersect. This curve is shown in blue in the aforementioned figure, showing

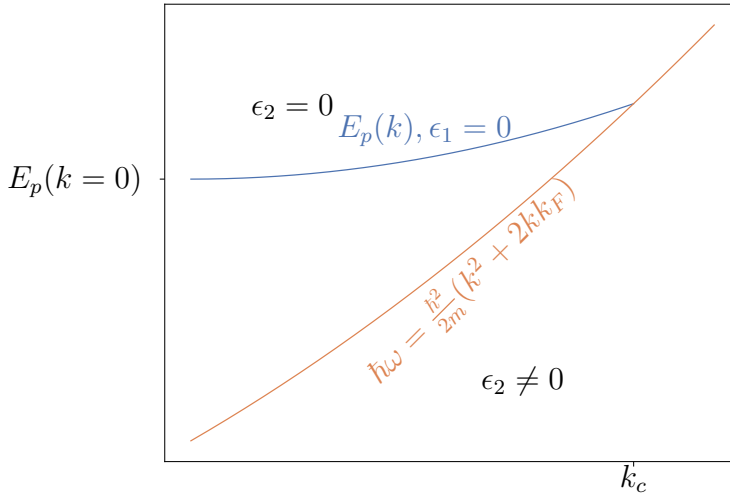


Figure 2.3: schematic of the (k, ω) -plane for the dielectric function as derived in the Lindhart model. The orange curve separates the regions where $\epsilon_2 = 0$ and where $\epsilon_2 \neq 0$

the relevant parts (for our purposes) of the (k, ω) -plane of the dielectric function.

Chapter 3

Method

3.1 Sample preparation

For this project, a 25 μm thick aluminium foil from Goodfellow with a purity of 99.999%¹ was disk punched to have a diameter of approximately 3 mm. The disk was then ion milled in a GATAN PIPS II Model 695, where the beam energy and time are summarized in Table 3.1. The sample was then investigated in a JEOL JEM 2100 TEM operating at 200 kV before extracting two suitable regions in a Helios G5 Plasma Focused ion beam (FIB) and placing them on a P.T.H.TS.1 DENS chip allowing in-situ heating of the sample. During transfer to the DENS chip, the regions of interest were illuminated as little as possible after precursor deposition steps to minimize contamination and damage. An image taken during FIB preparation is shown in Figure 3.1. The two samples were then plasma cleaned in a Fischione Model 1020 plasma cleaner for 1 min 30 s immediately before EELS analysis.

3.2 EELS acquisition

EELS data was acquired using a Gatan Continuum Gatan imaging filter (GIF) with a CMOS detector on a X-FEG monochromated Thermo Fisher Scientific Spectra 300 operated with an acceleration voltage of 300 kV. In order to narrow the ZLP and thus improve the energy resolution, the exposure time was set to 10 ms and the scans were recorded

¹Commercially available at <https://www.goodfellow.com/p/a100-fl-000131/aluminium-foil> (04.06.2023)

Table 3.1: beam energy and duration used in the ion milling step.

Beam energy [eV]	Time [min]
5	140
4	60
3.5	30
3	30
2.5	30
2	30
1.5	30
1	30
0.5	60

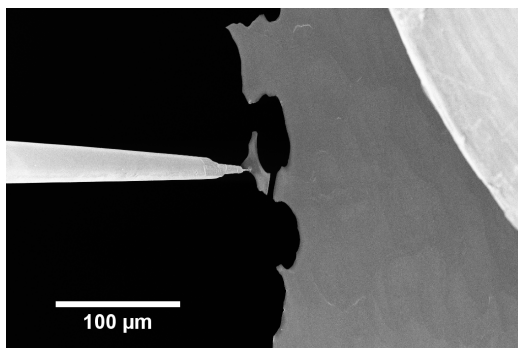


Figure 3.1: sample during transfer to DENS chip in a FIB. Image acquired by Per Erik Vullum

10 times in DualEELS[™] mode with a collection aperture of 100 mrad. The energy ranges for the two simultaneous EEL recordings were set to -2 to 14 eV and 8 to 24 eV with a dispersion of 10 meV per channel. The temperature of the sample was raised sequentially, starting off at room temperature and up to 500 °C as is plotted in Figure A.1. The imaging system was unstable and the resolution was prone to errors as the temperature of the sample was increased. Slight adjustments of the alignment were therefore performed before the first scan at each temperature.

The monochromator is an essential part of the EELS system when energy resolutions of $\lesssim 100$ meV are required. Instead of being limited by the uncertainty in the electron energy from the electron gun, the electrons are filtered out before entering the rest of the TEM column. The improved energy resolution comes at the expense of a lowered intensity (i.e. fewer electrons).

DualEELS[™] makes it possible to acquire two different energy loss intervals simultaneously. It is often used for recording both the ZLP (or low-loss) and a core-loss, where the latter is typically at energies in the keV range. For our purposes, it allows recording both the ZLP and the rest of the low-loss regime, thus avoiding overexposure from the ZLP, affecting the signal from the plasmon peak. Additionally, the energy dispersion can be lowered, in which for our purposes was 10 meV.

3.3 Data handling

Prior to all further data handling, the EEL signals were aligned such that the ZLP was placed at 0 eV using `HyperSpy` [20]. The ZLP used in Equation 2.28a,b was taken to be defined for all electrons having lost ≤ 2 eV. The different signal lines that were curve fitted to the signal was a Lorentzian (Equation 2.17), `SplitLorentzian` (Equation 2.17, see also section 3.4), and the inverted Drude-Sellmeier function² (Equation 2.15) using `SciPy` [21]. Both weighted and unweighted fits were performed, and the energy loss interval for optimization was between 13.5 and 16.5 eV. All pixels were curve fitted in this manner, followed by filtering out the pixels differing from the mean value by one standard deviation for all free parameters³. The remaining pixels were segmented using `scikit-image` [22] and added together to improve the SNR. Segmenting the pixels was based on their measured

²Results not shown as it was indistinguishable from the Lorentzian fit.

³Obvious outliers affecting the mean value to a sufficiently great extent was filtered out manually.

plasmon energy. Background subtraction was performed by subtracting the median value of the signal between 10-12 eV, before deconvolving the signal with the methods presented in section 2.7 using NumPy [23]. As means to obtain information regarding $Z(E)$ in Equation 2.24, the ZLP was curve fitted to both a Gaussian and a SplitGaussian function (see section 3.4). A linear regression was made for all different approaches, selecting the R^2 values as means to estimate how well the signal lines fit the signal [24].

All code can be found in Appendix D.

3.4 Note

As briefly mentioned, several signal lines have been curve fitted to the spectra. Due to asymmetry of the peak(s), it is has been necessary to define distributions with two different widths: one width for energy losses lower than the center, and one width for energy losses greater than the center. A signal line $S(E, E_0, \Delta E)$ will be tagged with “Split” if such a procedure has been performed. For the Split signal line, the function will take the form

$$\text{Split}S(E, E_0, \Delta E_1, \Delta E_2) = I_0 \begin{cases} \frac{1}{S_{E_0^-}} S(E, E_0, \Delta E_1) & , E \leq E_0 \\ \frac{1}{S_{E_0^+}} S(E, E_0, \Delta E_2) & , E > E_0, \end{cases} \quad (3.1)$$

where E_0 is the center, $\Delta E_{1,2}$ are FWHM and $1/S_{E_0^-}$ and $1/S_{E_0^+}$ are defined as

$$S_{E_0^-} = S(E = E_0, E_0, \Delta E_1) \quad ; \quad S_{E_0^+} = S(E = E_0, E_0, \Delta E_2)$$

making sure that the signal line is continuous at $E = E_0$.

Additionally, other attempts has been made in order to acquire the most information regarding the plasmon peak. This includes:

1. Curve fit to the derivative of a SplitLorentzian. This was done as it was hypothesized that the curve fit optimizer would prioritize the energy intervals that was rapidly increasing/decreasing and could thus lead to additional information regarding the FWHM. Additionally, the background level has been assumed to be a constant, and any potential misjudgement of this level would for this procedure be eliminated. It turned out to be equivalent to a weighted fit, though more prone to noise.

2. Altering the energy interval for optimization based on the estimated center. This was done to isolate the energy interval closer to the center, and in particular for the energies between the center and 500 meV lower than the center. This did not make any particular difference.
3. Adjusting the expression for the (Split)Lorentzian such that a change in FWHM would alter the integral and not the maximum value. This function would be mathematically equivalent to the original signal lines, but one could theoretically find the FWHM based on the ratio between the integral and the maximum value. This gave estimates that had a very high variance, and therefore considered inadequate for this purpose.
4. Finding the numerical inflection point for $E < E_p$ as well as the center value. Their difference can be used to find the FWHM, but this was (unsurprisingly) very prone to noise.
5. Inserting a peak with a FWHM derived from the macroscopic conductivity and setting hard restrictions for how far away its center value could be compared to previous estimates. The measured area of this peak was dropped to 0 unless restricted by the optimizer. If restricted, the optimizer would choose the lowest possible area for the peak, and the results from this procedure was therefore considered unphysical.
6. Centering the signal on the plasmon peak instead of the ZLP. This gave a bigger FWHM, likely due to that the SNR for the ZLP was far better, resulting in a less accurate estimate for the plasmon peak compared to the ZLP (also being evident that DualEELS[™] is working properly).

All these attempts did not provide any additional information regarding the peak, and their results will thus not be presented.

Chapter 4

Results and Discussion

As a brief overview, an angular dark field (ADF) image of the sample is shown in Figure 4.1 along with an image taken in the FIB during sample preparation. It shows that the sample is mostly pure aluminium, though contaminated on one of its edges¹. Examples of how the EEL signal looked like is plotted in Figure 4.2, and shows that summation over multiple pixels vastly enhances the SNR. A figure showing the energy resolution defined in section 2.7 is shown in Figure B.1 and shows that the FWHM of the ZLP was around 90 – 100 meV, with the exception of scans taken at room temperature and at 120 °C. Although some deviations of the results depending on the data handling methods were present, the overall tendencies and order of magnitudes were rather similar. The main differences were if the Split-Lorentzian method was used compared to the Lorentzian and if the signal was weighted or not during curve fitting. Unless otherwise specified, the data handling method presented has 10 pixels per segment, was Fourier-log deconvolved with a Gaussian modifier with a reconvolution function having a FWHM of 90 meV, and was not weighted in the curve fitting procedure.

The overall conductivity was measured to be far lower than the macroscopic conductivity, measuring a conductivity of $(4.3 \pm 0.2) \cdot 10^6 \text{ S m}^{-1}$ at room temperature with a temperature dependency as is shown in Figure 4.3 along with a linear regression of Equation 2.21. It is already here clear that the conductivity at room temperature has not been reproduced, as the measured value and the macroscopic value differ with a factor of 9. Further, the temperature dependency for the plasmon energy is shown in Figure 4.4 and was measured to be $14.94 \pm 0.01 \text{ eV}$ at room temperature, declining with approximately 0.5 meV K^{-1} . Solving Equation 2.11 for the electron density

¹This region was not included in the further data handling

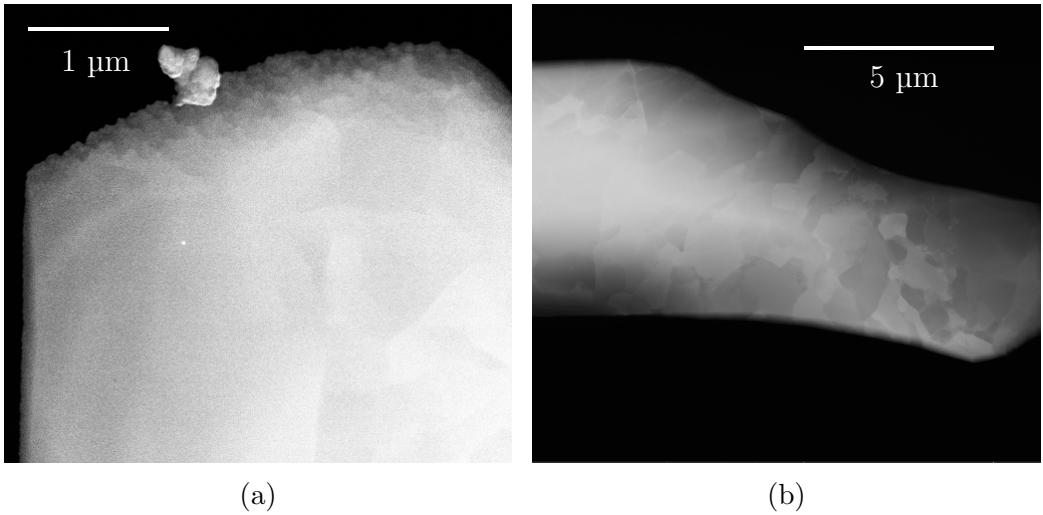


Figure 4.1: a) ADF image of the sample, and b) an image from the FIB during FIB-milling. The images a,b) were acquired by Simon Fairclough and Per Erik Vullum, respectively.

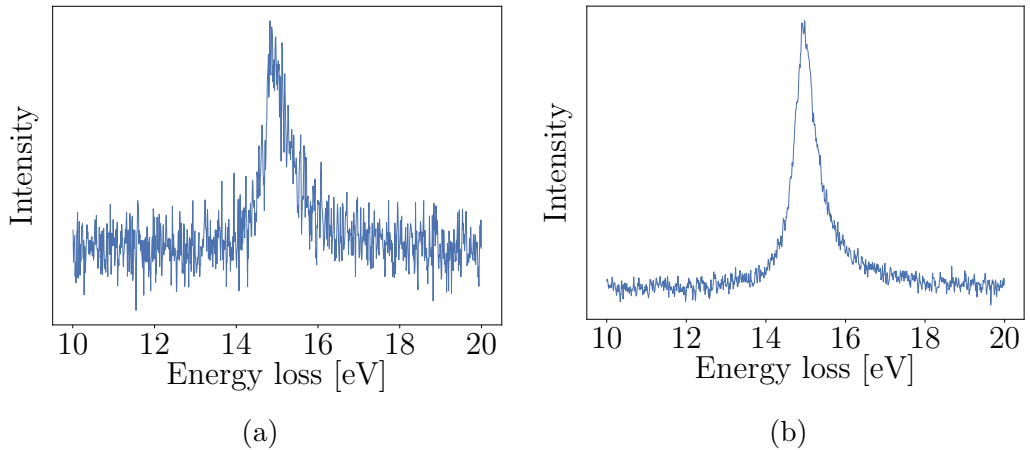


Figure 4.2: examples of unprocessed EELS signals from a) one pixel and b) segment with 100 pixels per segment.

gives $n = 18 \cdot 10^{22} \text{ cm}^{-3}$. The decline in plasmon energy for increasing temperature can be explained through Equation 2.22, measuring a temperature coefficient of approximately $23 \cdot 10^{-6} \text{ K}^{-1}$. Comparing both this decline and the electron density to the values enlisted in Table 1 reveals that these values are well within an acceptable accuracy, and the plasmon energy can therefore be considered to be found. This is also congruent with existing literature, showing both the same plasmon energy [25] and temperature dependency [19]. It must here be stressed that the plasmon energy at room temperature was slightly different for the SplitLorentzian and the Lorentzian curve fit, whereas the latter is more similar to previous EELS experiments.

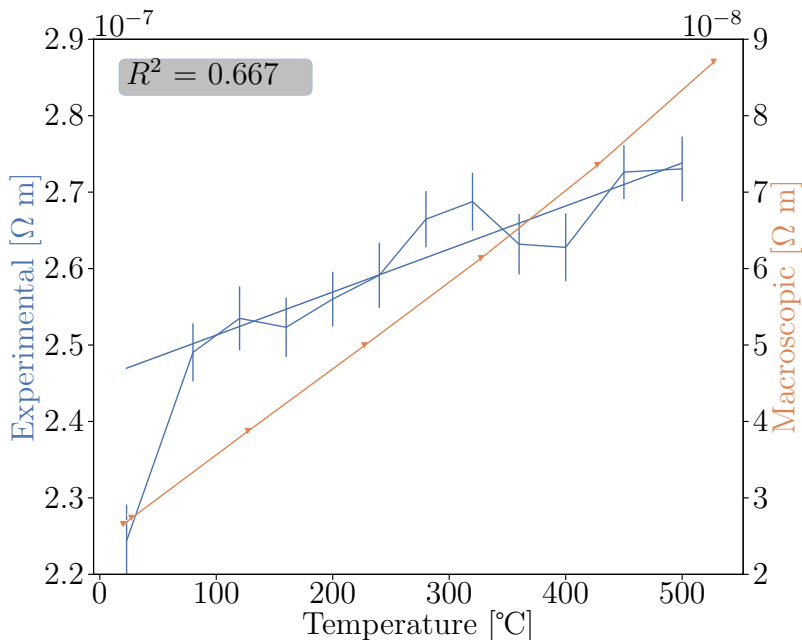


Figure 4.3: measured resistivity (inverse conductivity) of aluminium as function of temperature compared to the macroscopic resistivity. Note that the offset for the two colors are different, but the scale is similar. For the regression analysis, the data at room temperature were left out, see also Figure 4.5b. The macroscopic data points are taken from Ref. [2]

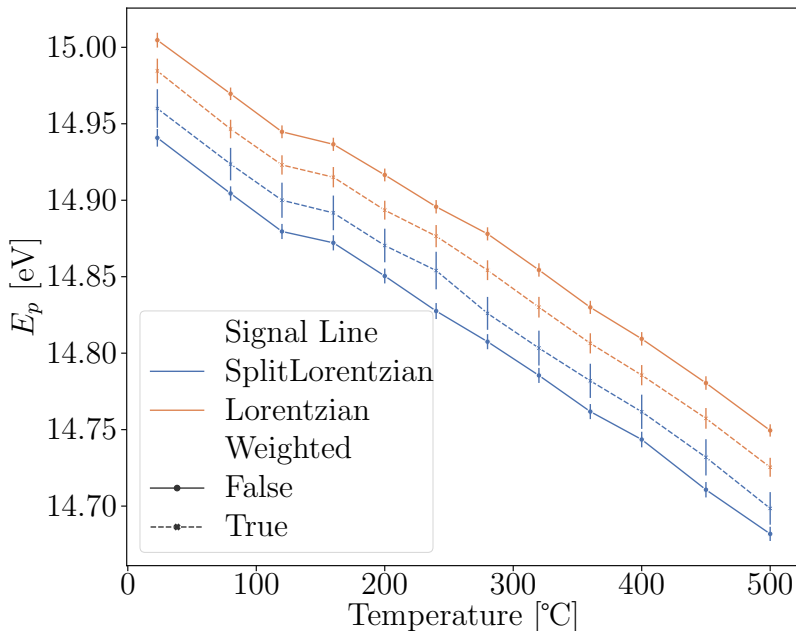


Figure 4.4: temperature dependency for the measured plasmon energy for the SplitLorentzian and the Lorentzian function.

It is therefore evident that it is the FWHM of the plasmon peak that is causing such a big discrepancy between our measurements and the macroscopic conductivity. Predicted at room temperature, the FWHM should have been around 80 meV, while our results gave a FWHM greater than 450 meV, and should have increased to about 240 meV at 500 °C. Although the instrumental resolution of $\lesssim 100$ meV would have failed to find the true FWHM at room temperature, there is a strong discrepancy between the measured FWHM and a convolution between the theoretical Lorentzian peak and the ZLP. This is shown in Figure 4.5a, showing the FWHM for $E < E_p$ and $E > E_p$ for the SplitLorentzian. In addition, Figure 4.5b shows how well the SplitLorentzian fit is compared to the Lorentzian fit, making it clear that the asymmetry of the SplitLorentzian function makes a better fit than the Lorentzian function. With this in mind, it is clearly evident that the plasmon has been broadened towards higher energies. One of the premises for the method to work, was that the EEL spectra was due to plasmon excitations in the long wavelength limit, $k \rightarrow 0$. As the collection aperture was set to 100 mrad and the acceleration voltage was 300 kV, the incoming electrons had an initial wave vector of $k_0 = 3191 \text{ nm}^{-1}$, allowing all momentum

transfers up to $k \approx k_0 \theta_{\text{collection aperture}} = 319.1 \text{ nm}^{-1}$ into the spectrometer. The dielectric scattering cross section in Equation 2.7 would predict a rapid decline in intensity for higher values of k as Θ_E is of the order 0.1 mrad [10] and the dispersion relation in Figure 2.1 predicts that the plasma oscillations quickly becomes transversal rather than longitudinal. However, experimental k -resolved evidence show that there are great contributions from higher k -values [26]. Batson and Silcox mapped the (k, ω) plane of the plasmon response, mapping out both the plasmon energy, linewidth and intensity [27]. They measured the coefficient, α' , in Equation 2.29 to be $\alpha' = 0.38 \pm 0.02$, the FWHM to increase for increasing k , and that the contribution from $k \neq 0$ is non-negligible even for $k > k_c = 1.3 \text{ \AA}^{-1}$ (see Figure 2.3, numerical value also given in Ref. [27]) with an incident beam of 75 keV. They found that the differential cross section in Equation 2.7 is insufficient to describe the observed EEL spectra as it only considers single scattering and that multiple scattering inflicts the measurements. This was for both their and our purposes eliminated through the Fourier-log deconvolution techniques, and the corrections to the FWHM this yielded in this experiment is shown in Figure C.1 for the SplitLorentzian. The Fourier-log deconvolution technique does not however, take the contribution for higher k -values into account, and a broadening to higher energy losses will still be present. Although Batson and Silcox indeed obtained an intensity map of the momentum transfer- and energy loss-dependency for the plasmon sufficient to explain our broadening and asymmetry in a qualitative way, the accuracy of their measurements were limited as their equipment was not up to modern standards². It is therefore evident that more sophisticated theoretical framework is needed to explain the contribution to our measured energy loss. Such calculations have been made by Ferrell, based on Hartree-Fock wave function analysis [28] and is shown in Figure 4.6 along with a sharp cut-off approximation taken from Ref. [10]. His calculations showed that despite the dispersion relation in Figure 2.1, there are indeed non-negligible contributions for $k < 0.74k_F = 1.3 \text{ \AA}^{-1}$, where k_F is the Fermi wave vector and the numerical value is taken from Ref. [2]. The cutoff wave vector will in the Lindhart extension to higher k -values gives a plasmon energy at approximately 18 eV. This contribution explains both the asymmetry and the broadening of the peak for $E > E_p$. However, as the contributions from scattering events not being in the long-wavelength limit strictly contributes to $E > E_p(k = 0)$ the energy losses from $E < E_p$ is not

²or in their words: “(...) for the most part, the accuracy of the intensity measurements reported here surpasses the ability of the draftsman and printer to reproduce them on plots of this nature” [27].

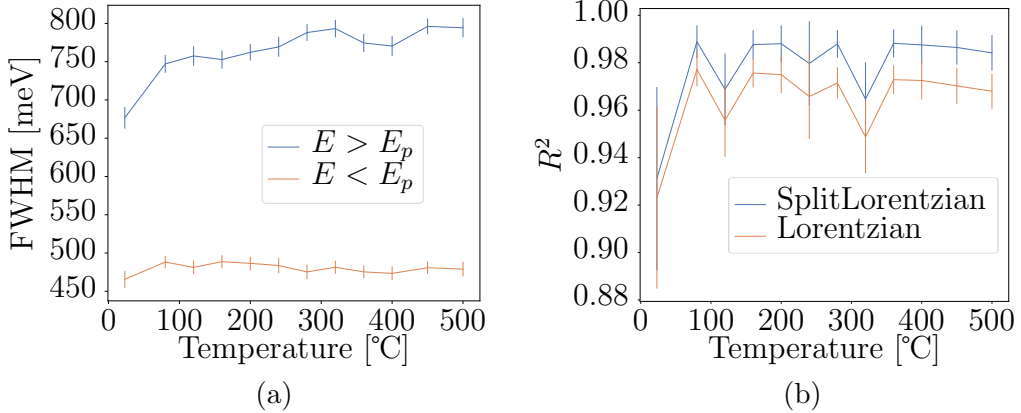


Figure 4.5: temperature dependency for a) the FWHM for the Split-Lorentzian, and b) the R^2 value for the SplitLorentzian and Lorentzian function.

completely explained. The additional intensity from higher values of k close to- but not *sufficiently* close to 0 still contribute with a rather high intensity. Their contribution may be enough to shift the maximum of the acquired energy loss peak to a higher energy than the plasmon energy at $k = 0$. As the spectrometer used in these experiments are not k -resolved, finding the intensity distribution as function of k is unavailable, and our resulting energy-loss spectra are effectively integrated over all k -values not filtered out by the collection aperture. Figure 4.7a shows a resulting prediction of how a signal would appear based on Ferrells calculations along with experimental data taken at 500 °C for a segment of 100 pixels. This figure makes it clear that the scattering cross section from Ferrell is indeed promising to explain the asymmetry of the plasmon peak for $E > E_p$, but fails for $E < E_p$. Close to the cutoff frequency, Ferrell's calculations has been experimentally verified by Schmuser [26] for an electron beam with an energy of 40 keV, though the aforementioned paper from Batson and Silcox suggested that the intensity was extending to even higher values of k with a beam energy of 75 keV. As far as the author has seen through literature, Ferrells calculations has not been experimentally verified for sufficiently low k . If the scattering cross section predicts an increasing intensity up to a wave vector k_E as suggested by the sharp cut-off approximation as seen as the dashed line in Figure 4.6, the intensity in the experimental peak would have a peak value shifted to even higher energies that would lead to a measured broadening also for $E < E_p$.

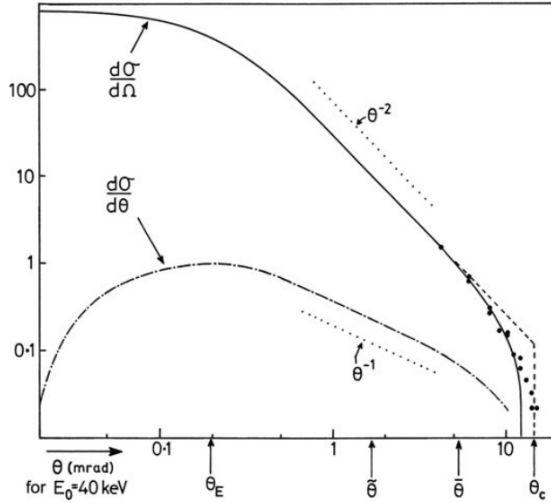


Figure 4.6: the differential cross section as calculated by Ferrell [28] (solid) and using a sharp cutoff approximation [10]. The figure is taken from Ref. [10].

This is illustrated in Figure 4.7b, and shows more promising features, also for $E < E_p$. More mathematically:

$$S_{\text{measured}}(E) = \int_{k>0} S(k, E) dk \neq S(k=0, E), \quad (4.1)$$

where $S(E)$ is the measured peak from Equation 2.27, even for $E < E_{p,\text{measured}}$. If so is the case, this would explain why the plasmon energy estimated from an unweighted fit does not coincide with the peak values obtained from a weighted fit on a quantitative level³: the intensity for $E < E_{p,\text{measured}}$ would simply not be dominated by scattering events from the optical limit.

Another interpretation has been suggested by Smith and Segall, arguing that the frequency ω_p of the bulk plasmon emerges as consequence of both intraband and interband transitions [29], and that the observed plasmon energy in the acquired signal is a consequence of that both contributions show the same asymptotic behavior at this frequency. The interband contributions are not a part of the analysis and their contribution would prevent the analysis as it has been an underlying assumption from Equation 2.21 that the observed scattering events are due to increased amount of phonon

³Quantitative analysis showed a noisy and unpredictable values of the energy shift, though of the order 20 – 30 meV, being only 2-3 times the energy dispersion.

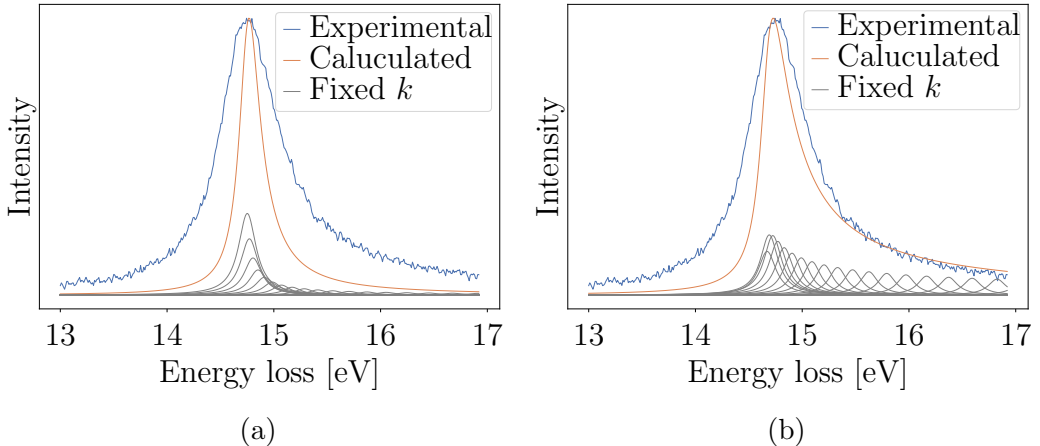


Figure 4.7: estimated shape of plasmon given a Lorentzian plasmon shape as calculated by a) Ferrell and b) the sharp cut-off approximation [10, 28]. The orange line is the sum of all the grey lines (intensity of grey lines not to scale with the other two).

excitations. This interpretation is supported by K. Sturm [30, 31], arguing that the Drude addition to the linewidth is drowning in the interband transitions due to Umklapp scattering. This would explain the shortcomings of both Ferrells calculations and the sharp cut-off approximation as is apparent in Figure 4.7. Further, Smith and Segall found the intraband component to have a frequency of $\Omega_p = 12.5 \pm 0.3$ eV with a relaxation time of $\tau = (1.06 \pm 1.2) \cdot 10^{-14}$ s, resulting in a conductivity at room temperature differing from the macroscopic electrical conductivity with approximately 10% [29].

Regardless of what has caused the broadening of the peak, it is clear that the FWHM indeed is too big to reproduce the scattering time in the Drude model. Equation 2.22 was derived on the premise that the dominating contribution to the overall relaxation was the Drude contribution. Although this is not congruent with Smith and Segall as well as Sterms work, the increase in relaxation time for the Drude contribution should increase the overall FWHM. As the reference conductivity in Equation 2.22 has been measured to be incorrect, it would be more sensible to choose the form $aE + b$, where Figure 4.3 suggests that there are similarities to macroscopic measurements. The slope in the aforementioned form from our experiments is a factor of 1.8 lower than its macroscopic counterpart. In terms of the Lindhart model, this can be explained through that the contribution is drowning in the effects of

higher k -values, whereas the same argument holds for the interband explanation. The fact that there is a temperature dependency however, is likely due to a decreasing conductivity. That the FWHM for $E > E_p$ is the side showing a more prominent temperature dependency can be explained through Figure 4.7, where the extension to higher k -values makes more impact for $E > E_p$, as well as Batson and Silcox findings of an increasing FWHM for increasing k .

Chapter 5

Conclusion

5.1 Conclusion

This project has attempted to find the DC-conductivity of pure aluminium using EELS. The signal line from the plasmon peak over a temperature range from room temperature to 500 °C has been investigated for this purpose. Overall, the plasmon peak has been measured to have a FWHM far too big for this purpose, and a thorough explanation has not been found. It has been speculated that this can be due to contributions from plasmon excitation involving sufficiently high momentum transfer and interband transitions. It has also been shown that the plasmon peak is too asymmetric to assert a Lorentzian shape, likely originating from the contributions from higher k -values. A temperature dependency of the width of the plasmon peak has been found, though more research is required to outline an explicit relation. Furthermore, it has been found that the plasmon energy is around 15 eV at room temperature, decreases linearly with increasing temperature. This tendency is well explained through linear thermal expansion of solids.

5.2 Future work

As has become clear throughout chapter 4, broadening effects due to k -dependency of the plasmon peak is likely a big reason of why the conductivity has not been retrieved. A more thorough understanding of especially the differential cross section as function of both energy loss and momentum transfer is then required for further analysis. This can be done similarly to what Batson and Silcox did [27], but now with instruments up to modern

standards, such as in Refs. [32, 33]. If the experiments are to be repeated without k -resolved EELS, again attempting to find the plasmon behaviour in the long wavelength limit, it would be necessary to lower the acceleration voltage and have a smaller collection aperture in order to prevent contributions from higher k -values. Other papers, as referenced in chapter 4 suggest that this would not be sufficient as they argue that the plasmon is dominated by interband transitions.

Bibliography

- [1] Neil W. Ashcroft and N. David Mermin. *Solid state physics*. Saunders College Publishing, 1976.
- [2] David R Lide. *CRC handbook of chemistry and physics*, volume 85. CRC press, 2004.
- [3] Ahmed Kadhim Hussein. Applications of nanotechnology in renewable energies-a comprehensive overview and understanding. *RENEWABLE & SUSTAINABLE ENERGY REVIEWS*, 42:460–476, FEB 2015.
- [4] Parshant Kumar Sharma, Shraddha Dorlikar, Pooja Rawat, Vidhu Malik, Nishant Vats, Manu Sharma, Jong Soo Rhyee, and Ajeet Kumar Kaushik. 1 - nanotechnology and its application: a review. In Kamil Reza Khondakar and Ajeet Kumar Kaushik, editors, *Nanotechnology in Cancer Management*, pages 1–33. Elsevier, 2021.
- [5] Lina Wang, Mavd P.R. Teles, Ahmad Arabkoohsar, Haoshui Yu, Kamal A.R. Ismail, Omid Mahian, and Somchai Wongwises. A holistic and state-of-the-art review of nanotechnology in solar cells. *Sustainable Energy Technologies and Assessments*, 54:102864, 2022.
- [6] Tolga Dursun and Costas Soutis. Recent developments in advanced aircraft aluminium alloys. *MATERIALS & DESIGN*, 56:862–871, APR 2014.
- [7] Ankitkumar K. Shriwas and Vidyadhar C. Kale. “impact of aluminium alloys and microstructures on engineering properties-review ”. 2016.
- [8] Lee Ying Shan. There isn’t enough copper in the world — and the shortage could last till 2030. <https://www.cnbc.com/2023/02/07/there-isnt-enough-copper-in-the-world-shortage-could-last-1.html>, February 2023.

- [9] Tusar Ranjan Soren, Ramanuj Kumar, Isham Panigrahi, Ashok Kumar Sahoo, Amlana Panda, and Rabin Kumar Das. Machinability behavior of aluminium alloys: A brief study. *Materials Today: Proceedings*, 18:5069–5075, 2019. 9th International Conference of Materials Processing and Characterization, ICMPC-2019.
- [10] R.F. Egerton. *Energy-Loss Instrumentation*. Springer US, Boston, MA, 2011.
- [11] Charles Kittel. *Introduction to solid state physics*. John Wiley & Sons, 8 edition, 2004.
- [12] J.-L. Verger-Gaugry and P. Guyot. PLASMON ELECTRON LOSS SPECTROSCOPY AND ELECTRICAL CONDUCTIVITY AT 300 K OF CRYSTALS AND QUASICRYSTALS IN AlMn AND AlMnSi . *Journal de Physique Colloques*, 47(C3):C3–477–C3–483, 1986.
- [13] Giuseppe Grosso and Giuseppe Pastori Parravicini. Chapter 7 - excitons, plasmons, and dielectric screening in crystals. In Giuseppe Grosso and Giuseppe Pastori Parravicini, editors, *Solid State Physics (Second Edition)*, pages 287–331. Academic Press, Amsterdam, second edition edition, 2014.
- [14] David B. Williams and C. Barry Carter. *The Transmission Electron Microscope*. Springer US, Boston, MA, 2009.
- [15] R. H. Ritchie. Plasma losses by fast electrons in thin films. *Phys. Rev.*, 106:874–881, Jun 1957.
- [16] Jens Lindhard. On the properties of a gas of charged particles. *Dan. Vid. Selsk Mat.-Fys. Medd.*, 28:8, 1954.
- [17] P. Drude. Zur elektronentheorie der metalle. *Annalen der Physik*, 306(3):566–613, 1900.
- [18] Lukas Novotny and Bert Hecht. *Principles of Nano-Optics*. Cambridge University Press, 2 edition, 2012.
- [19] Hiroyuki Abe, Masami Terauchi, Ryuichi Kuzuo, and Michiyoshi Tanaka. Temperature Dependence of the Volume-Plasmon Energy in Aluminum. *Journal of Electron Microscopy*, 41(6):465–468, 12 1992.

- [20] Francisco de la Peña, Eric Prestat, Vidar Tonaas Fauske, Pierre Burdet, Jonas Lähnemann, Petras Jokubauskas, Tom Furnival, Magnus Nord, Tomas Ostasevicius, Katherine E. MacArthur, Duncan N. Johnstone, Mike Sarahan, Joshua Taillon, Thomas Aarholt, pquinn dls, Vadim Mignunov, Alberto Eljarrat, Jan Caron, Carter Francis, T. Nemoto, Timothy Poon, Stefano Mazzucco, actions user, Nicolas Tappy, Niels Cautaerts, Suhas Somnath, Tom Slater, Michael Walls, Florian Winkler, and Håkon Wiik Ånes. *hyperspy/hyperspy: Release v1.7.3*, October 2022.
- [21] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods*, 17:261–272, 2020.
- [22] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. *scikit-image: image processing in Python. PeerJ*, 2:e453, 6 2014.
- [23] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Hal-dane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. *Array programming with NumPy. Nature*, 585(7825):357–362, September 2020.
- [24] Karl Pearson. *Vii. note on regression and inheritance in the case of two parents. proceedings of the royal society of London*, 58(347-352):240–242, 1895.

- [25] Heinz Raether. *Excitation of plasmons and interband transitions by electrons*, volume 88. Springer Tracts in Modern Physics, Springer, 1980.
- [26] Peter Schmüser. Anregung von volumen- und oberflächenplasmaschwingungen in al und mg durch mittelschnelle elektronen. *Zeitschrift für Physik*, 180:105–126, 1964.
- [27] P. E. Batson and J. Silcox. Experimental energy-loss function, $\text{Im}[-\frac{1}{\epsilon}(q, \omega)]$, for aluminum. *Phys. Rev. B*, 27:5224–5239, May 1983.
- [28] Richard A. Ferrell. Characteristic energy loss of electrons passing through metal foils. ii. dispersion relation and short wavelength cutoff for plasma oscillations. *Phys. Rev.*, 107:450–462, Jul 1957.
- [29] D. Y. Smith and B. Segall. Intraband and interband processes in the infrared spectrum of metallic aluminum. *Phys. Rev. B*, 34:5191–5198, Oct 1986.
- [30] Kurt Sturm. Pseudopotential theory of the width of the long wavelength plasmon in simple metals. *Zeitschrift für Physik B Condensed Matter*, 25(3):247–253, 1976.
- [31] Kurt Sturm. Pseudopotential theory of the k-dependent plasmon line width in simple metals. *Zeitschrift für Physik B Condensed Matter*, 28(1):1–7, 1977.
- [32] Ryosuke Senga, Kazu Suenaga, Paolo Barone, Shigeyuki Morishita, Francesco Mauri, and Thomas Pichler. Position and momentum mapping of vibrations in graphene nanostructures. *Nature*, 573(7773):247–250, 2019.
- [33] Hikaru Saito, Hugo Lourenço Martins, Noémie Bonnet, Xiaoyan Li, Tracy C. Lovejoy, Niklas Dellby, Odile Stéphan, Mathieu Kociak, and Luiz Henrique Galvão Tizei. Emergence of point defect states in a plasmonic crystal. *Phys. Rev. B*, 100:245402, Dec 2019.

Appendix A

Temperature plot

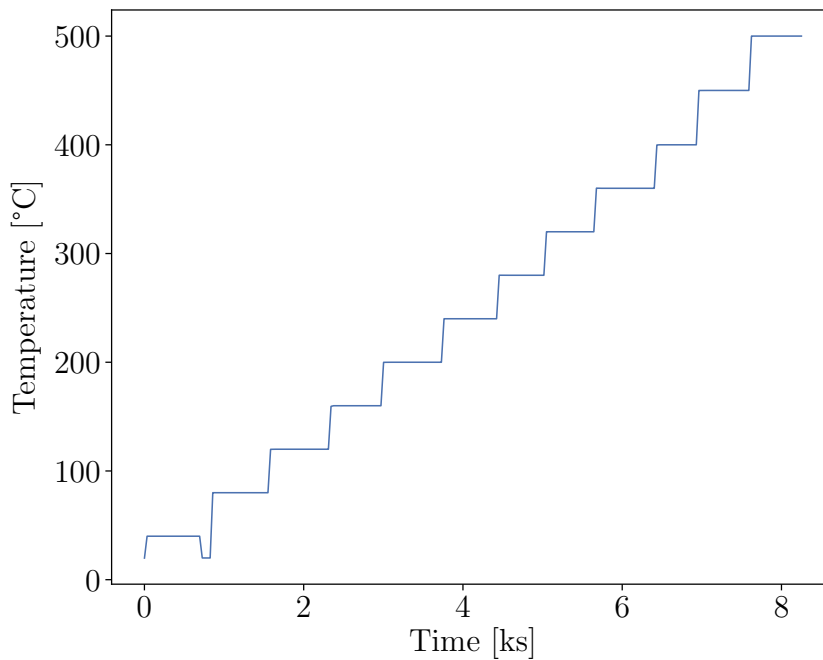


Figure A.1: measured temperature log of the sample during TEM operation.

Appendix B

Resolution

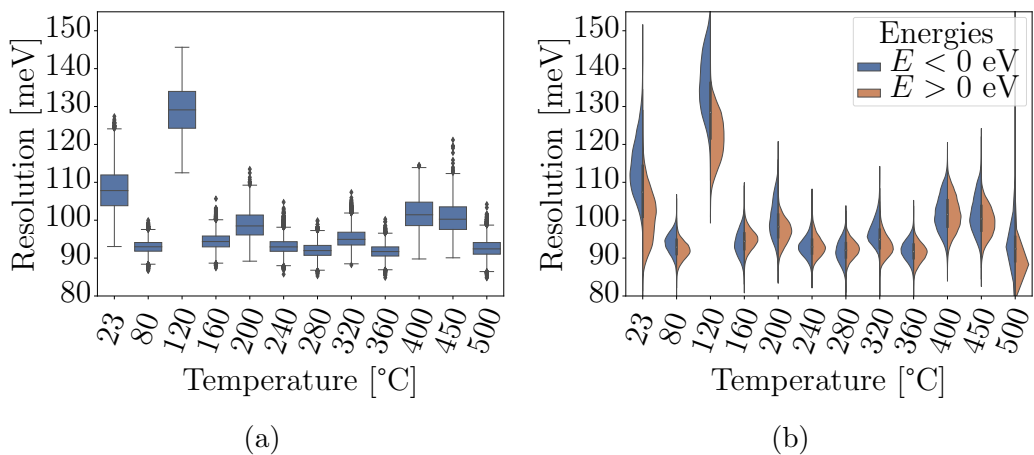


Figure B.1: FWHM of the ZLP after fitting the signal to a) Gaussian and b) SplitGaussian. Note that the temperature axis is not equidistant.

Appendix C

Effect of Deconvolution for SplitLorentzian

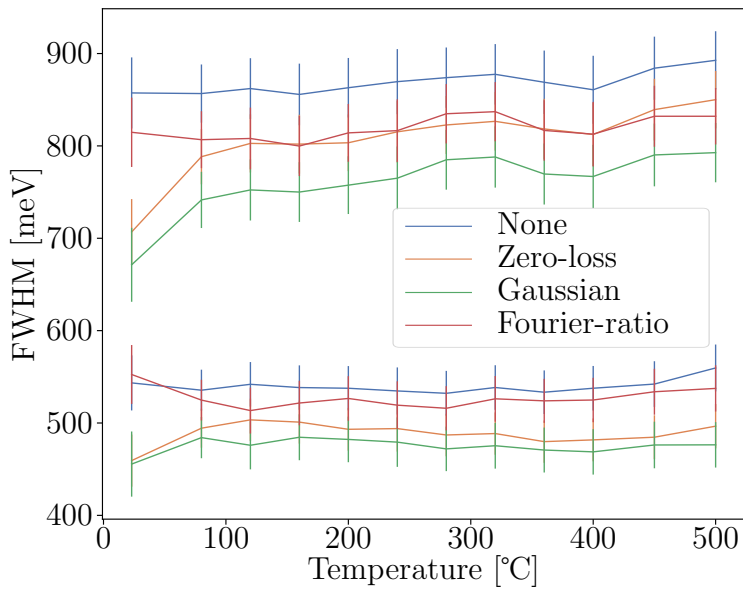


Figure C.1: measured temperature dependency for the signal for the different deconvolution techniques for a SplitLorentzian

Appendix D

Code

```
1 from numpy import pi
2 """This file contains the parameters that can be used for
   initialization of the curve-fitting algorithm
3
4 The order of the dictionary is essential. Only one example is
   shown.
5 """
6
7 class Param:
8     def __init__(self, value, bmin, bmax):
9         self.value = value
10        self.bmin = bmin
11        self.bmax = bmax
12
13
14 drudec = Param(14.8,12,18)
15 drundefwhm = Param(0.5 ,0 ,1)
16 drudea = Param(2e5 ,0 ,1e9)
17
18
19 drude = {'center' : drudec,
20         'fwhm' : drundefwhm,
21         'area' : drudea}
22
23
24
25 """ This file contains wrappers for the different signal lines.
   These include Gaussian, Lorentzian, Voigt, and more."""
26
27 from abc import ABC
28 import numpy as np
```

```

29 import sympy
30 import scipy.special
31 import logging
32
33
34
35
36 dx = '+0.01'
37 dx_ = '-0.01'
38 def prod_id(x):
39     while True:
40         yield chr(ord('a')+x)
41         x+=1
42 id_ = prod_id(0)
43
44 gauss_expr = '(area * 2 * sqrt(log(2))/(fwhm*sqrt(pi))
45             * exp(-4*(log(2)) * ( x -center)**2/fwhm**2 ) ) '
46 loren_expr = '(area / pi * fwhm / 2 / (( x -center)
47             **2 + fwhm**2/4))'
48 # gauss_expr = 'area * exp(-4*(log(2)) * (x-center)**2/fwhm
49             **2 ) '
50 # loren_expr = 'area * ((x-center)**2 + fwhm**2/4)'
51 # voigt_expr = 'area* real(Gfwhm / (4*sqrt(pi*log(2))) * scipy.
52             special.wofz(2*sqrt(log(2)) /Gfwhm * (x-center+1j*Lfwhm/2) )
53             ) '
54 voigt_expr = '(area * real(2*sqrt(log(2))/sqrt(pi)*1/
55             Gfwhm * wofz(2*sqrt(log(2)) /Gfwhm * (x-center+1j*Lfwhm/2)
56             ) ) )'
57 # splitvoigt_expr = 'area * (((1-frac)*'+gauss_expr.replace('
58             area','1').replace('fwhm','fwhm1') + ' + frac*'+loren_expr.
59             replace('area','1').replace('fwhm','fwhm1')+') * (x<=center*1)
60             ' \
61             '+ ((1-frac)*'+gauss_expr.replace('
62             area','1').replace('fwhm','fwhm2') + ' + frac*'+loren_expr.
63             replace('area','1').replace('fwhm','fwhm2')+') * ( x >center*1)
64             ) '
65 splitvoigt_expr = 'area * ( 1/((1-frac)*'+gauss_expr.
66             replace('area','1').replace('fwhm','fwhm1').replace(' x ','
67             center'+dx_) + ' + frac*'+loren_expr.replace('area','1').
68             replace('fwhm','fwhm1').replace(' x ','center'+dx_)+') * ((1-
69             frac)*'+gauss_expr.replace('area','1').replace('fwhm','fwhm1'
70             ) + ' + frac*'+loren_expr.replace('area','1').replace('fwhm',
71             'fwhm1')+') * (x<=center*1) ' \
72             '+ 1/((1-frac)*'+gauss_expr.replace('
73             area','1').replace('fwhm','fwhm2').replace(' x ','center'+dx
74             ) + ' + frac*'+loren_expr.replace('area','1').replace('fwhm',
75             'fwhm2').replace(' x ','center'+dx )+') * ((1-frac)*'+

```

```

    gauss_expr.replace('area','1').replace('fwhm','fwhm2') + ' +
    frac'+loren_expr.replace('area','1').replace('fwhm','fwhm2')
    +') * (x >center*1) ) '
54 volume_expr          = '(area * x * fwhm * center**2 / ((x**2-
    center**2)**2 + (x*fwhm)**2 ) )'
55 step_expr           = '(x>start*1)'
56 inter_expr          = '((x>start)-(x>stop)*1)'
57 power_expr          = 'area*(x-center)**(-k)'
58 # power_expr         = 'area*(x-center * ((x-0.3)>center*1))**(-k)'
    #Note!
59 eps_1_expr           = 'area * (1-center**2 / (x**2 + fwhm**2))'
60 eps_2_expr           = 'area * fwhm*center**2/(x*(x**2+fwhm**2))
    ,'
61 experimental_expr    = '(1-area*(center**2-x**2)/((x**2-center
    **2)**2+x**2*fwhm**2))'
62 lognormal_expr       = 'area * 1/(x*sigma*sqrt(2*pi))*exp(-(log(
    x)-center)**2/(2*sigma**2))' # see e.g. wikipedia
63 dho_expr             = 'area * sigma / (pi*(1-exp(-center/gamma)
    )) * (1/((x-center)**2 +sigma**2) - 1/((x+center)**2 +
    sigma**2)) '
64 # experimental_expr = 'area*(1+ (1+lj*fwhm/x)*() / '
65 fano_expr            = '(area * abs(fwhm/2- background*exp(lj*
    phi)*(lj*( x -center)-fwhm/2) )**2 / (( x -center)**2 +
    fwhm**2/4))'
66 # derivativeL_expr   = '-area * 16 * ( x - center) * fwhm / (
    pi *(4*( x -center)**2+fwhm**2)**2 )'
67 # derivativeL_expr   = '-area * 16 * ( x - center) * fwhm1 /
    (pi *(4*( x -center)**2+fwhm1**2)**2 )+ -area
    * 16 * ( x - center) * fwhm2 / (pi *(4*( x -center)**2+
    fwhm2**2)**2 )'
68 # derivativeL_expr   = '-area * 16 * ( x - center) * fwhm1 /
    (pi *(4*( x -center)**2+fwhm1**2)**2 )*(( x <= center)*1)-
    area * 16 * ( x - center) * fwhm2 / (pi *(4*( x -center)**2+
    fwhm2**2)**2 )*(( x > center)*1)'
69 derivativeL_expr     = '( -area * 16 * ( x - center) * fwhm1
    / (pi *(4*( x -center)**2+fwhm1**2)**2 )*(( x <= center)*1)
    -area * fwhm2/fwhm1 * 16 * ( x - center) * fwhm2 / (pi *(4*(
    x -center)**2+fwhm2**2)**2 )*(( x > center)*1) )'
70
71 splitLorent_expr     = '( 1/(2*pi) * area * fwhm1**2 / (( x -
    center)**2 + fwhm1**2/4) * (x <= center) + 1/(2*pi) *
    area * fwhm2**2 / (( x -center)**2 + fwhm2**2/4) * ( x >
    center) ) '
72 splitGauss_expr      = '(area *( exp(-4*(log(2)) * ( x -center)
    **2/fwhm1**2 ) * ( x <=center) + exp(-4*(log(2)) * ( x -
    center)**2/fwhm2**2 ) * ( x > center) ) )'#A is max value
73

```



```

74 def step_func(x, start):
75     return ((x-start) > 0)*1
76 def define_on_interval(x, start, stop):
77     return step_func(x,start) - step_func(x,stop)
78
79
80 def gaussian(x,center,fwhm,area):
81     sigma = fwhm/(2*np.sqrt(2*np.log(2)))
82     return area * 1/(sigma*np.sqrt(2*np.pi)) * np.exp(-(x-center
83     )**2/(2*sigma**2))
84
85
86 def lorentzian(x, center, fwhm, area):
87     return area * 1/np.pi * fwhm/2 * 1/((x-center)**2 + (fwhm/2)
88     **2)
89
90 def double_lorentzian(x,center,fwhm1,fwhm2,area):
91     return lorentzian(x,center,fwhm1,area)*(x<=center)+
92     lorentzian(x,center,fwhm2,area*fwhm2/fwhm1)*(x>center)
93
94 def voigt(x:np.array,c,FWHM,gamma,area):
95     gamma /= 2
96     sigma = FWHM/(2*np.sqrt(2*np.log(2)))
97     z = (x - c + 1j * gamma) / (np.sqrt(2)*sigma)
98     V = scipy.special.wofz(z)/(np.sqrt(2.0*np.pi)*sigma)
99     return area*np.real(V)
100
101 def volumePlasmon(x,center,fwhm,area):
102     return area * x * fwhm * center**2 / ((x**2-center**2)**2 +
103     (x*fwhm)**2 )
104
105
106 class SignalLine(ABC):
107     def __init__(self,parameters:dict, expr:str = None, id=None,
108     symbols=None) -> None:
109         super().__init__()
110         self.parameters = parameters
111         self.expr = expr
112         self.id = id
113         self.symbols = symbols
114
115     def append_step_func(self,start:float,stop:float = None):
116         """Defines the interval the signal line is defined at"""
117         if stop is None:
118             self.expr += '*' + step_expr.replace('start',str(
119             start))
120         else:

```

```

114         self.expr += '*' + inter_expr.replace('start', str(
115             start)).replace('stop', str(stop))
116
117 class Lorentzian(SignalLine):
118     def __init__(self, parameters:dict, name:str = None) -> None
119     :
120         if name is None:
121             self.id = next(id_)
122         else:
123             self.id = name
124             expr = loren_expr.replace('area', 'area'+self.id).
125             replace('center', 'center'+self.id).replace('fwhm', 'fwhm'+
126             self.id)
127             self.symbols = {}
128             for sym, param in zip(sympy.symbols('center fwhm area'.
129             replace('area', 'area'+self.id).replace('center', 'center'+
130             self.id).replace('fwhm', 'fwhm'+self.id)), parameters.keys())
131             :
132                 self.symbols[sym] = parameters[param]
133             super().__init__(parameters, expr, self.id, self.symbols
134             )
135
136 class Gaussian(SignalLine):
137     def __init__(self, parameters:dict, name:str = None) -> None
138     :
139         if name is None:
140             self.id = next(id_)
141         else:
142             self.id = name
143             expr = gauss_expr.replace('area', 'area'+self.id).
144             replace('center', 'center'+self.id).replace('fwhm', 'fwhm'+
145             self.id)
146             self.symbols = {}
147             for sym, param in zip(sympy.symbols('center fwhm area'.
148             replace('area', 'area'+self.id).replace('center', 'center'+
149             self.id).replace('fwhm', 'fwhm'+self.id)), parameters.keys())
150             :
151                 self.symbols[sym] = parameters[param]
152             super().__init__(parameters, expr, self.id, self.symbols
153             )
154
155 class Voigt(SignalLine):
156     def __init__(self, parameters:dict, name:str = None) -> None
157     :

```

```

145     if name is None:
146         self.id = next(id_)
147     else:
148         self.id = name
149         expr = voigt_expr.replace('area', 'area'+self.id).
replace('center', 'center'+self.id).replace('Gfwhm', 'Gfwhm'+
self.id).replace('Lfwhm', 'Lfwhm'+self.id)
150         self.symbols = {}
151         for sym, param in zip(sympy.symbols('center Gfwhm Lfwhm
area'.replace('area', 'area'+self.id).replace('center', '
center'+self.id).replace('Gfwhm', 'Gfwhm'+self.id).replace('
Lfwhm', 'Lfwhm'+self.id)), parameters.keys()):
152             self.symbols[sym] = parameters[param]
153         super().__init__(parameters, expr, self.id, self.symbols
)
154
155 class SplitVoigt(SignalLine):
156     """Takes in the center position, fwhm1 (<= center), fwhm2
(>= center), frac"""
157     def __init__(self, parameters:dict, name:str = None) -> None
:
158         if name is None:
159             self.id = next(id_)
160         else:
161             self.id = name
162             expr = splitvoigt_expr.replace('area', 'area'+self.id).
replace('center', 'center'+self.id).replace('fwhm1', 'fwhm1'+
self.id).replace('fwhm2', 'fwhm2'+self.id).replace('frac', '
frac'+self.id)
163             self.symbols = {}
164             for sym, param in zip(sympy.symbols('center fwhm1 fwhm2
frac area'.replace('area', 'area'+self.id).replace('center', '
center'+self.id).replace('fwhm1', 'fwhm1'+self.id).replace('
fwhm2', 'fwhm2'+self.id).replace('frac', 'frac'+self.id)),
parameters.keys()):
165                 self.symbols[sym] = parameters[param]
166             super().__init__(parameters, expr, self.id, self.symbols
)
167
168
169 class VolumePlasmon(SignalLine):
170     """Inverted Drude-Sellmeier"""
171
172     def __init__(self, parameters:dict, name:str = None) -> None
:
173         if name is None:
174             self.id = next(id_)

```

```

175     else:
176         self.id = name
177         expr = volume_expr.replace('area', 'area'+self.id).
replace('center', 'center'+self.id).replace('fwhm', 'fwhm'+
self.id)
178         self.symbols = {}
179         for sym, param in zip(sympy.symbols('center fwhm area'.
replace('area', 'area'+self.id).replace('center', 'center'+
self.id).replace('fwhm', 'fwhm'+self.id)), parameters.keys())
:
180             self.symbols[sym] = parameters[param]
181         super().__init__(parameters, expr, self.id, self.symbols
)
182
183
184 class DHO(SignalLine):
185     def __init__(self, parameters:dict, name:str = None) -> None
:
186         if name is None:
187             self.id = next(id_)
188         else:
189             self.id = name
190             expr = dho_expr.replace('area', 'area'+self.id).replace(
'center', 'center'+self.id).replace('sigma', 'sigma'+self.id).
replace('gamma', 'gamma'+self.id)
191             self.symbols = {}
192             for sym, param in zip(sympy.symbols('center sigma gamma
area'.replace('area', 'area'+self.id).replace('center', '
center'+self.id).replace('sigma', 'sigma'+self.id).replace('
gamma', 'gamma'+self.id)), parameters.keys()):
193                 self.symbols[sym] = parameters[param]
194             super().__init__(parameters, expr, self.id, self.symbols
)
195
196 class Fano(SignalLine):
197     def __init__(self, parameters:dict, name:str = None) -> None
:
198         if name is None:
199             self.id = next(id_)
200         else:
201             self.id = name
202             expr = fano_expr.replace('area', 'area'+self.id).replace
('center', 'center'+self.id).replace('fwhm', 'fwhm'+self.id).
replace('phi', 'phi'+self.id).replace('background', '
background'+self.id)
203             self.symbols = {}

```

```

204     for sym, param in zip(sympy.symbols('center fwhm area
phi background'.replace('area', 'area'+self.id).replace('
center', 'center'+self.id).replace('fwhm', 'fwhm'+self.id).
replace('phi', 'phi'+self.id).replace('background', '
background'+self.id)), parameters.keys()):
205         self.symbols[sym] = parameters[param]
206     super().__init__(parameters, expr, self.id, self.symbols
)
207
208
209 class DerivativeLorentzian(SignalLine):
210     def __init__(self, parameters:dict, name:str = None) -> None
:
211         if name is None:
212             self.id = next(id_)
213         else:
214             self.id = name
215             expr = derivativeL_expr.replace('area', 'area'+self.id).
replace('center', 'center'+self.id).replace('fwhm1', 'fwhm1'+
self.id).replace('fwhm2', 'fwhm2'+self.id)
216             self.symbols = {}
217             for sym, param in zip(sympy.symbols('center fwhm1 fwhm2
area'.replace('area', 'area'+self.id).replace('center', '
center'+self.id).replace('fwhm1', 'fwhm1'+self.id).replace('
fwhm2', 'fwhm2'+self.id)), parameters.keys()):
218                 self.symbols[sym] = parameters[param]
219             super().__init__(parameters, expr, self.id, self.symbols
)
220
221
222 class SplitLorentzian(SignalLine):
223     def __init__(self, parameters:dict, name:str = None) -> None
:
224         if name is None:
225             self.id = next(id_)
226         else:
227             self.id = name
228             expr = splitLorent_expr.replace('area', 'area'+self.id).
replace('center', 'center'+self.id).replace('fwhm1', 'fwhm1'+
self.id).replace('fwhm2', 'fwhm2'+self.id)
229             self.symbols = {}
230             for sym, param in zip(sympy.symbols('center fwhm1 fwhm2
area'.replace('area', 'area'+self.id).replace('center', '
center'+self.id).replace('fwhm1', 'fwhm1'+self.id).replace('
fwhm2', 'fwhm2'+self.id)), parameters.keys()):
231                 self.symbols[sym] = parameters[param]

```

```

232     super().__init__(parameters, expr, self.id, self.symbols
    )
233
234 class SplitGaussian(SignalLine):
235     def __init__(self, parameters:dict, name:str = None) -> None
    :
236     if name is None:
237         self.id = next(id_)
238     else:
239         self.id = name
240     expr = splitGauss_expr.replace('area', 'area'+self.id).
    replace('center', 'center'+self.id).replace('fwhm1', 'fwhm1'+
    self.id).replace('fwhm2', 'fwhm2'+self.id)
241     self.symbols = {}
242     for sym, param in zip(sympy.symbols('center fwhm1 fwhm2
    area'.replace('area', 'area'+self.id).replace('center', '
    center'+self.id).replace('fwhm1', 'fwhm1'+self.id).replace('
    fwhm2', 'fwhm2'+self.id)), parameters.keys()):
243         self.symbols[sym] = parameters[param]
244     super().__init__(parameters, expr, self.id, self.symbols
    )
245
246
247 class Eps_1(SignalLine):
248     """I dont think it is actually center, but just kept the
    name as it is the plasmon energy"""
249     def __init__(self, parameters:dict, name:str = None) ->
    None:
250     if name is None:
251         self.id = next(id_)
252     else:
253         self.id = name
254     expr = eps_1_expr.replace('area', 'area'+self.id).
    replace('center', 'center'+self.id).replace('fwhm', 'fwhm'+
    self.id)
255     self.symbols = {}
256     for sym, param in zip(sympy.symbols('center fwhm area'.
    replace('area', 'area'+self.id).replace('center', 'center'+
    self.id).replace('fwhm', 'fwhm'+self.id)), parameters.keys())
    :
257         self.symbols[sym] = parameters[param]
258     super().__init__(parameters, expr, self.id, self.symbols
    )
259 class Eps_2(SignalLine):
260     """I dont think it is actually center, but just kept the
    name as it is the plasmon energy"""

```

```

261     def __init__(self, parameters:dict, name:str = None) ->
None:
262         if name is None:
263             self.id = next(id_)
264         else:
265             self.id = name
266             expr = eps_2_expr.replace('area', 'area'+self.id).
replace('center', 'center'+self.id).replace('fwhm', 'fwhm'+
self.id)
267             self.symbols = {}
268             for sym, param in zip(sympy.symbols('center fwhm area'.
replace('area', 'area'+self.id).replace('center', 'center'+
self.id).replace('fwhm', 'fwhm'+self.id)), parameters.keys())
:
269                 self.symbols[sym] = parameters[param]
270             super().__init__(parameters, expr, self.id, self.symbols
)
271
272 class Experimental(SignalLine):
273     """I dont think it is actually center, but just kept the
name as it is the plasmon energy"""
274     def __init__(self, parameters:dict, name:str = None) ->
None:
275         if name is None:
276             self.id = next(id_)
277         else:
278             self.id = name
279             expr = experimental_expr.replace('area', 'area'+self.id)
.replace('center', 'center'+self.id).replace('fwhm', 'fwhm'+
self.id)
280             self.symbols = {}
281             for sym, param in zip(sympy.symbols('center fwhm area'.
replace('area', 'area'+self.id).replace('center', 'center'+
self.id).replace('fwhm', 'fwhm'+self.id)), parameters.keys())
:
282                 self.symbols[sym] = parameters[param]
283             super().__init__(parameters, expr, self.id, self.symbols
)
284
285
286 class LogNormal(SignalLine):
287     def __init__(self, parameters:dict, name:str = None) -> None
:
288         if name is None:
289             self.id = next(id_)
290         else:
291             self.id = name

```

```

292     expr = lognormal_expr.replace('area', 'area'+self.id).
replace('center', 'center'+self.id).replace('sigma', 'sigma'+
self.id)
293     self.symbols = {}
294     for sym, param in zip(sympy.symbols('center sigma area'.
replace('area', 'area'+self.id).replace('center', 'center'+
self.id).replace('sigma', 'sigma'+self.id)), parameters.keys
()):
295         self.symbols[sym] = parameters[param]
296     super().__init__(parameters, expr, self.id, self.symbols
)
297
298
299 class PowerLaw(SignalLine):
300
301     def __init__(self, parameters: dict, name=None) -> None:
302         if name is None:
303             self.id = next(id_)
304         else:
305             self.id = name
306         self.symbols = {}
307
308         for sym, param in zip(sympy.symbols('center k area'.
replace('area', 'area'+self.id).replace('center', 'center'+
self.id).replace('k', 'k'+self.id)), parameters.keys()):
309             self.symbols[sym] = parameters[param]
310             expr = power_expr.replace('area', 'area'+self.id).replace
('k', 'k'+self.id).replace('center', 'center'+self.id)
311
312             super().__init__(parameters, expr, self.id, self.symbols
)
313
314
315 class Constant(SignalLine):
316     """The parameter it takes in must be a dictionary with a key
C. All other parameters are not used.
317
318     TODO: fix the above"""
319     def __init__(self, parameters: dict, name=None) -> None:
320         if name is None:
321             self.id = next(id_)
322         else:
323             self.id = name
324         self.symbols = {}
325         # for sym, param in zip(sympy.symbols('C'.replace('C', 'C
'+self.id)), parameters.keys()):

```



```

326     sym, param = sympy.symbols('C'.replace('C','C'+self.id))
    , parameters['C']
327     self.symbols[sym] = param
328     super().__init__(parameters, 'C'.replace('C','C'+self.id)
    ), self.id, self.symbols)
329
330
331
332 """Model.py"""
333 import numpy as np
334 from multiprocessing.pool import Pool
335 import logging
336 from tqdm.notebook import trange, tqdm
337 import MyFunc
338 from scipy.optimize import curve_fit
339 from signal_lines import *
340 import scripting
341 from skimage.segmentation import slic
342 from time import time
343 from scipy.stats import skew, kurtosis
344 import pickle
345
346
347 hbar = 6.582119569e-16      #eV s
348 e_0   = 8.8541878128e-12   #F/m
349 e     = 1.6e-19
350
351
352
353
354
355
356 class Model:
357     """A model similar to that of hyperspy."""
358     def __init__(self, data, x:np.array=None) -> None:
359         self.data      = data
360         self.signal_lines = {}
361         self.symbols    = {}
362         self.expr        = ''
363         self.masks      = None
364         self.N           = None
365         self.x           = x
366
367
368
369     def append(self, curve: SignalLine)-> None:
370         """Appends a SignalLine to the model"""

```

```

371     if curve.id in self.signal_lines.keys():
372         logging.warning('This is already in the model')
373     self.signal_lines[curve.id] = curve
374
375     for sym in curve.symbols.keys():
376         self.symbols[sym] = curve.symbols[sym]
377     self.expr += '+'*(self.expr!='') + curve.expr
378
379
380     def append_labels(self, masks, N=None):
381         """If N < number of labels the multifit algorithm will
382         only fit for the first N labels. Can also use segment_model
383         to use slic for this purpose"""
384         if N is None:
385             N = masks.max()
386         self.masks = masks
387         self.N      = N
388
389     def segment_model(self, mask=None, n_segements=32,
390 compactness=0.015, enforce=False, **kwargs):
391         """The mask will ultimately make sure that only the TRUE
392         pixels are curve fitted."""
393         img = self.data
394         segments = slic(img, n_segments=n_segements,
395 compactness=compactness, mask=mask, enforce_connectivity=
396 enforce,**kwargs)
397         self.append_labels(segments)
398
399     def multifit(self, num_workers = 4, use_parallel = False,**
400 kwargs) -> np.array:
401         """Fits the appended curves to the dataset using scipy.
402         optimize.curve_fit. Default is not parallelizing"""
403         if self.masks is None:
404             self.masks = np.ones(shape=(self.data.shape[0],self.
405 data.shape[1]))
406             self.N      = 1
407             num_params = len(self.symbols.keys())
408             for i in self.symbols.keys():
409                 if self.symbols[i].bmin == -1:
410                     num_params -=1
411             results = np.zeros(shape=(self.data.shape[0], self.data
412 .shape[1], num_params)) #2 for skew and kurtosis
413             pcov     = np.zeros(shape=(self.data.shape[0], self.data
414 .shape[1], num_params, num_params)) # , dtype=object)
415             iterable = [[self, results,self.masks==i] for i in range
416 (1,self.N+1)]
417             t = []

```

```

406         if not use_parallel:
407
408             for i in range(1,self.N+1):
409                 tic = time()
410
411                 fit(self,results, pcov,self.masks==i, maxfev
=2000, **kwargs)
412                 toc = time()
413                 t.append((toc-tic)/60)
414                 # print(f'Segment {i} took {(toc-tic)/60} min')
415             return results, pcov
416         res = []
417         # the below works, but not with a tqdm
418         with Pool(num_workers) as pool:
419             for result in tqdm(pool.starmap(fit, iterable,
chunks=1)):
420                 res.append(result)
421             pass
422         res = sum([res[i][:,:] for i in range(len(res))])
423         #returns only one array
424         return res
425
426     def compile_function(self) -> None:
427         scripting.compile_function(self)
428
429 def fit(self, result, pcov ,mask = None, maxfev:int = 2000, **
kwargs):
430     if 'weighted' not in kwargs.keys():
431         weighted=False
432     else:
433         weighted=kwargs['weighted']
434     if mask is None:
435         mask = np.ones(shape= (self.data.shape[0], self.data.
shape[1]))
436         self.N = 2
437         param_values = []
438         param_min = []
439         param_max = []
440         for i in self.symbols.keys():
441             if self.symbols[i].bmin == -1:
442                 continue
443             param_values.append(self.symbols[i].value)
444             param_min.append(self.symbols[i].bmin)
445             param_max.append(self.symbols[i].bmax)
446         for i in range(self.data.shape[0]):
447             for j in range(self.data.shape[1]):

```

```

448         # print(mask[i,j]*1.0, (i,j))
449         if mask[i,j]*1.0 == 0:
450             continue
451         if np.any(np.isnan(self.data[i,j])):
452             mask[i,j] = 0
453             logging.info(f'Pixel {i,j} has been removed, as
it was not a number')
454             print(f'Pixel {i,j} has been removed, as it was
not a number')
455             continue
456         try:
457             result[i,j], _ = curve_fit(MyFunc.model_function
, self.x, self.data[i,j], p0=param_values, bounds=(param_min,
param_max), maxfev=maxfev)
458             if weighted:
459                 temp_result = MyFunc.model_function(self.x,*
result[i,j])
460                 result[i,j], pcov[i,j] = curve_fit(MyFunc.
model_function, self.x, self.data[i,j], p0=result[i,j],
bounds=(param_min,param_max), maxfev=maxfev, absolute_sigma=
True, sigma=1/temp_result**2)
461                 param_values = result[i,j] # uses previous
output as input. Improves speed by quite much
462                 if np.any((np.array(param_values) == np.array(
param_min)) )or np.any((np.array(param_values) == np.array(
param_max))):
463                     logging.warning(f'Boundary value reached for
coordinate {(i,j)}')
464
465             except RuntimeError:
466                 logging.warning('Couldnt find an appropriate fit
. Try to improve initialization of increase maxfev')
467                 continue
468                 return result
469
470     return result, pcov
471
472 class Result:
473     def __init__(self, model:Model, result:np.array) -> None:
474         self.model = model
475         self.data = result
476     def print_averages(self, data = None, label = None, mask =
None):
477         if data is None:
478             data = self.data.copy()
479         if label is not None:

```

```

481         data = self.data.copy()
482         data[self.model.masks != label] = np.nan
483     i = 0
484     for sym in self.model.symbols.keys():
485         if self.model.symbols[sym].bmin == -1:
486             print(f'{sym}: {self.model.symbols[sym].value}')
487         else:
488             print(f'{sym}: {np.nanmean(data[:, :, i])}      {np.
nanstd(data[:, :, i])}')
489             i+=1
490
491     def save_model(self, fname:str, mask, skewness=None,
kurtosis=None, in_correct_folder = True):
492         """Skewness and kurtosis is appended after everything
else. Pass it as a tuple of (mean, std)"""
493         if in_correct_folder:
494             fname = f'Results\{fname}'
495         n_array = self.data
496         # print(n_array.shape)
497         results = {}
498         i = 0
499         for sym in self.model.symbols.keys():
500             if self.model.symbols[sym].bmin == -1:
501                 # print(f'{sym}: is fixed')
502                 continue
503                 # print(np.nanmean(n_array[:, :, i][mask]))
504             results[sym.name] = (np.nanmean(n_array[:, :, i][mask
]) , np.nanstd(n_array[:, :, i][mask]))
505             i+=1
506             if skewness is not None:
507                 results['skewness'] = skewness
508             if kurtosis is not None:
509                 results['kurtosis'] = kurtosis
510             f = open(f'{fname}.pkl', 'wb')
511             pickle.dump(results, f)
512             f.close()
513
514     def get_pixel_results(self, x, y):
515         r = []
516         i = 0
517         for sym in self.model.symbols.keys():
518             if self.model.symbols[sym].bmin == -1:
519                 r.append(self.model.symbols[sym].value)
520             else:
521                 r.append(self.data[x, y, i])
522                 i+=1
523         return r

```

```

524
525     def as_dictionary(self, label=None):
526         d = {}
527         if label is not None:
528             data = self.data.copy()
529             data[self.model.masks != label] = np.nan
530         else:
531             data = self.data.copy()
532
533         i = 0
534         for sym in self.model.symbols.keys():
535             if self.model.symbols[sym].bmin == -1:
536                 d[str(sym)] = self.model.symbols[sym].value
537             else:
538                 d[str(sym)] = data[:, :, i]
539                 i+=1
540
541         # for sym,param in zip(self.model.symbols, range(data.
542         shape[-1])):
543             #     d[str(sym)] = data[:, :, param]
544             #     print(f'{sym}: {data[:, :, param]}      {data[:, :,
545             param}}')
546         return d
547
548     def prod_latex_file(self, filename=r'Tables\test_table.tex',
549     label=None, significant_digits = 2):
550         """This function is not the best"""
551
552         if label is not None:
553             data = self.data.copy()
554             data[self.model.masks != label] = np.nan
555         else:
556             data = self.data.copy()
557             file = open(filename, 'w')
558             for sym, param in zip(self.model.symbols.keys(), range(
559             data.shape[-1])):
560                 line = f""{sym} & {np.round(np.nanmean(data[:, :,
561                 param]), significant_digits)} \pm {np.round(np.nanstd(data
562                [:, :, param]), significant_digits)}\ \ n"""
563                 file.write(line)
564             file.close()
565
566 """scripting.py"""

```

```

564 """ Warning: running this file in the same cell (.ipynb) or in
the same .py file as a curve fit procedure won't work. The
file it writes to will only be scripted after the cell/python
-file has been completed."""
565 def compile_function(model, file_name = 'fitting\\MyFunc.py'):
566     with open(file_name, 'w') as file:
567         args = ''
568         for arg in (model.symbols.keys()):
569             if model.symbols[arg].bmin == -1:
570                 continue
571             args += str(arg) + ','
572         file.write('from numpy import pi, sqrt, log, exp, real\n
')
573         file.write('from scipy.special import wofz\n')
574         file.write('from scipy.stats import rv_continuous\n')
575         file.write(f'def model_function(x, {args[:-1]}):\n')
576         line = model.expr
577         for sym in model.symbols.keys():
578             if model.symbols[sym].bmin == -1:
579                 line = line.replace(str(sym), str(model.symbols[
sym].value))
580         file.write(f'\treturn {line}')
581         file.write('\n')
582         file.write(f'class my_func_gen(rv_continuous):\n\tdef
_pdf(self, x, {args[:-1]}):\n\t\t\t')
583         file.write(f'\treturn {line}\n')
584         file.write(f''my_func = my_func_gen(name='my_fun')''')
585
586     print('Function created')
587
588
589
590 # class my_func_gen(rv_continuous):
591 #     "Distribution for Scipy's goodness of fit"
592 #     def _pdf(self, x):
593 #         return np.exp(-x**2 / 2.) / np.sqrt(2.0 * np.pi)
594 # my_func = my_func_gen(name='my_func')
595
596
597
598
599 """concatenate.py
600 The function(s) in this file takes in the lowloss and highloss
from EELS data in the form of two Hyperspy files.
601
602 It thereby concatenates them into one single data file. The
plots are for now merged at 11eV

```

```

603
604 TODO: Fix the ""
605 import numpy as np
606 import hyperspy.api as hs
607 import hyperspy
608 import logging
609 import eels_addon_hyperspy as eah
610
611 merge_energy = 11.                                # eV, where the high- and
        lowloss are merged
612 rrtol          = 1e-3
613
614
615
616 def merge(hl, ll, merge_at:float = merge_energy, rtol:float=
        rrtol, modifier = 'zero_loss', FWHM=0.09, cut_zlp=5.):
617     """hl: Highloss, type hyperspy._signals.eels.EELSSpectrum
618
619     ll: Lowloss, type hyperspy._signals.eels.EELSSpectrum"""
620     # if modifier not in ['zero_loss', 'gaussian', '
        fourier_ratio']:
621         #     NotImplementedError('Only zero_loss and gaussian')
622         assert hl.data.shape[0] == ll.data.shape[0] and hl.data.
        shape[1] == ll.data.shape[1]
623         lowloss = ll.isig[:merge_at*1.0].deepcopy()
624         highloss = hl.isig[merge_at*1.0:].deepcopy()
625         start = highloss.axes_manager['Energy loss'].offset
626         stop = start + highloss.axes_manager['Energy loss'].scale*
        highloss.axes_manager['Energy loss'].size
627         x = np.linspace(start, stop, num=highloss.axes_manager['
        Energy loss'].size)
628         start = lowloss.axes_manager['Energy loss'].offset
629         stop = start + lowloss.axes_manager['Energy loss'].scale*
        lowloss.axes_manager['Energy loss'].size
630         x_zlp = np.linspace(start, stop, num=lowloss.axes_manager['
        Energy loss'].size)
631         x_tot = np.unique(np.concatenate([x_zlp, x]))
632         if not np.all(x_tot == np.sort(x_tot)):
633             logging.warning('x-values not sorted')
634         y = np.zeros(shape=(highloss.data.shape[0], highloss.data.
        shape[1], x_tot.shape[0]))
635         y[:, :, x_tot <= x_zlp[-1]] = lowloss
        #concatenates the counts
636         y[:, :, x_tot >= x_zlp[-1]] = highloss
        #concatenates the counts
637         s_ = hyperspy._signals.eels.EELSSpectrum(y)
638         s_.axes_manager[2].name = 'Energy loss'

```



```

639     s_.axes_manager['Energy loss'].offset = x_tot[0]
640     s_.axes_manager['Energy loss'].scale = (x_tot[-1]-x_tot[0])/
s_.axes_manager['Energy loss'].size
641     s_.axes_manager['Energy loss'].units = 'eV'
642     if not (np.isclose(s_.axes_manager['Energy loss'].scale, hl.
axes_manager['Energy loss'].scale ,rtol) and np.isclose(s_.
axes_manager['Energy loss'].scale, ll.axes_manager['Energy
loss'].scale , rtol)):
643         logging.warning('Scale is off!')
644         if modifier=='zero_loss':
645             return x_tot, s_, s_.fourier_log_deconvolution(lowloss,
add_zlp=False)
646         if modifier=='gaussian':
647             return x_tot, s_, eah.fourier_log_deconvolution(s_,s_.
isig[:cut_zlp*1.0], add_zlp=False, FWHM=FWHM)
648         if modifier=='fourier_ratio':# Not really a modifier, but
its convenient for now
649             return x_tot,s_, s_.fourier_ratio_deconvolution(s_.isig
[:cut_zlp*1.0], fwhm=FWHM, extrapolate_lowloss=False,
extrapolate_coreloss=False)
650         if modifier=='None':
651             return x_tot,s_, s_
652         NotImplementedError('modifier has to be specified. Set it to
str(None) if no deconvolution')
653
654
655
656
657
658
659 """Some handy functions to use in several python scripts"""
660 import numpy as np
661 temperatures = [23,80,120,160,200,240,280,320,360,400,450,500]
662
663
664
665 def create_reconvolve_func():
666     raise NotImplementedError
667
668
669 def fourier_ratio(y:np.array, zlp:np.array,reconvolve_func:np.
array=None):
670     """Returns the Fourier signal of the reconvolved signal"""
671     if reconvolve_func is None:
672         reconvolve_func = create_reconvolve_func()
673     return np.fft.fft(reconvolve_func)*np.fft.fft(y)/np.fft.fft(zlp)

```

```

674
675
676 def produce_filename(T,parameters):
677     # print(parameters)
678     components          = parameters['components']
679     class_type          = str(type(components[0]))[21:-2]
680     # class_type        = parameters['class_type']
681     modifier            = parameters['modifier']
682     if not parameters['deconvolve']:
683         modifier = 'None'
684     energy_interval     = parameters['energy_interval']
685     curve_FWHM         = parameters['curve_FWHM']
686     cut_zlp             = parameters['cut_zlp']
687     add_constant        = parameters['add_constant']
688     align_on_plasmon_peak = parameters['align_on_plasmon_peak']
689     subtract_median     = parameters['subtract_median']
690     add_pixels          = parameters['add_pixels']
691     # T                  = parameters['T']
692     if 'add_to_dict_name' in parameters.keys():
693         addition = parameters['add_to_dict_name']
694         return f'{class_type}_{modifier}_energyInterval{
energy_interval}_FWHM{curve_FWHM}_cutZLP{cut_zlp}_w'+ 'o' * (not
add_constant) + f'constant'+ '_alignedPP_' *
align_on_plasmon_peak+'medianSubtracted'*subtract_median+'
_unweighted' * (not parameters['weighted'])+'_summed_' *
add_pixels+f'{T}C'+f'_{addition}'
695
696     return f'{class_type}_{modifier}_energyInterval{
energy_interval}_FWHM{curve_FWHM}_cutZLP{cut_zlp}_w'+ 'o' * (not
add_constant) + f'constant'+ '_alignedPP_' *
align_on_plasmon_peak+'medianSubtracted'*subtract_median+'
_unweighted' * (not parameters['weighted'])+'_summed_' *
add_pixels+f'{T}C'
697 def get_zlp_thermal():
698     zlp = {}
699     for T in temperatures:
700         avg = []
701         std = []
702         val = []
703         for i in range(1,10):
704             try:
705                 f = f'ZLP/{(T,i)}.npy'
706                 # print(np.any(np.isnan(np.load(f))))
707                 val.append(np.load(f))
708             except:
709                 print(f)
710                 pass

```

```

711     avg = np.mean(np.array(val))
712     std = np.std(np.array(val))
713     zlp[T] = (avg,std)
714     return zlp
715
716 def derivative(y,x, order = 2):
717     """Takes the first derivative. Available for forward
718     differences (order=1) and central differences (order=2)"""
719     dx = (x[-1]-x[0])/x.shape[0]
720     if order==1:
721         return (y[1:] - y[:-1])/dx
722     elif order==2:
723         return np.gradient(y,x)
724     else:
725         raise NotImplementedError('Implementation for higher
726         order finite differences of the first derivative is not
727         implemented')
728
729 def get_zlp_thermal_intensities():
730     zlp = {}
731     for T in temperatures:
732         avg = []
733         std = []
734         val = []
735         for i in range(1,10):
736             try:
737                 f = f'ZLP/{(T,i)}_integral.npy'
738                 # print(np.any(np.isnan(np.load(f))))
739                 val.append(np.load(f))
740             except:
741                 print(f)
742                 pass
743         avg = np.mean(np.array(val))
744         std = np.std(np.array(val))
745         zlp[T] = (avg,std)
746     return zlp
747
748 def line(x,a,b):
749     return a*x+b
750 temp_coef = 0.00429
751 R_Al = 2.65e-8
752 def theoretical(T):
753     return R_Al * (1+temp_coef*(T-(20)))
754
755 mask = np.zeros(shape=(51,60)) # Shape of the images

```

```

755 mask[10:,10:] = 1
756 mask = mask > 0.5
757
758
759 def reshape_segments(segments, approved_files, end_shape):
760     """takes the in the good files and the end-shape of the
       experimental data as well as the segment mask and fits the
       shape appropriatiely"""
761     seg = {}
762     for T in segments.keys():
763         s = np.zeros(end_shape)
764         count = 0
765         for i in range(0,11):
766             if T == 23:
767                 if ('RM',i+1) not in approved_files:
768                     print(T,i)
769                     continue
770             else:
771                 if (T,i+1) not in approved_files:
772                     continue
773                 s[:, :, i] = segments[T][:, :, count]
774                 print(count)
775                 count+=1
776         seg[T] = s
777     return seg
778
779
780 %matplotlib qt # This is an .ipynb file used for
       getting the information regarding the resolution
781 import numpy as np
782 import os
783 import hyperspy.api as hs
784 import hyperspy
785 import matplotlib.pyplot as plt
786 import scipy.ndimage as nd
787 from datetime import date
788 from tqdm import trange, tqdm
789 import logging
790 from time import time
791 import sys
792 sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
       - 2023 V r\\Master\\Data handling\\fitting')
793 import signal_lines
794 import Params
795 import Model
796 import MyFunc

```

```

797 from Model import hbar, e_0          # note that hbar is taken in
    eV
798 import rcparams
799 rcparams.main()                      # sets the rcParams for
    plotting
800
801 # For automatic loading of the scripts
802 # This is needed in order for the MyFunc.py to be updated when
    needed
803 %load_ext autoreload
804 %autoreload 2
805
806 ##### New cell
807
808 ZLP = signal_lines.SplitGaussian(Params.testing_splitG, 'ZLP')
809 ZLP.append_step_func(-1,1)
810 m = Model.Model(np.array(0), np.array(0))
811 m.append(ZLP)
812 m.compile_function()
813
814 ##### New cell
815
816 ##### Loading the data
    #####
817 temperatures = ['80', '120', '160', '200', '240', '280', '320', '360', '
    400', '450', '500']
818 temperatures = [int(T) for T in temperatures]
819 lowloss = {}
820 tic = time()
821 result = {}
822 for T in tqdm([23, 160, 200, 240, 280, 320, 360, 400, 450, 500]):
823     result[T] = []
824     for i in range(1, 10):
825         if T == 23:
826             temp = hs.load(r'Cambridge_7\S1_RMTemp _ 0.01
    s_58nmpixelSTEM SI'+str(i)+'.dm4')
827         else:
828             temp = hs.load(r'Cambridge_7\S1_'+str(T)+'CTemp _
    0.01s_58nmpixel'+str(i)+'.dm4')
829         try:
830             l_ = temp[-2]
831             l_.align_zero_loss_peak(print_stats=False,
    show_progressbar=False, signal_range=(-1., 1.))
832             start = l_.axes_manager['Energy loss'].offset
833             # stop = start + l_.axes_manager['Energy loss'].
    scale*l_.axes_manager['Energy loss'].size

```

```

834         x = start+np.arange(l_.axes_manager['Energy loss'].
size)*l_.axes_manager['Energy loss'].scale
835
836         ZLP = signal_lines.SplitGaussian(Params.
testing_splitG, 'ZLP')
837         m = Model.Model(l_.data,x)
838         m.append(ZLP)
839         # m.compile_function()
840         # m.append(ZLP)
841         res, pcov = m.multifit(show_progressbar=False,
iterpath='serpentine')
842         result[T].append(res)
843         # val = m.components.Gaussian.sigma.map['values']*2*
np.sqrt(2*np.log(2))
844         np.save('ZLP/'+f'{T,i}_SplitG_L', res[:, :,1])
845         np.save('ZLP/'+f'{T,i}_SplitG_R', res[:, :,2])
846         # break
847     except FileNotFoundError:
848         print(i, 'FileNotFoundError')
849     except TypeError:
850         print(i, 'TypeError')
851     except ValueError:
852         print(i, 'ValueError')
853
854
855
856
857
858
859
860 """experimental.py
861
862 The compilation of the signal line may not be optimal
863
864 It is not very efficient as it loads all the data and then curve
fits. experimental2.py was therefore used, but this is left
as it has more options that were attempted, but ultimately
not used"""
865 import numpy as np
866 import os
867 import hyperspy.api as hs
868 import hyperspy
869 import matplotlib.pyplot as plt
870 import scipy.ndimage as nd
871 from datetime import date
872 from tqdm import trange
873 # import logging

```

```

874 import logging
875 import pickle
876 from scipy.stats import linregress
877 from time import time
878 from scipy.stats import skew, kurtosis
879 from scipy.stats import goodness_of_fit
880 import concatenate
881 import sys
882 sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
      - 2023 V r\\Master\\Data handling\\fitting')
883 import signal_lines
884 import Params
885 import Model
886 import utils
887 import MyFunc
888 from Model import hbar, e_0          # note that hbar is taken in
      eV
889 import rcparams
890 rcparams.main()                    # sets the rcParams for
      plotting
891 logging.basicConfig(level=logging.DEBUG)
892 # For automatic loading of the scripts
893 # This is needed in order for the MyFunc.py to be updated when
      needed
894
895 tic = time()
896
897
898 def run_experimental(T, **kwargs):
899     ###
900     ##### Set the parameters here #####
901     T = str(T)
902     if 'energy_interval' in kwargs.keys():
903         energy_interval = kwargs['energy_interval']
904     else:
905         energy_interval = (10.,16.)
906     if 'deconvolve' in kwargs.keys():
907         deconvolve = kwargs['deconvolve']
908     else:
909         deconvolve = False
910     if 'modifier' in kwargs.keys():
911         modifier = kwargs['modifier']
912     else:
913         modifier = 'zero_loss'
914         # modifier = 'gaussian'
915         # modifier = 'fourier_ratio' # not really a
      modifier, but it works as a quick fix

```

```

916     if 'curve_FWHM' in kwargs.keys():
917         curve_FWHM = kwargs['curve_FWHM'] # The
FWHM for the reconvolution function
918     else:
919         curve_FWHM = 0.2
920     if 'cut_zlp' in kwargs.keys():
921         cut_zlp = kwargs['cut_zlp']
922     else:
923         cut_zlp = 2.
924     if 'add_constant' in kwargs.keys():
925         add_constant = kwargs['add_constant']
926     else:
927         add_constant = False
928     if 'subtract_median' in kwargs.keys():
929         subtract_median = kwargs['subtract_median']
930     else:
931         subtract_median = True
932     if 'align_on_plasmon_peak' in kwargs.keys():
933         align_on_plasmon_peak = kwargs['align_on_plasmon_peak']
934     else:
935         align_on_plasmon_peak = False
936     if 'weighted' not in kwargs.keys():
937         kwargs['weighted'] = False
938     if 'add_pixels' in kwargs.keys():
939         add_pixels = kwargs['add_pixels']
940     else:
941         add_pixels=False
942     if 'derivative' in kwargs.keys():
943         derivative = kwargs['derivative']
944     else:
945         derivative = False
946     if 'components' in kwargs.keys():
947         components = kwargs['components']
948     else:
949         if 'signal_line' in kwargs.keys():
950             plasmon = kwargs['signal_line']
951         else:
952             if align_on_plasmon_peak:
953                 para = Params.splitvoigt_PP
954             else:
955                 para = Params.splitvoigt
956             plasmon = signal_lines.SplitVoigt(para, 'Plasmon1')
957             components = [plasmon]
958     if add_constant:
959         c = signal_lines.Constant(Params.const, 'Constant'
)
960         components.append(c)

```



```

961 # class_type = str(type(plasmon))[21:-2]
962 class_type = str(type(components[0]))[21:-2]
963 # dict_name = f'{class_type}_{modifier}_energyInterval{
964 energy_interval}_FWHM{curve_FWHM}_cutZLP{cut_zlp}_w'+ 'o'* (not
add_constant) + f'constant'+ '_alignedPP_'*
align_on_plasmon_peak+'medianSubtracted'*subtract_median+'
_unweighted'* (not kwargs['weighted'])+f'{T}C'
965 dict_name = utils.produce_filename(T,kwargs)
966 # if 'add_to_dict_name' in kwargs.keys():
967 #     dict_name += kwargs['add_to_dict_name']
968 ###
969 ##### Loading the data
#####
970 all_data = []
971 hyper = []
972 hyper_l = []
973 ll = []
974 x = []
975 x_tot = []
976 s = []
977 for i in range(1,11):
978     # print(str(i))
979     if T == 'RM':
980         temp = hs.load(r'Cambridge_7\S1_'+T+'Temp _ 0.01
s_58nmpixelSTEM SI'+str(i)+'.dm4')
981     elif T== '40':
982         temp = hs.load(r'Cambridge_7\S1_'+T+'CTemp _ 0.01
s_58nmpixel.060'+str(i)+'.dm4')
983     else:
984         temp = hs.load(r'Cambridge_7\S1_'+T+'CTemp _ 0.01
s_58nmpixel'+str(i)+'.dm4')
985     try:
986         l_ = temp[-2]
987         h_ = temp[-1]
988         l_.align_zero_loss_peak(also_align=[h_], print_stats
=False, show_progressbar=False, signal_range=(-1.,1.))
989         l = l_.isig[:12.]

        # To make sure that the pixels are defined over same
energy interval
990         h = h_.isig[10.:19.]

        # To make sure that the pixels are defined over same
energy interval
991         if deconvolve:

```

```

992         x_test, _, s_ = concatenate.merge(hl=h_, ll=l_,
modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
993     else:
994         x_test, s_, _ = concatenate.merge(hl=h_, ll=l_,
modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
995         if s_.axes_manager['Energy loss'].offset > -0.4:
996             # print(f'Skipped {i} during loading due to ')
997             logging.debug(f'Skipped {i} during loading due
to offset value')
998             continue
999         x_tot.append(x_test)
1000         if derivative:
1001             s.append(s_.isig[-1.5:20.].isig[:1900].
derivative(axis=-1))
1002             all_data.append(s_.derivative(axis=-1).data)
1003         else:
1004             s.append(s_.isig[-1.5:20.].isig[:1900])
1005             all_data.append(s_.data)
1006             hyper.append( h) # if
not sliced, the different scans will not have the same
dimension
1007             hyper_l.append( l) # if
not sliced, the different scans will not have the same
dimension
1008             # all_data.append(s_.derivative(axis=-1).data)
# used to be h
1009             ll.append( l.data) # if
not sliced, the different scans will not have the same
dimension
1010             # start = h.axes_manager['Energy loss'].offset
# stop = h.axes_manager['Energy loss'].scale*h.
axes_manager['Energy loss'].size + start
1011             # x.append(np.linspace(start, stop, num = h.
axes_manager['Energy loss'].size))
1012             # break
1013         except FileNotFoundError:
1014             print((T,i), 'FileNotFoundError')
1015         except TypeError:
1016             print((T,i), 'TypeError, typically EELS spectrum not
subscriptable')
1017         except ValueError:
1018             print((T,i), 'ValueError')
1019     x = x_tot # overriding to update for new programming
1020     logging.info(f'Straight after loading (and deconvolution if
performed): x_high = {x[0][-1]}, x_low = {x[0][0]}')
1021     # print(f'Straight after loading (and deconvolution if
performed): x_high = {x[0][-1]}, x_low = {x[0][0]}')
1022

```

```

1023     ###
1024
1025     if align_on_plasmon_peak:
1026         temp          = [signal.isig[10:].inav[20:,20:].deepcopy
1027         () for signal in s]
1028         [temp[i].align_zero_loss_peak(print_stats=False,
1029         show_progressbar=False) for i in range(len(temp))]
1030         s = temp
1031
1032     ###
1033     ##### Adding the pixels together
1034     #####
1035     if add_pixels:
1036         offset_low  = np.array([i.axes_manager['Energy loss'].
1037         offset for i in s]).max()
1038         adjusted    = [i.isig[offset_low:] for i in s]
1039         offset_high = np.array([i.axes_manager['Energy loss'].
1040         offset + i.axes_manager['Energy loss'].size * i.axes_manager
1041         ['Energy loss'].scale for i in adjusted]).min()
1042         adjusted    = [i.isig[:offset_high] for i in adjusted]
1043         size = adjusted[0].axes_manager['Energy loss'].size
1044         scale = adjusted[0].axes_manager['Energy loss'].scale
1045         hyper = adjusted[0]
1046         deleted = 0
1047         for i in range(1, len(adjusted)):
1048             try:
1049                 hyper += adjusted[i]
1050             except ValueError:
1051                 print(i, 'ValueError')
1052                 del adjusted[i-deleted]
1053                 deleted+=1
1054                 continue
1055             except IndexError:
1056                 print(i, 'IndexError')
1057
1058         if subtract_median:
1059             all_data = np.array([i.data for i in adjusted]).mean
1060             (axis=0)
1061             all_data -= np.median(all_data, axis=-1).reshape((
1062             all_data.shape[0], all_data.shape[1],1))
1063             all_data = [all_data]
1064         else:
1065             all_data = [np.array([i.data for i in adjusted]).
1066             mean(axis=0)]

```

```

1061     x = [np.linspace(start=offset_low, stop=offset_high, num
1062 =size)]
1063     logging.info(f'After summing over the scans: x_high={x
1064 [0][-1]}, x_low={x[0][0]}')
1065     # print(f'After summing over the scans: x_high={x
1066 [0][-1]}, x_low={x[0][0]}')
1067
1068     ##### Run the curve-fitting algorithm and get the
1069     results #####
1070     # np.ones(shape=(data.shape[0],data.shape[1]))
1071     print('Starting curve fitting')
1072     for i in range(len(all_data)):
1073         if 'mask' in kwargs.keys():
1074             mask = kwargs['mask']
1075         else:
1076             mask = np.zeros((all_data[i].shape[0],all_data[i].
1077 shape[1]))
1078             mask[10:,10:] = 1
1079             mask = mask > 0.5
1080             m = Model.Model(all_data[i],x[i])
1081             skewness = (np.mean(skew(all_data[i][mask]
1082 , bias=
1083 True, axis=-1)), np.std(skew(all_data[i][mask]
1084 , bias=
1085 True, axis=-1)))
1086             kurt = (np.mean(kurtosis(all_data[i][mask], bias=
1087 True, axis=-1)), np.std(kurtosis(all_data[i][mask], bias=
1088 True, axis=-1)))
1089             for comp in components:
1090                 m.append(comp)
1091             if np.any(np.isnan(all_data[i])):
1092                 mask[np.any(np.isnan(all_data[i]), axis=-1)] = False
1093                 m.append_labels(masks=mask)
1094             else:
1095                 m.segment_model(mask = mask)
1096             res, pcov = m.multifit(num_workers=8, use_parallel=False
1097 , **kwargs)
1098
1099                 # TODO: pcov now only for last scan!
1100             r = Model.Result(m,res)
1101             r.save_model(dict_name+str(i) ,mask, kurtosis=kurt,
1102 skewness=skewness)
1103             test = r.data.copy()
1104             j = 0
1105             # print(T,'C')
1106             for sym in r.model.symbols.keys():
1107                 if r.model.symbols[sym].bmin == -1:
1108                     # print(f'{sym}: is fixed to {fwhm_gauss[i]}')
1109                     continue

```

```

1096         logging.debug(f'{T}: {sym}: {np.nanmean(test[:, :, j] [
1097 mask])} {np.nanstd(test[:, :, j] [mask])}')
1098         j+=1
1099         # np.save(f'\numpy_results\{dict_name}', res)
1100         np.save(os.path.join('numpy_results', dict_name+f'_{i}'),
1101 res)
1101         np.save(os.path.join('numpy_results', dict_name+f'_{i}
1102 _std'), pcov)
1102         print('Saved')
1103
1104     ###
1105
1106
1107     # ##### Some plotting #####
1108     # filter_zlp = 500*(not align_on_plasmon_peak)
1109     # divide = 2
1110     # i, j = -1, -2
1111     # y = m.data[i:, j:].mean(axis=(0,1))-m.data[i:, j:].mean(axis
1112 = (0,1)) [-1000:].max()/divide
1113     # y = y[filter_zlp:].copy()
1114     # # print(np.where(y==y.max()))
1115     # if not modifier=='fourier_ratio':
1116     #     max = np.where(y==y.max())[0][0]
1117     #     # print(max[0])
1118     #     x_low_ = np.where(np.abs(y[:max])==np.abs(y[:max]).min
1119 ()) [0][0]
1120     #     x_high_ = np.where(np.abs(y[max:]) == np.abs(y[max:]).min
1121 ()) [0][0]+max
1122
1123     #     x_low = x[-1][x_low_+filter_zlp]
1124     #     x_high = x[-1][x_high_+filter_zlp]
1125
1126     #     logging.debug(T+'C', 'Numerical FWHM:', (x_high-x_low)
1127 , 'eV')
1128     #     np.save(os.path.join('numpy_results', dict_name+'
1129 numerical'), np.array(x_high-x_low))
1130
1131     # plot_s = slice(1500, -100)
1132
1133     # ##### Plotting the fit #####
1134
1135     # plt.figure()
1136     # i, j = -3, -3
1137     # plt.scatter(x[-1][plot_s], m.data[i, j][plot_s]-m.data[i, j
1138 ] [-1000:].max()
1139 /
1140 divide, s=1, alpha=1, label = '1 Pixel')

```

```

1133     # # plt.scatter()
1134     # plt.scatter(x[-1][plot_s],m.data[i:,j:,plot_s].mean(axis
=(0,1))-m.data[i:,j:].mean(axis=(0,1))[-1000:].max() /
divide, s=1, alpha=1, label='Multiple pixels')
1135     # plt.scatter(x[-1][plot_s],m.data[mask,plot_s].mean(axis
=(0)) - m.data[mask].mean(axis=(0))[-1000:].max() /
divide , s=1, alpha=1, label='All pixels')
1136     # plt.plot(x[-1][plot_s] , MyFunc.model_function(x[-1][
plot_s],*res[i,j])- MyFunc.model_function(x[-1][plot_s],*res[
i,j])).max()/divide , label='Other' ,color='red')
1137
1138     # if not modifier=='fourier_ratio':
1139     #     plt.scatter(x_low, 0)
1140     #     plt.scatter(x_high,0)
1141     # plt.plot(x[-1][plot_s], np.zeros(x[-1][plot_s].shape)
, '--')
1142     # # plt.scatter(x[-1],m.data[i:,j:].mean(axis=(0,1))-MyFunc.
model_function(x[-1],*res[i,j]))
1143     # plt.title(f' {i,j}'+'_FWHM='+ str(curve_FWHM) +'_'+'T+'+'C')
1144     # plt.legend()
1145     # plt.savefig(f'Figures\Curve_fit\{dict_name}.png')
1146     # # for sym,val in zip(m.symbols, res[i,j]):
1147     # #     print(sym,val)
1148
1149     # known_params = {}
1150     # count = 0
1151     # for sym,val in zip(m.symbols, res[i,j]):
1152     #     # print(sym,val)
1153     #     known_params[str(sym)] = (val, count)
1154     #     count+=1
1155
1156     # ###
1157     # # ##### Plotting the distribution of a parameter #####
1158     # if class_type=='Lorentzian' or class_type =='Fano':
1159     #     param = 'fwhmPlasmon1'
1160     #     param_index = known_params[param][-1]
1161     #     plt.figure()
1162     #     plt.hist(res[:, :, param_index][mask], bins=50)
1163     #     plt.title(param+'_FWHM='+ str(curve_FWHM) +'_'+'T+'+'C')
1164     #     plt.savefig(f'Figures\{dict_name}.png')
1165     #     # plt.colorbar()
1166
1167     # if class_type=='SplitVoigt':
1168     #     param = 'fwhm2Plasmon1'
1169     #     param_index = known_params[param][-1]
1170     #     plt.figure()

```

```

1171 # plt.hist(res[:, :, param_index][mask], bins=50, range
    =(0,1))
1172 # plt.title(param)
1173 # plt.savefig(f'Figures\{dict_name}.png')
1174 # # plt.colorbar()
1175 # None
1176 # param = 'fwhm1Plasmon1'
1177 # param_index = known_params[param][-1]
1178 # plt.figure()
1179 # plt.hist(res[:, :, param_index][mask], bins=50, range
    =(0,1))
1180 # plt.title(param)
1181 # plt.savefig(f'Figures\{dict_name}.png')
1182 # # plt.colorbar()
1183 # ###
1184 # ##### Plotting the map over the values #####
1185 # plt.figure()
1186 # plt.imshow(res[:, :, param_index])
1187 # plt.title(param)
1188
1189
1190
1191 # toc = time()
1192 # print(f'At {T}C, it took {(toc-tic)/60} min')
1193
1194 # index_a = 1500
    #TODO: improve these
1195 # index_b = 1700
    #TODO: improve these
1196 # assert modifier == 'zero_loss' or not deconvolve or
    modifier=='fourier_ratio'
1197 # i, j=30, 30
1198 # reg = (linregress(MyFunc.model_function(x[-1][index_a:
    index_b], *res[i, j]), m.data[i, j, index_a:index_b]))
1199 # reg_ = np.zeros(shape=mask.shape, dtype=object)
1200 # for i in range(res.shape[0]):
1201 #     for j in range(res.shape[1]):
1202 #         if not mask[i, j]:
1203 #             continue
1204 #         reg_[i, j] = (linregress(MyFunc.model_function(x
    [-1][index_a:index_b], *res[i, j]), m.data[i, j, index_a:index_b]
    ))
1205 # np.save(os.path.join('Lin_results', dict_name), reg_)
1206
1207
1208
1209

```

```

1210
1211 """ experimental2.py
1212 Some parameters not availavle compared to experimental.py, but
      deals with memory better"""
1213 import numpy as np
1214 import os
1215 import hyperspy.api as hs
1216 import hyperspy
1217 import matplotlib.pyplot as plt
1218 import scipy.ndimage as nd
1219 from datetime import date
1220 from tqdm import trange
1221 # import logging
1222 import logging
1223 import pickle
1224 from scipy.stats import linregress
1225 from time import time
1226 from scipy.stats import skew, kurtosis
1227 from scipy.stats import goodness_of_fit
1228 import concatenate
1229 import sys
1230 sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
      - 2023 V r\\Master\\Data handling\\fitting')
1231 import signal_lines
1232 import Params
1233 import Model
1234 import utils
1235 import MyFunc
1236 from Model import hbar, e_0          # note that hbar is taken in
      eV
1237 import rcparams
1238 rcparams.main()                    # sets the rcParams for
      plotting
1239
1240
1241
1242 tic = time()
1243 prel_mask = utils.mask             # preliminary mask, taking out vacuum
      and contamination
1244
1245 def run_experimental(T, **kwargs):
1246     ###
1247     ##### Set the parameters here #####
1248     T = str(T)
1249     if 'energy_interval' in kwargs.keys():
1250         energy_interval = kwargs['energy_interval']
1251     else:

```



```

1252     energy_interval = (10.,16.)
1253     if 'deconvolve' in kwargs.keys():
1254         deconvolve = kwargs['deconvolve']
1255     else:
1256         deconvolve = False
1257     if 'modifier' in kwargs.keys():
1258         modifier = kwargs['modifier']
1259     else:
1260         modifier          = 'zero_loss'
1261         # modifier          = 'gaussian'
1262         # modifier          = 'fourier_ratio' # not really a
modifier, but it works as a quick fix
1263     if 'curve_FWHM' in kwargs.keys():
1264         curve_FWHM = kwargs['curve_FWHM']           # The
FWHM for the reconvolution function
1265     else:
1266         curve_FWHM      = 0.2
1267     if 'cut_zlp' in kwargs.keys():
1268         cut_zlp = kwargs['cut_zlp']
1269     else:
1270         cut_zlp          = 2.
1271     if 'add_constant' in kwargs.keys():
1272         add_constant = kwargs['add_constant']
1273     else:
1274         add_constant    = False
1275     if 'subtract_median' in kwargs.keys():
1276         subtract_median = kwargs['subtract_median']
1277     else:
1278         subtract_median = True
1279     if 'align_on_plasmon_peak' in kwargs.keys():
1280         align_on_plasmon_peak = kwargs['align_on_plasmon_peak']
1281     else:
1282         align_on_plasmon_peak = False
1283     if 'weighted' not in kwargs.keys():
1284         kwargs['weighted'] = False
1285     if 'add_pixels' in kwargs.keys():
1286         add_pixels = kwargs['add_pixels']
1287     else:
1288         add_pixels=False
1289     if 'derivative' in kwargs.keys():
1290         derivative = kwargs['derivative']
1291     else:
1292         derivative = False
1293     if 'components' in kwargs.keys():
1294         components = kwargs['components']
1295     else:
1296         if 'signal_line' in kwargs.keys():

```

```

1297     plasmon = kwargs['signal_line']
1298     else:
1299         if align_on_plasmon_peak:
1300             para = Params.splitvoigt_PP
1301         else:
1302             para = Params.splitvoigt
1303             plasmon = signal_lines.SplitVoigt(para, 'Plasmon1')
1304             components = [plasmon]
1305     if add_constant:
1306         c = signal_lines.Constant(Params.const, 'Constant'
1307 )
1308         components.append(c)
1309
1310     # class_type = str(type(plasmon))[21:-2]
1311     class_type = str(type(components[0]))[21:-2]
1312     # dict_name = f'{class_type}_{modifier}_energyInterval{
1313     energy_interval}_FWHM{curve_FWHM}_cutZLP{cut_zlp}_w'+ 'o' * (not
1314     add_constant) + f'constant'+ '_alignedPP_' *
1315     align_on_plasmon_peak + 'medianSubtracted' * subtract_median +
1316     '_unweighted' * (not kwargs['weighted']) + f'{T}C'
1317     dict_name = utils.produce_filename(T, kwargs)
1318     x = []
1319     x_tot = []
1320     s = []
1321     for i in range(1, 11):
1322         # print(str(i))
1323         if T == 'RM':
1324             fname = r'Cambridge_7\S1_'+T+'Temp _ 0.01
1325 s_58nmpixelSTEM SI'+str(i)+'.dm4'
1326         elif T == '40':
1327             fname = r'Cambridge_7\S1_'+T+'CTemp _ 0.01
1328 s_58nmpixel.060'+str(i)+'.dm4'
1329         else:
1330             fname = r'Cambridge_7\S1_'+T+'CTemp _ 0.01
1331 s_58nmpixel'+str(i)+'.dm4'
1332         try:
1333             temp = hs.load(fname)
1334             l_ = temp[-2]
1335             h_ = temp[-1]
1336             l_.align_zero_loss_peak(also_align=[h_], print_stats
1337 =False, show_progressbar=False, signal_range=(-1., 1.))
1338             # l = l_.isig[:12.]
1339
1340             # To make sure that the pixels are defined over
1341             same energy interval
1342             # h = h_.isig[10.:19.]

```

```

# To make sure that the pixels are defined over
same energy interval
1331     if deconvolve:
1332         x_test, _, scan = concatenate.merge(hl=h_, ll=l_,
modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
1333     else:
1334         x_test, scan, _ = concatenate.merge(hl=h_, ll=l_,
modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
1335         if scan.axes_manager['Energy loss'].offset > -0.4:
1336             print(f'Skipped {(T,i)} during loading due to
offset value')
1337             logging.debug(f'Skipped {i} during loading due
to offset value')
1338             continue
1339         scan = scan.isig[-1.5:20.].isig[:1900]
1340         if derivative:
1341             scan = scan.derivative(axis=-1)
1342
1343         ##### Switching to np arrays
1344         x = scan.axes_manager['Energy loss'].offset +scan.
axes_manager['Energy loss'].scale *np.arange(scan.
axes_manager['Energy loss'].size)
1345         data = scan.data
1346
1347         ##### Loading and
deconvolving and derivative done, starting curve fit
#####
1348         if 'mask' in kwargs.keys():
1349             mask = kwargs['mask']
1350         else:
1351             mask = prel_mask
1352         print(data.shape, x.shape)
1353         m = Model.Model(data,x)
1354         for comp in components:
1355             m.append(comp)
1356         if np.any(np.isnan(data)):
1357             mask[np.any(np.isnan(data), axis=-1)] = False
1358             m.append_labels(masks=mask)
1359         else:
1360             m.segment_model(mask = mask)
1361         res, pcov = m.multifit(num_workers=8, use_parallel=
False, **kwargs)
1362         r = Model.Result(m,res)
1363         r.save_model(dict_name+str(i) ,mask)
1364         for j,sym in enumerate(r.model.symbols.keys()):
1365             if r.model.symbols[sym].bmin == -1:
1366                 continue

```

```

1367         logging.debug(f'{T}: {sym}: {np.nanmean(r.data
1368        [:, :, j][mask])} {np.nanstd(r.data[:, :, j][mask])}')
1369         j+=1
1370         np.save(os.path.join('numpy_results', dict_name+f'_{i
1371         }'), res)
1372         np.save(os.path.join('numpy_results', dict_name+f'_{i
1373         }_std'), pcov)
1374         print(f'Saved {(T,i)}')
1375     except FileNotFoundError:
1376         print((T,i), 'FileNotFoundError')
1377     except TypeError:
1378         print((T,i), 'TypeError, typically EELS spectrum not
1379         subscriptable')
1380     except ValueError as e:
1381         print((T,i), 'ValueError:' , e)
1382
1383
1384 """plot_class.py
1385 Big thanks to Emil F. Christiansen for providing a notebook this
1386 .py file is heavily based on. Changed slightly to suit my
1387 purposes better"""
1388 import matplotlib.pyplot as plt
1389 from matplotlib.colors import SymLogNorm
1390 import seaborn as sns
1391 import pandas as pd
1392
1393 class SplitLorentzSegmented():
1394     """
1395     A class for storing Lorentz fit parameters of a plasmon peak
1396     after segmentation
1397     """
1398     def __init__(self, data, name=None, temperature=None):
1399         """
1400         Create a VoigtFit instance
1401
1402         Parameters:
1403         -----
1404         data: numpy.ndarray of shape (M, 4)
1405             Data array with plasmon energy, left fwhm, right
1406             fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of
1407             third axis, respectively.
1408         name: str
1409             The name of the dataset

```

```

1405     temperature: float
1406         The temperature used in the experiment in K
1407     """
1408     self.data = data
1409     self.name = name
1410     self.temperature = temperature
1411     self._parameter_mapping = {'Ep': self.energy, 'fwhm1':
self.fwhm1, 'fwhm2': self.fwhm2, 'integral': self.integral}
1412
1413     def __repr__(self):
1414         return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'
1415
1416     def __str__(self):
1417         return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'
1418
1419     @property
1420     def energy(self):
1421         return self.data[:, 0]
1422
1423     @property
1424     def fwhm1(self):
1425         return self.data[:, 1]
1426
1427     @property
1428     def fwhm2(self):
1429         return self.data[:, 2]
1430
1431     @property
1432     def integral(self):
1433         return self.data[:, 3]
1434
1435
1436
1437     @property
1438     def dataframe(self):
1439         df = pd.DataFrame(self.data.reshape((self.data.shape[0],
self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', '
integral'])
1440         df.insert(0, 'T', self.temperature)
1441         return df
1442
1443     def get_parameter(self, parameter):
1444         """
1445         Return a parameter from the fit
1446         """

```

```

1447     return self._parameter_mapping.get(parameter, None)
1448
1449     def plot(self, kind, parameters, *args, **kwargs):
1450         """
1451         Plot the fit results
1452
1453         Parameters:
1454         -----
1455         kind: str
1456             The kind of plot. Should be either "img" or "hist"
1457         parameters: str or list
1458             The parameter(s) to plot.
1459         *args: Optional arguments passed to plotting functions
1460         **kwargs: Optional keyword arguments passed to plotting
1461         functions
1462
1463         Returns:
1464         -----
1465         Returns the figure and axes generated by the plotting
1466         functions
1467         """
1468         if parameters is None:
1469             parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
1470         else:
1471             if isinstance(parameters, str):
1472                 parameters = [parameters]
1473             else:
1474                 parameters = list(parameters)
1475
1476         if kind == 'img':
1477             fig, axes = plt.subplots(ncols=len(parameters),
1478             nrows=1, sharex=True, sharey=True)
1479             fig.suptitle(f'{self.name}')
1480             if len(parameters) == 1:
1481                 axes = list([axes])
1482             kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
1483             )
1484             for ax, parameter in zip(axes, parameters):
1485                 ax.imshow(self.get_parameter(parameter), *args,
1486                 **kwargs)
1487                 ax.set_title(f'{parameter}')
1488         elif kind == 'hist':
1489             fig, axes = plt.subplots(ncols=len(parameters),
1490             nrows=1)
1491             fig.suptitle(f'{self.name}')
1492             if len(parameters) == 1:

```

```

1488         axes = list([axes])
1489         df = self.dataframe
1490         for ax, parameter in zip(axes, parameters):
1491             sns.histplot(df, x=parameter, ax=ax, **kwargs)
1492         else:
1493             return NotImplementedError(f'Kind {kind} is not
1494 implemented')
1495         return fig, axes
1496
1497 class VoigtSegmented():
1498     """
1499     A class for storing Voigt fit parameters of a plasmon peak
1500     after segmentation
1501     """
1502     def __init__(self, data, name=None, temperature=None):
1503         """
1504         Create a VoigtFit instance
1505
1506         Parameters:
1507         -----
1508         data: numpy.ndarray of shape (M, 4)
1509             Data array with plasmon energy, Gaussian fwhm,
1510             Lorentzian fwhm and integral (area) in 1st, 2nd, 3rd and 4th
1511             position of third axis, respectively.
1512         name: str
1513             The name of the dataset
1514         temperature: float
1515             The temperature used in the experiment in K
1516         """
1517         self.data = data
1518         self.name = name
1519         self.temperature = temperature
1520         self._parameter_mapping = {'Ep': self.energy, 'Gfwhm':
1521 self.Gfwhm, 'Lfwhm': self.Lfwhm, 'integral': self.integral}
1522
1523     def __repr__(self):
1524         return f'{self.__class__.__name__}({self.data!r}, name={
1525 self.name!r}, temperature={self.temperature!r})'
1526
1527     def __str__(self):
1528         return f'{self.__class__.__name__} with shape {self.data
1529 .shape}:\n{self.dataframe.describe()}'
1530
1531 @property
1532 def energy(self):
1533     return self.data[:, 0]

```

```

1528 @property
1529 def Gfwhm(self):
1530     return self.data[:, 1]
1531
1532 @property
1533 def Lfwhm(self):
1534     return self.data[:, 2]
1535
1536 @property
1537 def integral(self):
1538     return self.data[:, 3]
1539
1540
1541
1542 @property
1543 def dataframe(self):
1544     df = pd.DataFrame(self.data.reshape((self.data.shape[0],
1545                                         self.data.shape[1])), columns=['Ep', 'Gfwhm', 'Lfwhm', '
1546     integral'])
1547     df.insert(0, 'T', self.temperature)
1548     return df
1549
1550 def get_parameter(self, parameter):
1551     """
1552     Return a parameter from the fit
1553     """
1554     return self._parameter_mapping.get(parameter, None)
1555
1556 def plot(self, kind, parameters, *args, **kwargs):
1557     """
1558     Plot the fit results
1559
1560     Parameters:
1561     -----
1562     kind: str
1563         The kind of plot. Should be either "img" or "hist"
1564     parameters: str or list
1565         The parameter(s) to plot.
1566     *args: Optional arguments passed to plotting functions
1567     **kwargs: Optional keyword arguments passed to plotting
1568     functions
1569
1570     Returns:
1571     -----
1572     Returns the figure and axes generated by the plotting
1573     functions
1574     """

```



```

1571     if parameters is None:
1572         parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']
1573     else:
1574         if isinstance(parameters, str):
1575             parameters = [parameters]
1576         else:
1577             parameters = list(parameters)
1578
1579
1580     if kind == 'img':
1581         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1, sharex=True, sharey=True)
1582         fig.suptitle(f'{self.name}')
1583         if len(parameters) == 1:
1584             axes = list([axes])
1585         kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
)
1586         for ax, parameter in zip(axes, parameters):
1587             ax.imshow(self.get_parameter(parameter), *args,
**kwargs)
1588             ax.set_title(f'{parameter}')
1589     elif kind == 'hist':
1590         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1)
1591         fig.suptitle(f'{self.name}')
1592         if len(parameters) == 1:
1593             axes = list([axes])
1594         df = self.dataframe
1595         for ax, parameter in zip(axes, parameters):
1596             sns.histplot(df, x=parameter, ax=ax, **kwargs)
1597     else:
1598         return NotImplementedError(f'Kind {kind} is not
implemented')
1599     return fig, axes
1600
1601
1602
1603 class SplitVoigtSegmented():
1604     """
1605     A class for storing Lorentz fit parameters of a plasmon peak
after segmentation
1606     """
1607     def __init__(self, data, name=None, temperature=None):
1608         """
1609         Create a VoigtFit instance
1610
1611         Parameters:

```

```

1612     -----
1613     data: numpy.ndarray of shape (M, 4)
1614     Data array with plasmon energy, left fwhm, right
1615     fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of
1616     third axis, respectively.
1617     name: str
1618     The name of the dataset
1619     temperature: float
1620     The temperature used in the experiment in K
1621     """
1622     self.data = data
1623     self.name = name
1624     self.temperature = temperature
1625     self._parameter_mapping = {'Ep': self.energy, 'fwhm1':
1626     self.fwhm1, 'fwhm2': self.fwhm2, 'eta' : self.eta, 'integral'
1627     : self.integral}
1628
1629     def __repr__(self):
1630         return f'{self.__class__.__name__}({self.data!r}, name={
1631         self.name!r}, temperature={self.temperature!r})'
1632
1633     def __str__(self):
1634         return f'{self.__class__.__name__} with shape {self.data
1635         .shape}:\n{self.dataframe.describe()}'
1636
1637     @property
1638     def energy(self):
1639         return self.data[:, 0]
1640
1641     @property
1642     def fwhm1(self):
1643         return self.data[:, 1]
1644
1645     @property
1646     def fwhm2(self):
1647         return self.data[:, 2]
1648
1649     @property
1650     def eta(self):
1651         return self.data[:, 3]
1652
1653     @property
1654     def integral(self):
1655         return self.data[:, 4]

```

```

1653 @property
1654 def dataframe(self):
1655     df = pd.DataFrame(self.data.reshape((self.data.shape[0],
1656     self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', 'eta'
1657     , 'integral'])
1658     df.insert(0, 'T', self.temperature)
1659     return df
1660
1661 def get_parameter(self, parameter):
1662     """
1663     Return a parameter from the fit
1664     """
1665     return self._parameter_mapping.get(parameter, None)
1666
1667 def plot(self, kind, parameters, *args, **kwargs):
1668     """
1669     Plot the fit results
1670
1671     Parameters:
1672     -----
1673     kind: str
1674         The kind of plot. Should be either "img" or "hist"
1675     parameters: str or list
1676         The parameter(s) to plot.
1677     *args: Optional arguments passed to plotting functions
1678     **kwargs: Optional keyword arguments passed to plotting
1679     functions
1680
1681     Returns:
1682     -----
1683     Returns the figure and axes generated by the plotting
1684     functions
1685     """
1686     if parameters is None:
1687         parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
1688     else:
1689         if isinstance(parameters, str):
1690             parameters = [parameters]
1691         else:
1692             parameters = list(parameters)
1693
1694     if kind == 'img':
1695         fig, axes = plt.subplots(ncols=len(parameters),
1696         nrows=1, sharex=True, sharey=True)
1697         fig.suptitle(f'{self.name}')
1698         if len(parameters) == 1:

```

```

1695         axes = list([axes])
1696         kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
)
1697         for ax, parameter in zip(axes, parameters):
1698             ax.imshow(self.get_parameter(parameter), *args,
**kwargs)
1699             ax.set_title(f'{parameter}')
1700     elif kind == 'hist':
1701         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1)
1702         fig.suptitle(f'{self.name}')
1703         if len(parameters) == 1:
1704             axes = list([axes])
1705             df = self.dataframe
1706             for ax, parameter in zip(axes, parameters):
1707                 sns.histplot(df, x=parameter, ax=ax, **kwargs)
1708         else:
1709             return NotImplementedError(f'Kind {kind} is not
implemented')
1710         return fig, axes
1711
1712
1713 class FanoSegmented():
1714     """
1715     A class for storing Fano fit parameters of a plasmon peak
after segmentation
1716     """
1717     def __init__(self, data, name=None, temperature=None):
1718         """
1719         Create a VoigtFit instance
1720
1721         Parameters:
1722         -----
1723         data: numpy.ndarray of shape (M, 4)
1724             Data array with plasmon energy, fwhm, integral (area
), phi and background in 1st, 2nd, 3rd, 4th and 5th position
of third axis, respectively.
1725         name: str
1726             The name of the dataset
1727         temperature: float
1728             The temperature used in the experiment in K
1729         """
1730         self.data = data
1731         self.name = name
1732         self.temperature = temperature
1733         self._parameter_mapping = {'Ep': self.energy, 'fwhm1':
self.fwhm1, 'phi': self.phi, 'integral': self.integral, '

```

```

background' : self.background}
1734
1735     def __repr__(self):
1736         return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'
1737
1738     def __str__(self):
1739         return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'
1740
1741     @property
1742     def energy(self):
1743         return self.data[:, 0]
1744
1745     @property
1746     def fwhm1(self):
1747         return self.data[:, 1]
1748
1749
1750     @property
1751     def integral(self):
1752         return self.data[:, 2]
1753
1754     @property
1755     def phi(self):
1756         return self.data[:, 3]
1757
1758
1759     @property
1760     def background(self):
1761         return self.data[:, 4]
1762
1763
1764
1765     @property
1766     def dataframe(self):
1767         df = pd.DataFrame(self.data.reshape((self.data.shape[0],
self.data.shape[1])), columns=['Ep', 'fwhm1', 'integral', '
phi', 'background'])
1768         df.insert(0, 'T', self.temperature)
1769         return df
1770
1771     def get_parameter(self, parameter):
1772         """
1773         Return a parameter from the fit
1774         """
1775         return self._parameter_mapping.get(parameter, None)

```

```

1776
1777 def plot(self, kind, parameters, *args, **kwargs):
1778     """
1779     Plot the fit results
1780
1781     Parameters:
1782     -----
1783     kind: str
1784         The kind of plot. Should be either "img" or "hist"
1785     parameters: str or list
1786         The parameter(s) to plot.
1787     *args: Optional arguments passed to plotting functions
1788     **kwargs: Optional keyword arguments passed to plotting
1789     functions
1790
1791     Returns:
1792     -----
1793     Returns the figure and axes generated by the plotting
1794     functions
1795     """
1796     if parameters is None:
1797         parameters = ['Ep', 'fwhm1', 'phi', 'integral', '
1798 background']
1799     else:
1800         if isinstance(parameters, str):
1801             parameters = [parameters]
1802         else:
1803             parameters = list(parameters)
1804
1805     if kind == 'img':
1806         fig, axes = plt.subplots(ncols=len(parameters),
1807 nrows=1, sharex=True, sharey=True)
1808         fig.suptitle(f'{self.name}')
1809         if len(parameters) == 1:
1810             axes = list([axes])
1811         kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
1812 )
1813         for ax, parameter in zip(axes, parameters):
1814             ax.imshow(self.get_parameter(parameter), *args,
1815 **kwargs)
1816             ax.set_title(f'{parameter}')
1817     elif kind == 'hist':
1818         fig, axes = plt.subplots(ncols=len(parameters),
1819 nrows=1)
1820         fig.suptitle(f'{self.name}')
1821         if len(parameters) == 1:

```

```

1816         axes = list([axes])
1817         df = self.dataframe
1818         for ax, parameter in zip(axes, parameters):
1819             sns.histplot(df, x=parameter, ax=ax, **kwargs)
1820     else:
1821         return NotImplementedError(f'Kind {kind} is not
implemented')
1822     return fig, axes
1823
1824
1825
1826
1827 class VolumePlasmon():
1828     """
1829     A class for storing Voigt fit parameters of a plasmon peak
after segmentation
1830     """
1831     def __init__(self, data, name=None, temperature=None):
1832         """
1833         Create a Volume Plasmon instance
1834
1835         Parameters:
1836         -----
1837         data: numpy.ndarray of shape (M, 4)
1838             Data array with plasmon energy, fwhm, and integral (
area) in 1st, 2nd and 3rd position of third axis,
respectively.
1839         name: str
1840             The name of the dataset
1841         temperature: float
1842             The temperature used in the experiment in C
1843         """
1844         self.data = data
1845         self.name = name
1846         self.temperature = temperature
1847         self._parameter_mapping = {'Ep': self.energy, 'fwhm':
self.fwhm, 'integral': self.integral}
1848
1849     def __repr__(self):
1850         return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'
1851
1852     def __str__(self):
1853         return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'
1854
1855     @property

```

```

1856 def energy(self):
1857     return self.data[:, 0]
1858
1859 @property
1860 def fwhm(self):
1861     return self.data[:, 1]
1862
1863 @property
1864 def integral(self):
1865     return self.data[:, 2]
1866
1867
1868
1869 @property
1870 def dataframe(self):
1871     df = pd.DataFrame(self.data.reshape((self.data.shape[0],
1872     self.data.shape[1])), columns=['Ep', 'fwhm', 'integral'])
1873     df.insert(0, 'T', self.temperature)
1874     return df
1875
1876 def get_parameter(self, parameter):
1877     """
1878     Return a parameter from the fit
1879     """
1880     return self._parameter_mapping.get(parameter, None)
1881
1882 def plot(self, kind, parameters, *args, **kwargs):
1883     """
1884     Plot the fit results
1885
1886     Parameters:
1887     -----
1888     kind: str
1889         The kind of plot. Should be either "img" or "hist"
1890     parameters: str or list
1891         The parameter(s) to plot.
1892     *args: Optional arguments passed to plotting functions
1893     **kwargs: Optional keyword arguments passed to plotting
1894     functions
1895
1896     Returns:
1897     -----
1898     Returns the figure and axes generated by the plotting
1899     functions
1900     """
1901     if parameters is None:
1902         parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']

```



```

1900     else:
1901         if isinstance(parameters, str):
1902             parameters = [parameters]
1903         else:
1904             parameters = list(parameters)
1905
1906
1907     if kind == 'img':
1908         fig, axes = plt.subplots(ncols=len(parameters),
1909     nrows=1, sharex=True, sharey=True)
1910         fig.suptitle(f'{self.name}')
1911         if len(parameters) == 1:
1912             axes = list([axes])
1913         kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
1914     )
1915         for ax, parameter in zip(axes, parameters):
1916             ax.imshow(self.get_parameter(parameter), *args,
1917     **kwargs)
1918             ax.set_title(f'{parameter}')
1919     elif kind == 'hist':
1920         fig, axes = plt.subplots(ncols=len(parameters),
1921     nrows=1)
1922         fig.suptitle(f'{self.name}')
1923         if len(parameters) == 1:
1924             axes = list([axes])
1925         df = self.dataframe
1926         for ax, parameter in zip(axes, parameters):
1927             sns.histplot(df, x=parameter, ax=ax, **kwargs)
1928     else:
1929         return NotImplementedError(f'Kind {kind} is not
1930     implemented')
1931     return fig, axes
1932
1933
1934 """plot_class.py
1935 Big thanks to Emil F. Christiansen for providing a notebook this
1936     .py file is heavily based on. Changed slightly to suit my
1937     purposes better"""
1938
1939 import matplotlib.pyplot as plt
1940 from matplotlib.colors import SymLogNorm
1941 import seaborn as sns
1942 import pandas as pd
1943
1944 class SplitLorentzSegmented():
1945     """
1946     A class for storing Lorentz fit parameters of a plasmon peak
1947     after segmentation

```

```

1939     """
1940     def __init__(self, data, name=None, temperature=None):
1941         """
1942         Create a VoigtFit instance
1943
1944         Parameters:
1945         -----
1946         data: numpy.ndarray of shape (M, 4)
1947             Data array with plasmon energy, left fwhm, right
1948             fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of
1949             third axis, respectively.
1950         name: str
1951             The name of the dataset
1952         temperature: float
1953             The temperature used in the experiment in K
1954         """
1955         self.data = data
1956         self.name = name
1957         self.temperature = temperature
1958         self._parameter_mapping = {'Ep': self.energy, 'fwhm1':
1959         self.fwhm1, 'fwhm2': self.fwhm2, 'integral': self.integral}
1960
1961         def __repr__(self):
1962             return f'{self.__class__.__name__}({self.data!r}, name={
1963             self.name!r}, temperature={self.temperature!r})'
1964
1965         def __str__(self):
1966             return f'{self.__class__.__name__} with shape {self.data
1967             .shape}:\n{self.dataframe.describe()}'
1968
1969         @property
1970         def energy(self):
1971             return self.data[:, 0]
1972
1973         @property
1974         def fwhm1(self):
1975             return self.data[:, 1]
1976
1977         @property
1978         def fwhm2(self):
1979             return self.data[:, 2]
1980
1981         @property
1982         def integral(self):
1983             return self.data[:, 3]

```

```

1981
1982 @property
1983 def dataframe(self):
1984     df = pd.DataFrame(self.data.reshape((self.data.shape[0],
1985     self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', '
1986     integral'])
1987     df.insert(0, 'T', self.temperature)
1988     return df
1989
1990 def get_parameter(self, parameter):
1991     """
1992     Return a parameter from the fit
1993     """
1994     return self._parameter_mapping.get(parameter, None)
1995
1996 def plot(self, kind, parameters, *args, **kwargs):
1997     """
1998     Plot the fit results
1999
2000     Parameters:
2001     -----
2002     kind: str
2003         The kind of plot. Should be either "img" or "hist"
2004     parameters: str or list
2005         The parameter(s) to plot.
2006     *args: Optional arguments passed to plotting functions
2007     **kwargs: Optional keyword arguments passed to plotting
2008     functions
2009
2010     Returns:
2011     -----
2012     Returns the figure and axes generated by the plotting
2013     functions
2014     """
2015     if parameters is None:
2016         parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
2017     else:
2018         if isinstance(parameters, str):
2019             parameters = [parameters]
2020         else:
2021             parameters = list(parameters)
2022
2023     if kind == 'img':
2024         fig, axes = plt.subplots(ncols=len(parameters),
2025         nrows=1, sharex=True, sharey=True)
2026         fig.suptitle(f'{self.name}')

```

```

2023         if len(parameters) == 1:
2024             axes = list([axes])
2025             kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
)
2026         for ax, parameter in zip(axes, parameters):
2027             ax.imshow(self.get_parameter(parameter), *args,
**kwargs)
2028             ax.set_title(f'{parameter}')
2029     elif kind == 'hist':
2030         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1)
2031         fig.suptitle(f'{self.name}')
2032         if len(parameters) == 1:
2033             axes = list([axes])
2034             df = self.dataframe
2035             for ax, parameter in zip(axes, parameters):
2036                 sns.histplot(df, x=parameter, ax=ax, **kwargs)
2037     else:
2038         return NotImplementedError(f'Kind {kind} is not
implemented')
2039     return fig, axes
2040
2041 class VoigtSegmented():
2042     """
2043     A class for storing Voigt fit parameters of a plasmon peak
after segmentation
2044     """
2045     def __init__(self, data, name=None, temperature=None):
2046         """
2047         Create a VoigtFit instance
2048
2049         Parameters:
2050         -----
2051         data: numpy.ndarray of shape (M, 4)
2052             Data array with plasmon energy, Gaussian fwhm,
Lorentzian fwhm and integral (area) in 1st, 2nd, 3rd and 4th
position of third axis, respectively.
2053         name: str
2054             The name of the dataset
2055         temperature: float
2056             The temperature used in the experiment in K
2057         """
2058         self.data = data
2059         self.name = name
2060         self.temperature = temperature
2061         self._parameter_mapping = {'Ep': self.energy, 'Gfwhm':
self.Gfwhm, 'Lfwhm': self.Lfwhm, 'integral': self.integral}

```

```

2062
2063     def __repr__(self):
2064         return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'
2065
2066     def __str__(self):
2067         return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'
2068
2069     @property
2070     def energy(self):
2071         return self.data[:, 0]
2072
2073     @property
2074     def Gfwhm(self):
2075         return self.data[:, 1]
2076
2077     @property
2078     def Lfwhm(self):
2079         return self.data[:, 2]
2080
2081     @property
2082     def integral(self):
2083         return self.data[:, 3]
2084
2085
2086
2087     @property
2088     def dataframe(self):
2089         df = pd.DataFrame(self.data.reshape((self.data.shape[0],
self.data.shape[1])), columns=['Ep', 'Gfwhm', 'Lfwhm', '
integral'])
2090         df.insert(0, 'T', self.temperature)
2091         return df
2092
2093     def get_parameter(self, parameter):
2094         """
2095         Return a parameter from the fit
2096         """
2097         return self._parameter_mapping.get(parameter, None)
2098
2099     def plot(self, kind, parameters, *args, **kwargs):
2100         """
2101         Plot the fit results
2102
2103         Parameters:
2104         -----

```

```

2105     kind: str
2106         The kind of plot. Should be either "img" or "hist"
2107     parameters: str or list
2108         The parameter(s) to plot.
2109     *args: Optional arguments passed to plotting functions
2110     **kwargs: Optional keyword arguments passed to plotting
functions
2111
2112     Returns:
2113     -----
2114     Returns the figure and axes generated by the plotting
functions
2115     """
2116     if parameters is None:
2117         parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']
2118     else:
2119         if isinstance(parameters, str):
2120             parameters = [parameters]
2121         else:
2122             parameters = list(parameters)
2123
2124
2125     if kind == 'img':
2126         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1, sharex=True, sharey=True)
2127         fig.suptitle(f'{self.name}')
2128         if len(parameters) == 1:
2129             axes = list([axes])
2130         kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
)
2131         for ax, parameter in zip(axes, parameters):
2132             ax.imshow(self.get_parameter(parameter), *args,
**kwargs)
2133             ax.set_title(f'{parameter}')
2134     elif kind == 'hist':
2135         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1)
2136         fig.suptitle(f'{self.name}')
2137         if len(parameters) == 1:
2138             axes = list([axes])
2139         df = self.dataframe
2140         for ax, parameter in zip(axes, parameters):
2141             sns.histplot(df, x=parameter, ax=ax, **kwargs)
2142     else:
2143         return NotImplementedError(f'Kind {kind} is not
implemented')
2144     return fig, axes

```

```

2145
2146
2147
2148 class SplitVoigtSegmented():
2149     """
2150     A class for storing Lorentz fit parameters of a plasmon peak
2151     after segmentation
2152     """
2153     def __init__(self, data, name=None, temperature=None):
2154         """
2155         Create a VoigtFit instance
2156
2157         Parameters:
2158         -----
2159         data: numpy.ndarray of shape (M, 4)
2160             Data array with plasmon energy, left fwhm, right
2161             fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of
2162             third axis, respectively.
2163         name: str
2164             The name of the dataset
2165         temperature: float
2166             The temperature used in the experiment in K
2167         """
2168         self.data = data
2169         self.name = name
2170         self.temperature = temperature
2171         self._parameter_mapping = {'Ep': self.energy, 'fwhm1':
self.fwhm1, 'fwhm2': self.fwhm2, 'eta' : self.eta, 'integral'
: self.integral}
2172
2173     def __repr__(self):
2174         return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'
2175
2176     def __str__(self):
2177         return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'
2178
2179     @property
2180     def energy(self):
2181         return self.data[:, 0]
2182
2183     @property
2184     def fwhm1(self):
2185         return self.data[:, 1]
2186
2187     @property

```

```

2185     def fwhm2(self):
2186         return self.data[:, 2]
2187
2188     @property
2189     def eta(self):
2190         return self.data[:, 3]
2191
2192     @property
2193     def integral(self):
2194         return self.data[:, 4]
2195
2196
2197
2198     @property
2199     def dataframe(self):
2200         df = pd.DataFrame(self.data.reshape((self.data.shape[0],
2201                                             self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', 'eta'
2202                                             , 'integral'])
2203         df.insert(0, 'T', self.temperature)
2204         return df
2205
2206     def get_parameter(self, parameter):
2207         """
2208         Return a parameter from the fit
2209         """
2210         return self._parameter_mapping.get(parameter, None)
2211
2212     def plot(self, kind, parameters, *args, **kwargs):
2213         """
2214         Plot the fit results
2215
2216         Parameters:
2217         -----
2218         kind: str
2219             The kind of plot. Should be either "img" or "hist"
2220         parameters: str or list
2221             The parameter(s) to plot.
2222         *args: Optional arguments passed to plotting functions
2223         **kwargs: Optional keyword arguments passed to plotting
2224         functions
2225
2226         Returns:
2227         -----
2228         Returns the figure and axes generated by the plotting
2229         functions
2230         """
2231         if parameters is None:

```



```

2228     parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
2229     else:
2230         if isinstance(parameters, str):
2231             parameters = [parameters]
2232         else:
2233             parameters = list(parameters)
2234
2235
2236     if kind == 'img':
2237         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1, sharex=True, sharey=True)
2238         fig.suptitle(f'{self.name}')
2239         if len(parameters) == 1:
2240             axes = list([axes])
2241         kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
)
2242         for ax, parameter in zip(axes, parameters):
2243             ax.imshow(self.get_parameter(parameter), *args,
**kwargs)
2244             ax.set_title(f'{parameter}')
2245     elif kind == 'hist':
2246         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1)
2247         fig.suptitle(f'{self.name}')
2248         if len(parameters) == 1:
2249             axes = list([axes])
2250         df = self.dataframe
2251         for ax, parameter in zip(axes, parameters):
2252             sns.histplot(df, x=parameter, ax=ax, **kwargs)
2253     else:
2254         return NotImplementedError(f'Kind {kind} is not
implemented')
2255     return fig, axes
2256
2257
2258 class FanoSegmented():
2259     """
2260     A class for storing Fano fit parameters of a plasmon peak
after segmentation
2261     """
2262     def __init__(self, data, name=None, temperature=None):
2263         """
2264         Create a VoigtFit instance
2265
2266         Parameters:
2267         -----
2268         data: numpy.ndarray of shape (M, 4)

```

```

2269         Data array with plasmon energy, fwhm, integral (area
        ), phi and background in 1st, 2nd, 3rd, 4th and 5th position
        of third axis, respectively.
2270         name: str
2271             The name of the dataset
2272         temperature: float
2273             The temperature used in the experiment in K
2274         """
2275         self.data = data
2276         self.name = name
2277         self.temperature = temperature
2278         self._parameter_mapping = {'Ep': self.energy, 'fwhm1':
self.fwhm1, 'phi': self.phi, 'integral': self.integral, '
background' : self.background}
2279
2280     def __repr__(self):
2281         return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'
2282
2283     def __str__(self):
2284         return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'
2285
2286     @property
2287     def energy(self):
2288         return self.data[:, 0]
2289
2290     @property
2291     def fwhm1(self):
2292         return self.data[:, 1]
2293
2294
2295     @property
2296     def integral(self):
2297         return self.data[:, 2]
2298
2299     @property
2300     def phi(self):
2301         return self.data[:, 3]
2302
2303
2304     @property
2305     def background(self):
2306         return self.data[:, 4]
2307
2308
2309

```

```

2310 @property
2311 def dataframe(self):
2312     df = pd.DataFrame(self.data.reshape((self.data.shape[0],
2313     self.data.shape[1])), columns=['Ep', 'fwhm1', 'integral', '
2314     phi', 'background'])
2315     df.insert(0, 'T', self.temperature)
2316     return df
2317
2318 def get_parameter(self, parameter):
2319     """
2320     Return a parameter from the fit
2321     """
2322     return self._parameter_mapping.get(parameter, None)
2323
2324 def plot(self, kind, parameters, *args, **kwargs):
2325     """
2326     Plot the fit results
2327
2328     Parameters:
2329     -----
2330     kind: str
2331         The kind of plot. Should be either "img" or "hist"
2332     parameters: str or list
2333         The parameter(s) to plot.
2334     *args: Optional arguments passed to plotting functions
2335     **kwargs: Optional keyword arguments passed to plotting
2336     functions
2337
2338     Returns:
2339     -----
2340     Returns the figure and axes generated by the plotting
2341     functions
2342     """
2343     if parameters is None:
2344         parameters = ['Ep', 'fwhm1', 'phi', 'integral', '
2345     background']
2346     else:
2347         if isinstance(parameters, str):
2348             parameters = [parameters]
2349         else:
2350             parameters = list(parameters)
2351
2352     if kind == 'img':
2353         fig, axes = plt.subplots(ncols=len(parameters),
2354         nrows=1, sharex=True, sharey=True)
2355         fig.suptitle(f'{self.name}')

```

```

2351         if len(parameters) == 1:
2352             axes = list([axes])
2353             kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
)
2354         for ax, parameter in zip(axes, parameters):
2355             ax.imshow(self.get_parameter(parameter), *args,
**kwargs)
2356             ax.set_title(f'{parameter}')
2357         elif kind == 'hist':
2358             fig, axes = plt.subplots(ncols=len(parameters),
nrows=1)
2359             fig.suptitle(f'{self.name}')
2360             if len(parameters) == 1:
2361                 axes = list([axes])
2362                 df = self.dataframe
2363                 for ax, parameter in zip(axes, parameters):
2364                     sns.histplot(df, x=parameter, ax=ax, **kwargs)
2365             else:
2366                 return NotImplementedError(f'Kind {kind} is not
implemented')
2367             return fig, axes
2368
2369
2370
2371
2372 class VolumePlasmon():
2373     """
2374     A class for storing Voigt fit parameters of a plasmon peak
after segmentation
2375     """
2376     def __init__(self, data, name=None, temperature=None):
2377         """
2378         Create a Volume Plasmon instance
2379
2380         Parameters:
2381         -----
2382         data: numpy.ndarray of shape (M, 4)
2383             Data array with plasmon energy, fwhm, and integral (
area) in 1st, 2nd and 3rd position of third axis,
respectively.
2384         name: str
2385             The name of the dataset
2386         temperature: float
2387             The temperature used in the experiment in C
2388         """
2389         self.data = data
2390         self.name = name

```

```

2391         self.temperature = temperature
2392         self._parameter_mapping = {'Ep': self.energy, 'fwhm':
self.fwhm, 'integral': self.integral}
2393
2394     def __repr__(self):
2395         return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'
2396
2397     def __str__(self):
2398         return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'
2399
2400     @property
2401     def energy(self):
2402         return self.data[:, 0]
2403
2404     @property
2405     def fwhm(self):
2406         return self.data[:, 1]
2407
2408     @property
2409     def integral(self):
2410         return self.data[:, 2]
2411
2412
2413
2414     @property
2415     def dataframe(self):
2416         df = pd.DataFrame(self.data.reshape((self.data.shape[0],
self.data.shape[1])), columns=['Ep', 'fwhm', 'integral'])
2417         df.insert(0, 'T', self.temperature)
2418         return df
2419
2420     def get_parameter(self, parameter):
2421         """
2422         Return a parameter from the fit
2423         """
2424         return self._parameter_mapping.get(parameter, None)
2425
2426     def plot(self, kind, parameters, *args, **kwargs):
2427         """
2428         Plot the fit results
2429
2430         Parameters:
2431         -----
2432         kind: str
2433             The kind of plot. Should be either "img" or "hist"

```

```

2434     parameters: str or list
2435         The parameter(s) to plot.
2436     *args: Optional arguments passed to plotting functions
2437     **kwargs: Optional keyword arguments passed to plotting
functions
2438
2439     Returns:
2440     -----
2441     Returns the figure and axes generated by the plotting
functions
2442     """
2443     if parameters is None:
2444         parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']
2445     else:
2446         if isinstance(parameters, str):
2447             parameters = [parameters]
2448         else:
2449             parameters = list(parameters)
2450
2451     if kind == 'img':
2452         fig, axes = plt.subplots(ncols=len(parameters),
2453 nrows=1, sharex=True, sharey=True)
2454         fig.suptitle(f'{self.name}')
2455         if len(parameters) == 1:
2456             axes = list([axes])
2457         kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01)
)
2458         for ax, parameter in zip(axes, parameters):
2459             ax.imshow(self.get_parameter(parameter), *args,
**kwargs)
2460             ax.set_title(f'{parameter}')
2461     elif kind == 'hist':
2462         fig, axes = plt.subplots(ncols=len(parameters),
nrows=1)
2463         fig.suptitle(f'{self.name}')
2464         if len(parameters) == 1:
2465             axes = list([axes])
2466         df = self.dataframe
2467         for ax, parameter in zip(axes, parameters):
2468             sns.histplot(df, x=parameter, ax=ax, **kwargs)
2469     else:
2470         return NotImplementedError(f'Kind {kind} is not
implemented')
2471     return fig, axes
2472
2473

```

```

2474 ##### A .ipynb file used to compile and run the
      algorithm
2475 %matplotlib qt
2476 import numpy as np
2477 import sys
2478 sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
      - 2023 V r\\Master\\Data handling\\fitting')
2479 import methods
2480 # import voigt_methods
2481 import splitLorentz_methods
2482 import pickle
2483 import pandas as pd
2484 from Model import e_0, hbar
2485 import experimental
2486 import plot_class
2487 import summation_and_deconvolve
2488 import analyse_results
2489 import seaborn as sns
2490 import matplotlib.pyplot as plt
2491 from functools import cache
2492 from tqdm import tqdm
2493 import MyFunc
2494 from scipy.stats import linregress
2495 %load_ext autoreload
2496 %autoreload 2
2497
2498 ##### New cell
2499 @cache
      # Convenient, but sort of
      hardcoded. The loading can and perhaps should have been dealt
      with in a more proper way, but it worked as a quick work
      around
2500 def load_data(temperatures, mask_num, approved_files):
      # overriding the function so that
      it can be rerun multiple times, requires input as tuple
2501     return summation_and_deconvolve.load_data(list(temperatures)
      ,mask_num, list(approved_files))
2502
2503 ##### New cell
2504 # example
2505 import splitLorentz_methods
2506 import lorentzian_methods
2507 import volumeplasmon_methods
2508 import fano_methods
2509 # all_methods = splitLorentz_methods.all_methods[:12]
2510
2511 # all_methods = voigt_methods.all_methods

```

```

2512 # all_methods = volumeplasmon_methods.all_methods
2513 # all_methods = [all_methods[0]]
2514 # all_methods = [splitLorentz_methods.method1000]
2515
2516 # all_methods = [splitLorentz_methods.methodpreSplitLorentz]
2517 # all_methods = [splitLorentz_methods.methodpreFano      ]
2518 # all_methods = [splitLorentz_methods.methodpreLorentz   ]
2519 # all_methods = [splitLorentz_methods.methodpreVolume    ]
2520 # all_methods = [splitLorentz_methods.method1000,
                splitLorentz_methods.method1001]
2521
2522 all_methods = splitLorentz_methods.tenseg[-4:]
2523 all_methods[0].compile()
2524
2525 ##### New cell      - where the algorithm is ran after a
                pixel-wise scan
2526
2527 delta = 0 # To be removed
2528 narrow_linregress_interval = False
2529 for method in all_methods:
2530     print(f'Starting method {method.name}')
2531     exclude_scans = method.part2['exclude_scans']
2532     _, approved_files = analyse_results.get_numpy_arrays(**method
                .part1)
2533
2534     pre_segmentation_data = {}
2535     _[23] = _['RM']
2536     for T in _.keys():
2537         if T=='RM' :
2538             _[23] = _[T]
2539             continue
2540             l1 = np.array([i[0] for i in approved_files])
2541             l2 = np.array([i[1] for i in approved_files])
2542             pre_segmentation_data[int(T)] = _[T]
2543
2544             pre_segmentation_data[int(T)] = _[T]
2545             # arrays=arrays_
2546             segments, mean_value, sigma_value, mask_num =
                summation_and_deconvolve.segment(pre_segmentation_data,
                pre_plasmon = method.part1['components'][0],
                num_params_pre_segment=method.part1['num_params_pre_segment'
                ], **method.part2)
2547             temperatures = tuple(method.part2['temperatures'])
2548             experimental_data, offset, scale = load_data(temperatures,
                mask_num, tuple(approved_files))
2549             mask_num = min(mask_num, 4)
2550             # assert False

```



```

2551     ##### Skipping linear regression for pre-segmentation data
#####
2552     ##### Pre-segmentation #####
2553     d = experimental_data
2554     val = np.array(pre_segmentation_data[23]).swapaxes(0,1).
swapaxes(1,2)[:,:,:mask_num]
2555
2556
2557     sh = (val.shape[0],val.shape[1],val.shape[2])
2558     x = 10 + np.arange(d[23].shape[-1])*0.010
        #Hardcoded
2559
2560     ##### Running for
premethods
2561     if method.name[:3] == 'pre':
2562         for T in tqdm(experimental_data.keys()):
2563             linreg = np.zeros(shape=sh, dtype=object)
2564             val = np.array(pre_segmentation_data[T]).swapaxes
(0,1).swapaxes(1,2)[:,:,:mask_num]
2565             for i in range(sh[0]):
2566                 for j in range(sh[1]):
2567                     for k in range(sh[2]):
2568                         if val[i,j,k,0] == 0:
2569                             continue
2570                             # linreg[i,j,k] = linregress(d[T][i,j,k
],MyFunc.model_function(x, *val[i,j,k])).rvalue**2
2571                             if narrow_linregress_interval:
2572                                 raise NotImplementedError('Are you
sure you dont want to change this?')
2573                                 delta = 1
2574                                 c = np.nanmean(val[:,:,:,:0],where
= (val[:,:,:,:0] > 0))
2575                                 I = ((c-delta) < x) & (x < (c+delta)
)
2576                                 linreg[i,j,k] = linregress(d[T][i,j,
k][I],MyFunc.model_function(x[I], *val[i,j,k])).rvalue**2
2577                                 else:
2578                                 curve_fitted_interval = (method.
part2['energy_interval'][0] < x) & (x < method.part2['
energy_interval'][1])
2579                                 linreg[i,j,k] = linregress(d[T][i,j,
k][curve_fitted_interval],MyFunc.model_function(x[
curve_fitted_interval], *val[i,j,k])).rvalue**2
2580
2581                                 np.save(f'linregress_result\{method.name}
_rvalueSquared_{T}'+narrow_linregress_interval*('narrowed'+
str(delta)),linreg, allow_pickle=True )

```

```

2582         continue
2583
2584
2585 ##### Filtering out empty segments
2586 #####
2587     labels = {}
2588     for T in temperatures:
2589         labels[T] = []
2590         for i in range(segments[T].max()+1):
2591             if (segments[T]==i).sum()!=0:
2592                 labels[T].append(i)
2593             else:
2594                 print(T,i)
2595     x = offset + np.arange(experimental_data[method.part2['
2596 temperatures']][0]).shape[-1])*scale
2597     new_fit = summation_and_deconvolve.sum_and_deconvolve(
2598 experimental_data,segments, pre_segmentation_data,mask_num,
2599 labels, mean_value,x, array=pre_segmentation_data, **method.
2600 part2) # Dont really know why I take in the array twice???
2601 ##### Running curve fit #####
2602     popt,pcov = summation_and_deconvolve.fit_curves(new_fit,x,
2603 mean_value,labels,**method.part2) # inclusion of only
2604 eneries within energy interval taken care of in fit_curves()
2605
2606 ##### Linear regression analysis #####
2607     d = new_fit
2608     val = np.array(new_fit[23])
2609
2610     # sh = (val.shape[0],val.shape[1],val.shape[2])
2611     x = 10 + np.arange(np.array(d[23]).shape[-1])*0.010
2612         #Hardcoded
2613
2614     for T in tqdm(experimental_data.keys()):
2615         linreg = np.zeros(shape=np.array(new_fit[T]).shape[0],
2616 dtype=object)
2617         val = np.array(new_fit[T])
2618         for i in range(1,len(new_fit[T])):
2619
2620             # Skipping the
2621             0th
2622             en_interval = (method.part2['energy_interval'][0] <
2623 x) & (x < method.part2['energy_interval'][1])
2624             linreg[i] = linregress(val[i][en_interval] ,MyFunc.
2625 model_function(x[en_interval], *popt[T][i])).rvalue**2
2626             # raise RuntimeError('Husk p bytte navn!')
2627             np.save(f'linregress_result\{method.name}_rvalueSquared
2628 {T}'+(exclude_scans)*'_excluded_scans',linreg, allow_pickle=
2629 True )

```

```

2614
2615 ##### Visualize a plot #####
2616     # plt.figure()
2617     # plt.plot(x,new_fit[200][3])
2618     # plt.plot(x,      MyFunc.model_function(x,*popt[200][3]))
2619     ##### Converting the results to a DataFrame to make
plotting easier #####
2620     temp = np.array
([23, 80, 120, 160, 200, 240, 280, 320, 360, 400, 450, 500])
2621     class_type = method.part2['class_type']
2622     df = pd.DataFrame()
2623     df_cov = pd.DataFrame()
2624     data={}
2625     data_cov = {}
2626     for T in temp:
2627         covariance = np.diagonal(np.array(pcov[T]), axis1=1,
axis2=2)
2628         if class_type == 'Fano':
2629             data[T] = plot_class.FanoSegmented(np.array(popt[T])
, str(T), int(T))
2630             data_cov[T] = plot_class.FanoSegmented(covariance,
str(T), int(T))
2631         if class_type == 'SplitVoigt':
2632             data[T] = plot_class.SplitVoigtSegmented(np.array(
popt[T]), str(T), int(T))
2633             data_cov[T] = plot_class.SplitVoigtSegmented(
covariance, str(T), int(T))
2634         if class_type == 'DerivativeLorentzian' or class_type == '
SplitLorentzian':
2635             data[T] = plot_class.SplitLorentzSegmented(np.array(
popt[T]), str(T), int(T))
2636             data_cov[T] = plot_class.SplitLorentzSegmented(
covariance, str(T), int(T))
2637         if class_type == 'Voigt': #
NOTE: Integral is not really integral as of now
2638             data[T] = plot_class.VoigtSegmented(np.array(popt[T]
)], str(T), int(T))
2639             data_cov[T] = plot_class.VoigtSegmented(covariance,
str(T), int(T))
2640         if class_type == 'VolumePlasmon' or class_type == '
Lorentzian':
2641             data[T] = plot_class.VolumePlasmon(np.array(popt[T])
, str(T), int(T))
2642             data_cov[T] = plot_class.VolumePlasmon(covariance,
str(T), int(T))
2643
2644

```

```

2645     df      = df.append(data[T].dataframe, ignore_index=True)
2646     df_cov  = df_cov.append(data_cov[T].dataframe,
2647     ignore_index=True)
2648     # unfiltered_df = df.copy()
2649
2650     ##### Filtering end-result #####
2651     param_list = ['Ep', 'fwhm1', 'fwhm2'] # Split
Lorentz
2652     if class_type=='Fano':
2653         param_list = ['Ep', 'fwhm1']
2654     if class_type == 'Voigt':
2655         param_list = ['Ep', 'Lfwhm']
2656     if class_type == 'VolumePlasmon' or class_type == '
Lorentzian':
2657         param_list = ['Ep', 'fwhm']
2658     for param in param_list:
2659         for T in temp:
2660             q_low = df[(df['T'] == int(T))][param].quantile
(0.1)
2661             q_high = df[(df['T'] == int(T))][param].quantile
(0.9)
2662             df_cov = df_cov[((df[param]>= q_low) & (df[param]
<= q_high)) | (df['T']!=int(T)) ] # Has to be
before the next line
2663             df      = df[((df[param]>= q_low) & (df[param] <=
q_high)) | (df['T']!=int(T)) ]
2664             # df_cov = df_cov[((df[param]>= q_low) & (df[param]
] <= q_high)) | (df['T']!=int(T)) ]
2665             # zlp = zlp[((df[param]>= q_low) & (df[param] <=
q_high)) | (df['T']!=int(T)) ]
2666             if (class_type=='SplitLorentzian' ) or (class_type=='
DerivativeLorentzian' ):
2667                 df['FWHM'] = (df['fwhm1'] + df['fwhm2'])/2 # add
additional column for mean FWHM # for Lorentzian split
2668             elif class_type == 'Fano':
2669                 df['FWHM'] = df['fwhm1']
# for Fano
2670
2671             elif class_type == 'Voigt':
2672                 df['FWHM'] = df['Lfwhm']
2673             elif class_type == 'VolumePlasmon' or class_type == '
Lorentzian':
2674                 df['FWHM'] = df['fwhm']
2675             else:
2676                 raise NotImplementedError
2677             df['Resistivity'] = (e_0/hbar*df['Ep']**2/df['FWHM'])**-1

```

```

2677     df.to_pickle('dataframes_results\\'+method.name+'
excluded_scans'*exclude_scans) #
    save the method for future reference
2678     df_cov.to_pickle('dataframes_results\\'+method.name+'
excluded_scans'*exclude_scans+'_cov')
        # save the method for future reference
2679     f = open(f'dataframes_results\\'+ method.name + '
excluded_scans'*exclude_scans+'_part1.pkl', 'wb')
2680     pickle.dump(method.part1,f)
2681     f.close()
2682     f = open(f'dataframes_results\\'+ method.name + '
excluded_scans'*exclude_scans+'_part2.pkl', 'wb')
2683     pickle.dump(method.part2,f)
2684     f.close()
2685     ##### Plot resistivity #####
2686     # plt.figure()
2687     # sns.scatterplot(df, x='T', y='Resistivity')
2688
2689
2690     """ summation_and_deconvolve.py """
2691     import experimental
2692     from tqdm import tqdm, trange
2693     import numpy as np
2694     import matplotlib.pyplot as plt
2695     import logging
2696
2697     import sys
2698     sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
- 2023 V r\\Master\\Data handling\\fitting')
2699     import signal_lines
2700     import Params
2701     import Model
2702     # import MyFuncscopy
2703     from skimage.segmentation import slic
2704     import utils
2705
2706
2707     lowest_ev = 10.
2708     temperatures = ['80', '120', '160', '200', '240', '280', '320', '360', '
400', '450', '500']
2709
2710     def segment(arrays, pre_plasmon, num_params_pre_segment, **
kwargs):
2711         """This function takes in the results and segments them.
Class type is the class used pre-segmentation.
2712

```

```

2713     num_params_pre_segment is the number of parameters used for
the curve fitting prior to segmentation.
2714     """
2715     class_type = str(type(pre_plasmon))[21:-2]
2716     ##### Getting rid of outliers
#####
2717     if 'temperatures' in kwargs.keys():
2718         temp = kwargs['temperatures']
2719     else:
2720         temp = ['80', '120', '160', '200', '240', '280', '320', '360', '
400', '450', '500']
2721     if 'filter_outliers_max' in kwargs.keys():
2722         filter_outliers_max = kwargs['filter_outliers_max']
2723     else:
2724         filter_outliers_max = None
2725     if 'filter_outliers_min' in kwargs.keys():
2726         filter_outliers_min = kwargs['filter_outliers_min']
2727     else:
2728         filter_outliers_min = None
2729     if 'additional_constraints' in kwargs.keys():
2730         additional_constraints = kwargs['additional_constraints']
]
2731     else:
2732         additional_constraints = False
2733     if 'pixels_per_segment' in kwargs.keys():
2734         pixels_per_segment = kwargs['pixels_per_segment']
2735     else:
2736         pixels_per_segment = 20
2737     if 'num_param' in kwargs.keys():
2738         num_param = kwargs['num_param']
2739     else:
2740         num_param = 3 # number of
parameters to be included. Ended up with 1 for the thesis (i
.e. only center) as this provided the best results
2741     if 'compactness' in kwargs.keys():
2742         compactness = kwargs['compactness']
2743     else:
2744         compactness = 1
2745
2746     mask__ = arrays[temp[0]][0][:,:,0] > 10 #
To be removed
2747     mask_num = np.array([len(arrays[T]) for T in temp]).min()
#TODO: make the code susceptible for ragged
arrays
2748
2749     segments = {}
2750

```

```

2751     n_sigma = kwargs['n_sigma']
           # sigma value for including/excluding pixels.
2752                                     # all pixels included in the
analysis. Should be same for all known values when using
SplitVoigt
2753
2754     mask = np.zeros(shape=(mask__.shape[0],mask__.shape[1],
mask_num), dtype=bool)           # img_sizeX, img_sizeY, number of
scans
2755     mean_value = {}
2756     sigma_value = {}
2757     _ = {}
2758     for k,T in enumerate(temp):
2759         mask = np.zeros(shape=(mask__.shape[0],mask__.shape
[1],len(arrays[T])), dtype=bool)           # img_sizeX, img_sizeY,
number of scans
2760         # print(mask_num)
2761         _[T] = np.array(arrays[T][:]).swapaxes(0,1).swapaxes
(1,2)
2762         # for i in range(mask_num):
2763         for i in range(len(arrays[T])):
2764             if (int(T), i) in utils.exclude and kwargs['
exclude_scans']:
2765                 mask[:, :, i] = False
2766                 print((T,i), 'excluded')
2767                 continue
2768             if additional_constraints:
2769                 if class_type == 'SplitLorentzian':
2770                     mask_ = (arrays[T][i][:,:,0] > 13 ) & (
arrays[T][i][:,:,1] > 0.3) & (arrays[T][i][:,:,2] > 0.3) & (
arrays[T][i][:,:,1] < 0.8) & (arrays[T][i][:,:,2] < 1.2)
2771                 elif class_type == 'Fano':
2772                     mask_ = (arrays[T][i][:,:,0] > 13 ) #& (
arrays[T][i][:,:,1] > 0.3) & (arrays[T][i][:,:,1] < 0.8)
2773                 elif class_type == 'Lorentzian':
2774                     mask_ = (arrays[T][i][:,:,0] > 13 ) & (
arrays[T][i][:,:,1] > 0.1)
2775                 elif class_type == 'VolumePlasmon':
2776                     mask_ = (arrays[T][i][:,:,0] > 13 ) & (
arrays[T][i][:,:,1] > 0.1)
2777
2778
2779             else:
2780                 mask_ = arrays[T][i][:,:,0] > 10
2781                 mask[:, :, i] = mask_==1
2782                 for param in range(num_params_pre_segment):
2783                     arr = arrays[T][i][:,:,param]

```

```

2784         arr_ = arr[mask_]
2785         if (filter_outliers_min is not None) and (
filter_outliers_min is not None):
2786             mean_value[(T,i,param)] = np.mean(np.sort(
arr_)[filter_outliers_min:-filter_outliers_max]) # excluding
extreme outliers
2787             sigma_value[(T,i,param)] = np.std( np.sort(
arr_)[filter_outliers_min:-filter_outliers_max]) # excluding
extreme outliers
2788         else:
2789             mean_value[(T,i,param)] = np.nanmean(arr,
where=mask_)
2790             sigma_value[(T,i,param)] = np.nanstd( arr,
where=mask_)
2791             mask[:, :, i] = mask[:, :, i] & (np.abs
((arr - mean_value[(T,i,param)])) < n_sigma*sigma_value[(T,
i,param)] )
2792             valid_num_pixels = mask.sum()
2793             n_segements = int(valid_num_pixels/
pixels_per_segment)
2794             segments[T] = slice(_[T][:, :, :, :num_param], n_segements=
n_segements, compactness=compactness, mask=mask,
enforce_connectivity=kwargs['enforce_connectivity'],
min_size_factor=0.9)#, max_size_factor=1.4)
2795             print(T, f'C. valid_num_pixels={valid_num_pixels},
n_segements={n_segements}; Result: n_segements={segments[T].max
()}, pixels_per_segment={valid_num_pixels/segments[T].max()}
')
2796             return segments, mean_value, sigma_value, mask_num
2797
2798
2799
2800 def load_data(temp, mask_num, approved_files):
2801     """Mask_num is to be removed"""
2802     import hyperspy.api as hs
2803     # mask_num = min(mask_num, 4)
2804     T = temp
2805     last_index = 1000
2806     experimental_data = {}
2807     for T in tqdm(temp):
2808         if (T == 23) or (T=='RM'):
2809             lowloss = []
2810             highloss = []
2811
2812             for i in range(1,12):
2813                 # print(f'Checking {(T,i)}')
2814                 if ('RM',i) not in approved_files:

```



```

2815         logging.warning(f'Skipping{(T,i)}')
2816         continue
2817         lowloss.append(hs.load(r'Cambridge_7\S1_RMTemp _
0.01s_58nmpixelSTEM SI'+str(i)+'.dm4')[-2]) #
special note! The scans before were not good enough
2818         highloss.append(hs.load(r'Cambridge_7\S1_RMTemp
_ 0.01s_58nmpixelSTEM SI'+str(i)+'.dm4')[-1]) #
special note! The scans before were not good enough
2819         # logging.warning('Note to self: tar kun med
scannene fom scan 6 for romtemperatur, mulig det blir ragged
da')
2820     else:
2821         lowloss = []
2822         highloss= []
2823         for i in range(1,12):
2824             if (str(T),i) not in approved_files:
2825                 logging.warning(f'Skipping{(T,i)}')
2826                 continue
2827                 lowloss .append(hs.load(r'Cambridge_7\S1_'+str(T)
)+'CTemp _ 0.01s_58nmpixel'+str(i)+'.dm4')[-2])# for i in
range(1,mask_num+1)]
2828                 highloss.append(hs.load(r'Cambridge_7\S1_'+str(T)
)+'CTemp _ 0.01s_58nmpixel'+str(i)+'.dm4')[-1])# for i in
range(1,mask_num+1)]
2829                 for l,h in zip(lowloss,highloss):
2830                     l.align_zero_loss_peak(also_align=[h],
show_progressbar=False, print_stats=False, signal_range
=(-1.,1.))
2831                     print(T, (highloss))
2832                     scale = h.axes_manager['Energy loss'].scale
2833                     experimental_data[T] = np.array([h.isig[lowest_ev:].isig
[:last_index].data for h in highloss]).swapaxes(0,1).
swapaxes(1,2)
2834                     # logging.debug(T+ [h.isig[lowest_ev:].axes_manager['
Energy loss'].offset for h in highloss])
2835                     print(T, [h.isig[lowest_ev:].axes_manager['Energy loss'].
offset for h in highloss])
2836                     print(T, [h.isig[lowest_ev:].axes_manager['Energy loss'].
scale for h in highloss])
2837                     return experimental_data, lowest_ev, scale
2838
2839
2840 def sum_and_deconvolve(experimental_data, segments, arrays,
mask_num, labels, mean_value, x, array=None, class_type='
SplitVoigt', **kwargs):
2841     ##### Summing the pixels and deconvolving
#####

```

```

2842 from scipy.optimize import curve_fit
2843 import MyFunc
2844 import utils
2845 import hyperspy.api as hs
2846 if 'temperatures' in kwargs.keys():
2847     temperatures = kwargs['temperatures']
2848 if 'subtract_center' in kwargs.keys():
2849     subtract_center = kwargs['subtract_center']
2850     assert array is not None
2851 else:
2852     subtract_center = False
2853 known_params = {}
2854 offset = x[0]
2855 scale = (x[-1]-x[0])/x.shape[0]
2856 # class_type = str(type(plasmon))[21:-2]
2857 if class_type=='SplitVoigt':
2858     known_params = {'center': 0,
2859                    'fwhm1' : 1,
2860                    'fwhm2' : 2,
2861                    'frac'  : 3,
2862                    'area'  : 4}
2863 if class_type=='Fano':
2864     known_params = {'center': 0,
2865                    'fwhm' : 1,
2866                    'area' : 2,
2867                    'phi'  : 3,
2868                    'background' : 4}
2869 if 'FWHM' in kwargs.keys():
2870     FWHM = kwargs['FWHM'] #
eV
2871 else:
2872     FWHM = 0.08
2873 deconv_func = signal_lines.gaussian((np.arange(x.shape[0])-x
2874 .shape[0]/2)*scale,0,FWHM,1) #Keep it centered
2875 modifier = kwargs['modify']
2876 new_fit = {}
2877 for T in temperatures:
2878     # print(T)
2879     if T==23:
2880         zlp = hs.load(r'Cambridge_7\S1_RMTemp _ 0.01
2881 s_58nmpixelSTEM SI'+str(1)+' .dm4')[2]
2882     else:
2883         zlp = hs.load(r'Cambridge_7\S1_'+str(T)+'CTemp _
2884 0.01s_58nmpixel'+str(1)+' .dm4')[2]
2885     zlp = np.roll(zlp.data[0,0,:x.shape[0]],300)
2886     # centerig the ZLP prior to
deconvolution, but hardcoded, taken from vacuum

```

```

2883     new_fit[T] = []
2884     arr = np.array(arrays[int(T)]).swapaxes(0,1).swapaxes
(1,2)[:,:,:mask_num]
2885     for i in range(0,segments[int(T)].max()+1):
2886         if subtract_center:
2887             # Not used for end-results
2888             y = []
2889             for j in range(experimental_data[T].shape[0]):
2890                 for k in range(experimental_data[T].shape
[1]):
2891                     for l in range(experimental_data[T].
shape[2]):
2892                         if segments[T][j,k,l] != i:
2893                             continue
2894                         center = np.array(array[T]).swapaxes
(0,1).swapaxes(1,2)[j,k,l,0].copy() # the center has
index 0. TODO: this is not very convenient programming
2895                         y.append(np.roll(experimental_data[T
][j,k,l], -int(np.round(center/scale)+500))) ## Adding same
value for all pixels. Not supposed to find the center now
anyway
2896
2897                         y = np.array(y).mean(axis=0)
2898
2899                     else:
2900                         y = experimental_data[T][segments[T]==i].mean(
axis=0).copy() # Take the mean over the
pixels
2901                         y -= np.median(np.roll(y,200)[200:2*200])
#
2902                         # Hardcoded. Median is taken for 10<E<12
2903                         if modifier == 'fourier_ratio':
2904                             # deconvolve the summed regions
2905                             s = np.fft.ifft(utils.fourier_ratio(y,zlp,
deconv_func))
2906                         elif modifier == 'zero_loss':
2907                             y += zlp
2908                             s = np.fft.ifft(np.fft.fft(zlp)
*np.log(
np.fft.fft(y)/np.fft.fft(zlp)))
2909                         elif modifier == 'gaussian':
2910                             y += zlp
2911                             s = np.fft.ifft(np.fft.fft(deconv_func)*np.log(
np.fft.fft(y)/np.fft.fft(zlp)))
2912                         else:
2913                             s = y
2914                             new_fit[T].append(s.real)
2915     return new_fit

```

```

2913
2914
2915 def fit_curves(new_fit,x_, mean_value,labels, class_type='
SplitVoigt',**kwargs):
2916     ##### running the curve fit again #####
2917     from scipy.optimize import curve_fit
2918     import MyFunc
2919     from tqdm import tqdm
2920     if class_type=='SplitVoigt':
2921         known_params = {'center': 0,
2922                         'fwhm1' : 1,
2923                         'fwhm2' : 2,
2924                         'frac'   : 3,
2925                         'area'   : 4}
2926         bounds       = ([13.8,0,0,0,0],[16,3,3,1,1e5])
2927         init_guess   = [15,0.5,0.5,1,1e3]
2928     if class_type=='Fano':
2929         known_params = {'center': 0,
2930                         'fwhm'  : 1,
2931                         'area'   : 2,
2932                         'phi'    : 3,
2933                         'background' : 4}
2934         bounds       = ([14,0,0,-2*np.pi,0],[16,3,1e6,2*np.pi,1])
2935         init_guess   = [15,0.2,2e2,1,0.5]
2936
2937     if (class_type=='SplitLorentzian') or (class_type=='
DerivativeLorentzian'):
2938         known_params = {'center': 0,
2939                         'fwhm1' : 1,
2940                         'fwhm2' : 2,
2941                         'area'   : 3}
2942         bounds       = ([13.8,0,0,0],[16,3,3,1e5])
2943         init_guess   = [15,0.5,0.5,1e3]
2944         if class_type=='DerivativeLorentzian':
2945             init_guess = [15,0.5,0.5,1e1]
2946     if class_type=='Voigt':
2947         known_params = {'center': 0,
2948                         'Gfwhm' : 1,
2949                         'Lwhm'  : 2,
2950                         'area'   : 3}
2951         bounds       = ([13.8,0,0,0],[16,3,3,1e5])
2952         init_guess   = [15,0.1,0.5,1e3]
2953     if (class_type=='VolumePlasmon') or (class_type=='Lorentzian
'):
2954         known_params = {'center': 0,
2955                         'fwhm'  : 1,
2956                         'area'   : 2}

```

```

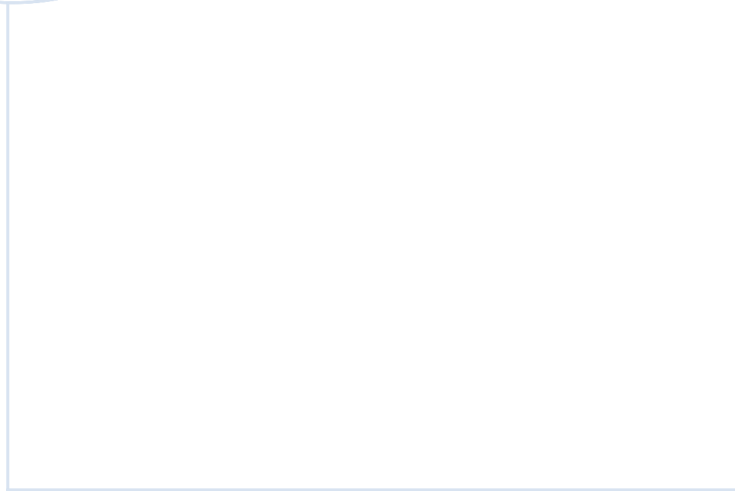
2957     bounds      = ([13.8,0,0],[16,3,1e5])
2958     init_guess  = [15,0.5,1e3]
2959
2960     weight = kwargs['weighted']
2961     popt = {}
2962     pcov = {}
2963     for T in tqdm(new_fit.keys()):
2964         popt[T] = []
2965         pcov[T] = []
2966         for i,y_ in enumerate(new_fit[T]):
2967
2968             try:
2969                 if i not in labels[T] or i ==0:
2970                     # dont include the 0th as it is all
the pixels that were excluded from the scans
2971                     raise RuntimeError
2972
2973                 if kwargs['num_bin'] != -1:
2974                     y = np.histogram(x_,weights=y_, bins =
kwargs['num_bin'])[0]
2975                     x = np.histogram(x_,weights=y_, bins =
kwargs['num_bin'])[1][: -1]
2976
2977                 else:
2978                     y = y_.copy()
2979                     x = x_.copy()
2980
2981                 en_interval = (kwargs['energy_interval'][0] < x)
& (x < kwargs['energy_interval'][1])
2982                 x = x[en_interval]
2983
2984                                     # Not considering
what is outside of the pre-defined energy interval
2985                 y = y[en_interval]
2986
2987                                     # Not considering
what is outside of the pre-defined energy interval
2988                 if True:
2989                     # print(kwargs['derivative'])
2990                     if kwargs['derivative'] or class_type=='
DerivativeLorentzian': # TODO: fix all weighting
specifications
2991
2992                         if class_type != 'DerivativeLorentzian':
2993                             raise NotImplementedError
2994                         y = utils.derivative(y,x)
2995                         if weight:
2996                             logging.warning('Taking the weighted
fit of a derivative.')
2997                         sigma = 1/y**2*weight+1*(not weight)

```

```

2992         popt_, pcov_ = curve_fit(MyFunc.
model_function, x,y, p0=init_guess, bounds=bounds, sigma=
sigma, absolute_sigma=True)
2993         if kwargs['temp_adjusted_interval']:
2994             if class_type != 'SplitLorentzian':
2995                 raise NotImplementedError('Only
available for SplitLorentzian')
2996         en_interval = ((popt_[0]-0.5)< x) & (x <
popt_[0])
2997         bounds_      = ([14.4,0,0,0], [popt_[0],1,1,1
e5])
2998         init_guess_ = [popt_[0],0.1,0.1,1e3]
2999         x = x[en_interval]
3000         y = y[en_interval]
3001         popt_, pcov_ = curve_fit(MyFunc.
model_function, x,y, p0=init_guess_, bounds=bounds_, sigma=
sigma[en_interval], absolute_sigma=True)
3002         popt[T].append(popt_)
3003         pcov[T].append(pcov_)
3004         except RuntimeError:
3005             popt[T].append(np.array([np.nan]*len(
known_params.keys())))
3006             pcov[T].append(np.ones(    (len(known_params.keys
()),len(known_params.keys()))    ) *np.nan)
3007             print(T,i,'Runtimeerror, adding nans')
3008             continue
3009         return (popt, pcov)

```



 **NTNU**

Norwegian University of
Science and Technology