Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Physics

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

NTNU
Norwegian University of
Science and Technology

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

NTNU
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℃ have been investigated with an energy resolution of $\lesssim 100\,\mathrm{meV}$. The plasmon energy was found to be $\approx 15\,\mathrm{eV}$ at room temperature, decreasing with approximately $0.5\,\mathrm{meV\,K^{-1}}$ and had an asymmetric linewidth of $\approx 500\,\mathrm{meV}$ and $700\,\mathrm{meV}$ 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 energioppløsning 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 konkluderene 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 plasmonspredning 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 plasmonspredning 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 transmissjonselektronmikroskopskolonnen 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 (wwwwiiiiiiiiiiiiiiiii) og NTNUI Samba (Samba-toget ruller videre!).
Jeg er glad i dere < 3

[illegible]

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	$37.7 \cdot 10^6\ \mathrm{S\,m^{-1}}$
Temperature coefficient	α	0.0053
Coefficient of linear expansion	β	$23 \cdot 10^{-6}\ \mathrm{K^{-1}}$
Electron density	n	$18.1{\cdot}10^{22}\ \mathrm{cm^{-1}}$

Contents

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 todays 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 form 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 literautre, 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

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

where the external force has been taken to originate from an external electrical field $\boldsymbol{\mathcal{E}}$, 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

$$\mathbf{j} = -ne\langle \mathbf{v} \rangle = \sigma \boldsymbol{\mathcal{E}} \tag{2.3}$$

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

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

where σ_0 is defined through the relationship

$$\sigma_0 = \frac{ne^2\tau}{m}, \tag{2.5}$$

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 $\epsilon_r(\mathbf{k} \to 0, \omega)$ 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,

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

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

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

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 $\mathfrak{Im}(-1/\epsilon(k, E))$ 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

$$\nabla \cdot \mathbf{D} = \rho_{\text{ext}} \qquad ; \qquad \nabla \cdot \boldsymbol{\mathcal{E}} = \rho_{\text{tot}}/\epsilon_0 \tag{2.8a,b}$$

where $\mathbf{D}$ and $\boldsymbol{\mathcal{E}}$ 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

$$\mathbf{D} = \epsilon_0 \boldsymbol{\mathcal{E}} + \mathbf{P} = \epsilon_r \epsilon_0 \boldsymbol{\mathcal{E}}, \tag{2.9}$$

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

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

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

$$\bar{\omega}_p^2 = \omega_p^2/\epsilon_{\omega\to\infty} = \frac{1}{\epsilon_{\omega\to\infty}} \frac{ne^2}{\epsilon_0 m}, \tag{2.11}$$

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

$$\mu_0 \ddot{\mathbf{D}} = \nabla^2 \boldsymbol{\mathcal{E}}, \tag{2.12}$$

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

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

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 $\mathbf{P} = -\epsilon_0 \boldsymbol{\mathcal{E}}$. 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

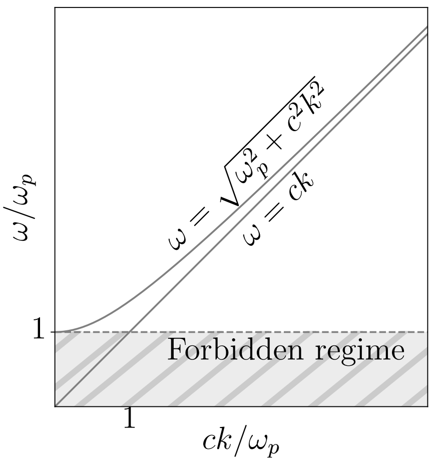


Figure 2.1: the dispersion relation for the plasma waves as well as the optical limit. No propagation occurs in the shaded region.

the plasma is given by

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

where $\mathbf{H}$ is the magnetic field strength, and the latter holds true in the long wavelength limit $\langle\mathbf{v}\rangle = \hbar\langle\mathbf{k}\rangle/m \to 0$, 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

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

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 $\mathbf{P} = -\epsilon_0 \boldsymbol{\mathcal{E}}$. 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

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

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

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

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

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

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:

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

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

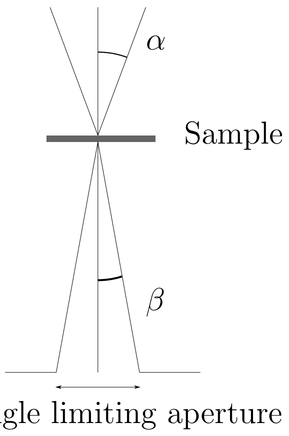


Figure 2.2: schematic of convergence angle, α, collection angle, β, and the angle limiting aperture in EELS experiments.

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:

$$R = \frac{\gamma m \nu}{eB}. \tag{2.23}$$

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

$$Z(E) = I_0 R(E), \tag{2.24}$$

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]:

$$J(E) = Z(E) * S(E). \tag{2.25}$$

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:

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

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]

$$s(\nu) = I_0 \log(j(\nu)/z(\nu)). \tag{2.27}$$

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:

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

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

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

where

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

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

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

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

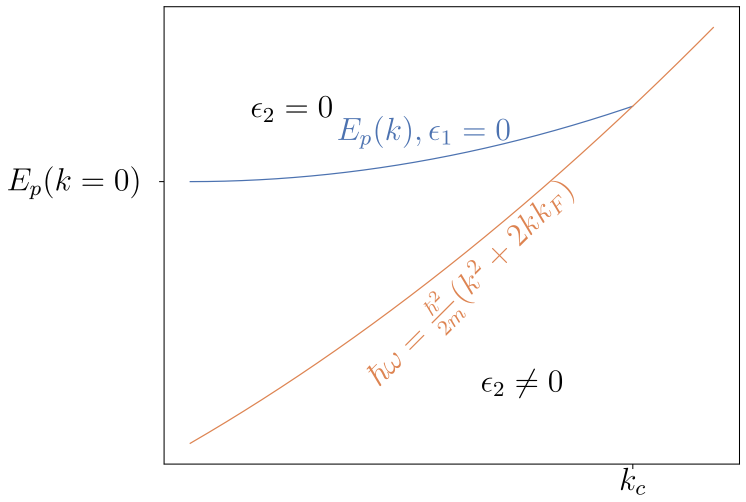


Figure 2.3: schematic of the (k, ω)-plane for the dielectric function as derived in the Lindhart model. The orange curve separates the regions where $\epsilon_2 = 0$ and where $\epsilon_2 \neq 0$

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

[1]Commercially available at `https://www.goodfellow.com/p/al00-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

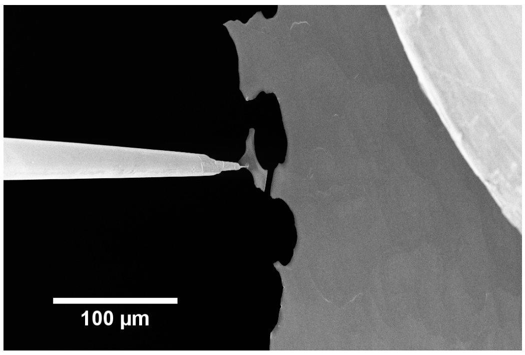


Figure 3.1: sample during transfer to `DENS` chip in a FIB. Image acquired by Per Erik Vullum

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 $\lesssim 100\,\text{meV}$ 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 $\leq 2\,\text{eV}$. 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 function[2] (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 parameters[3]. The remaining pixels were segmented using `scikit-image` [22] and added togehter to improve the SNR. Segmenting the pixels was based on their measured

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

[3]Obvious 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

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

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

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

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.

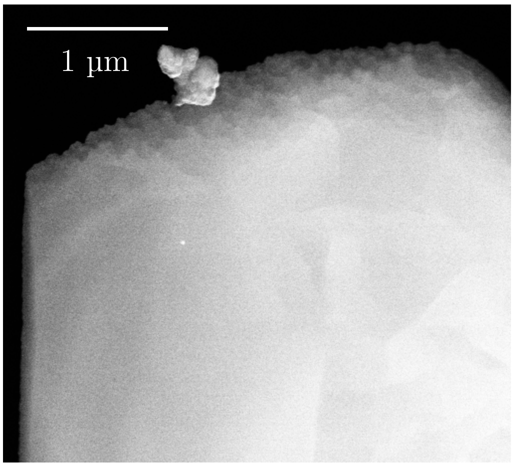
Chapter 4

Results and Discussion

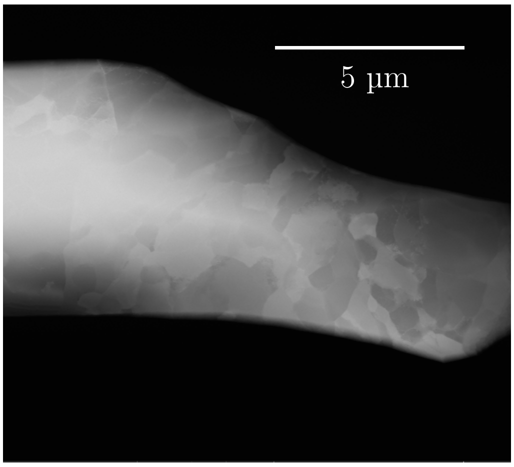
As a brief overview, an angular dark field (ADF) image of the sample is shown in Figure 4.1 along with an image taken in the FIB during sample preparation. It shows that the sample is mostly pure aluminium, though contaminated on one of its edges[1]. Examples of how the EEL signal looked like is plotted 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 $90 - 100\,\mathrm{meV}$, with the exception of scans taken at room temperature and at $120\,^{\circ}\mathrm{C}$. 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 $90\,\mathrm{meV}$, 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 $(4.3 \pm 0.2) \cdot 10^{6}\ \mathrm{S\,m^{-1}}$ 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 $14.94 \pm 0.01\,\mathrm{eV}$ at room temperature, declining with approximately $0.5\,\mathrm{meV\,K^{-1}}$. Solving Equation 2.11 for the electron density

[1]This region was not included in the further data handling

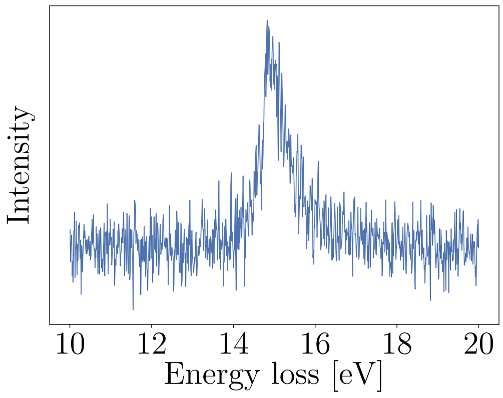


(a)

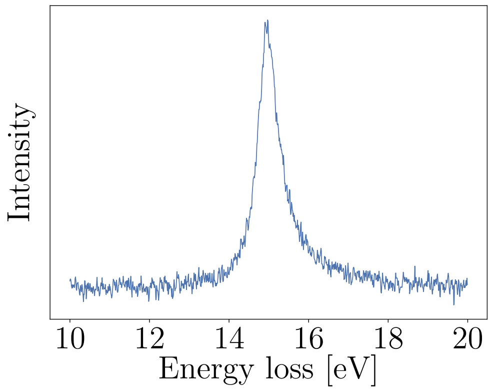


(b)

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.



(a)



(b)

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

gives $n = 18 \cdot 10^{22}$ cm^{-1}. The decline in plasmon energy for increasing temperature can be explained through Equation 2.22, measuring a temperature coefficient of approximately $23 \cdot 10^{-6}$ K^{-1}. Comparing both this decline and the electron density to the values enlisted in Table 1 reveals that these values are is 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.

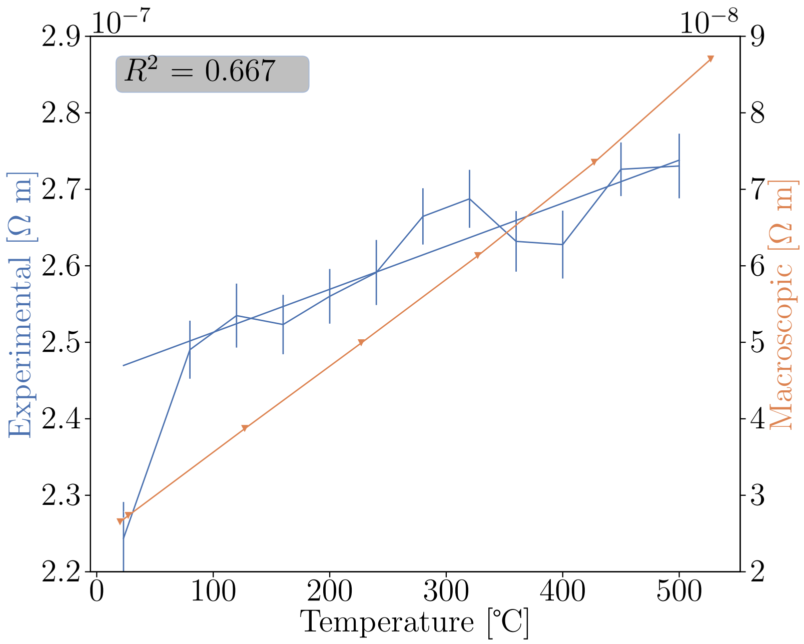


Figure 4.3: measured resistivity (inverse conductivity) of aluminium as function of temperature compared to the macroscopic resistivity. Note that the offset for the two colors are different, but the scale is similar. For the regression analysis, the data at room temperature were left out, see also Figure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

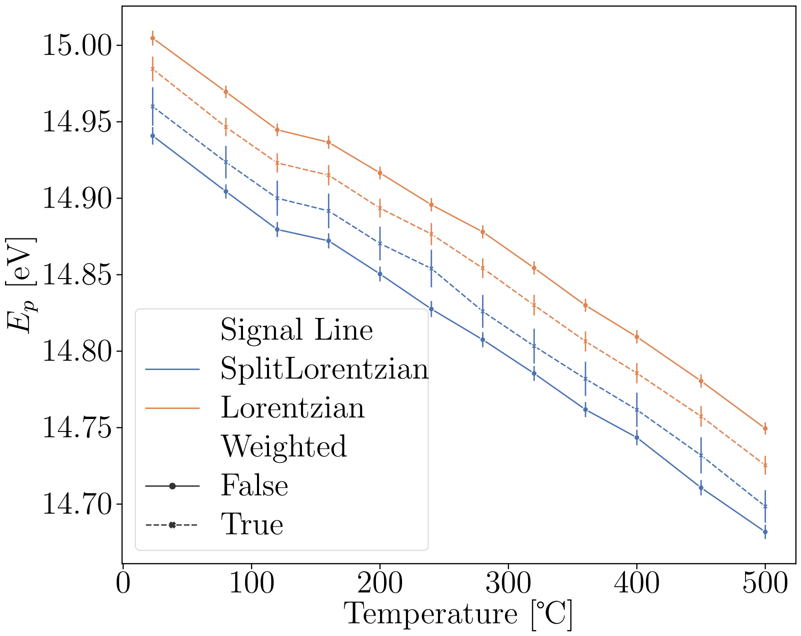


Figure 4.4: temperature dependency for the measured plasmon energy for the SplitLorentzian and the Lorentzian function.

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, $k \to 0$. 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 $k_0 = 3191\,\mathrm{nm}^{-1}$, allowing all momentum transfers up to $k \approx k_0\theta_{\text{collection aperture}} = 319.1\,\mathrm{nm}^{-1}$ 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 $k \not\to 0$ is non-negligible even for $k > k_c = 1.3\ \text{Å}^{-1}$ (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 standards[2]. 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 $k < 0.74k_F = 1.3\ \text{Å}^{-1}$, 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

[2]or 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].

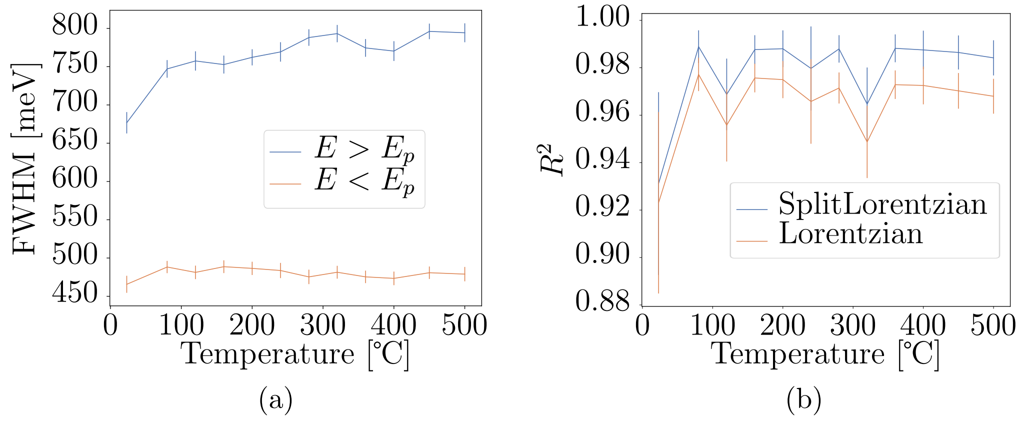


Figure 4.5: temperature dependency for a) the FWHM for the SplitLorentzian, and b) the R^2 value for the SplitLorentzian and Lorentzian function.

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 $E < E_p$.

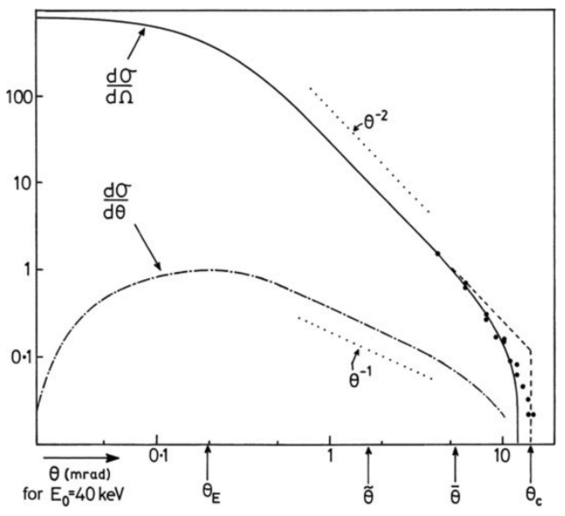


Figure 4.6: the differential cross section as calculated by Ferrell [28] (solid) and using a sharp cutoff approximation [10]. The figure is taken from Ref. [10].

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

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

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 level[3]: 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

[3]Quantitative analysis showed a noisy and unpredictable values of the energy shift, though of the order $20 - 30\,\text{meV}$, being only 2-3 times the energy dispersion.

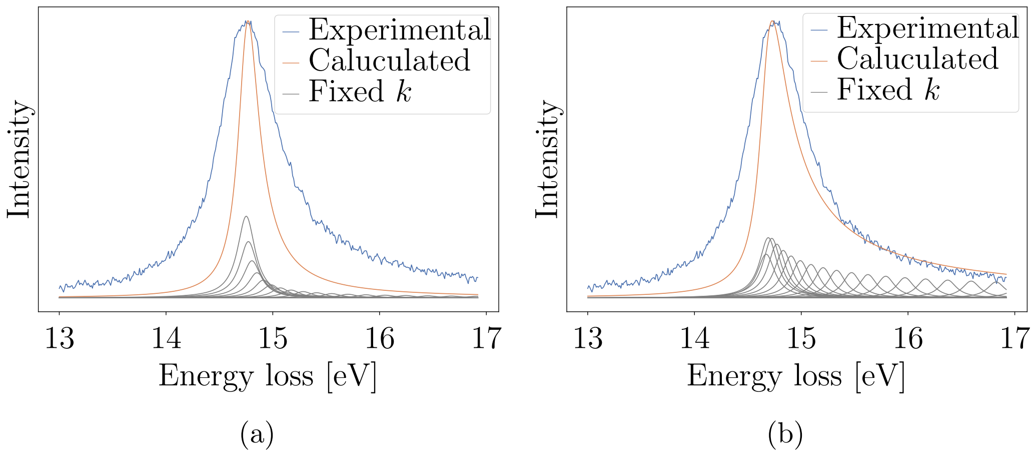


Figure 4.7: estimated shape of plasmon given a Lorentzian plasmon shape as calculated by a) Ferrell and b) the sharp cut-off approximation [10, 28]. The orange line is the sum of all the grey limes (intensity of grey lines not to scale with the other two).

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 $\Omega_p = 12.5 \pm 0.3\,\text{eV}$ 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-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 Migunov, 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 Haldane, 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, $\mathrm{Im}[-\frac{1}{\epsilon}(q,\omega)]$, 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 Louren ço 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

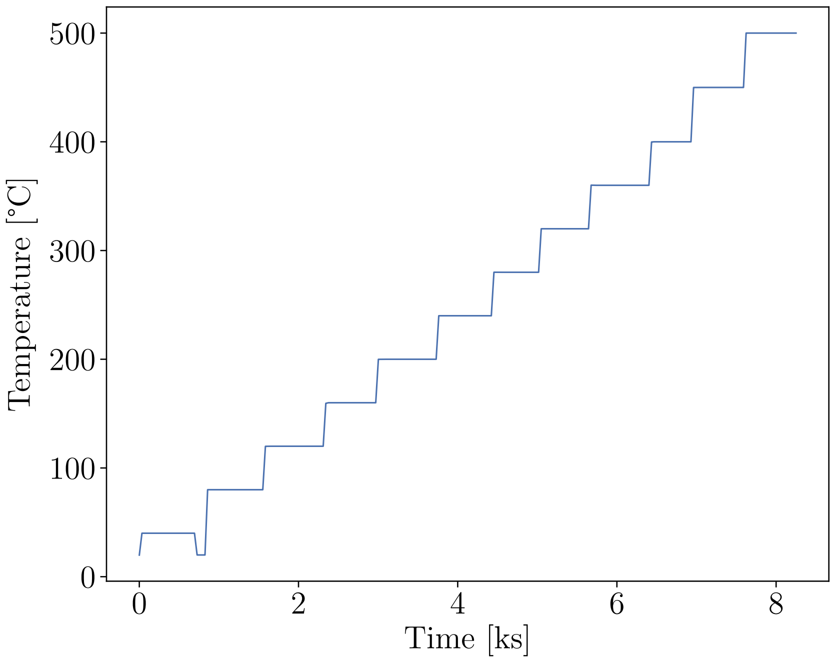
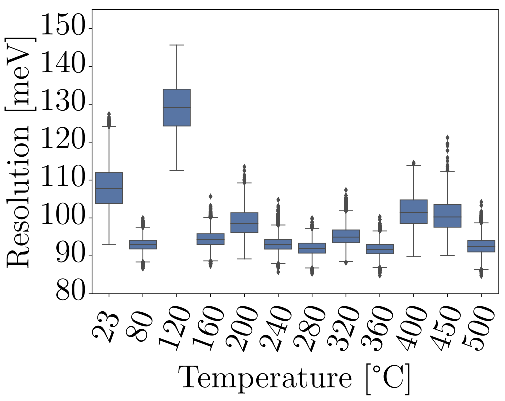


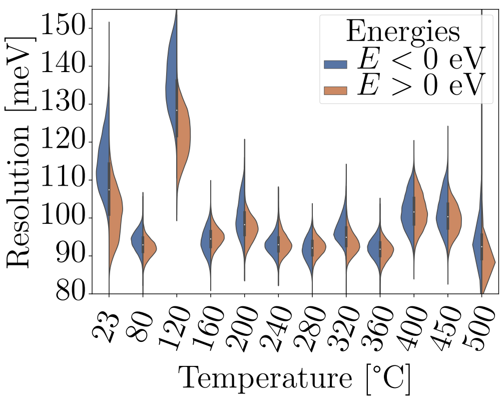
Figure A.1: measured temperature log of the sample during TEM operation.

Appendix B

Resolution



(a)



(b)

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

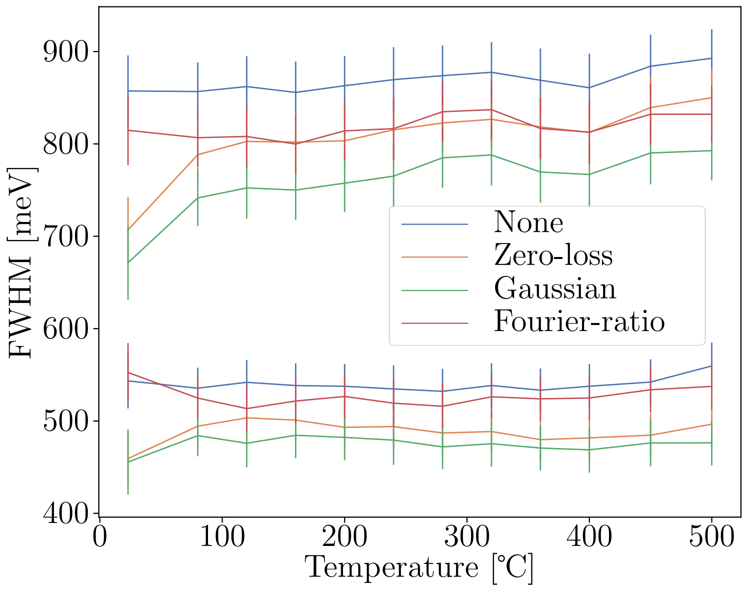


Figure C.1: measured temperature dependency for the signal for the different deconvolution techniques for a SplitLorentzian

Appendix D

Code

```python
from numpy import pi
"""This file contains the parameters that can be used for
    initialization of the curve-fitting alogrithm

The order of the dictionary is essential. Only one example is
    shown.
"""

class Param:
    def __init__(self, value, bmin, bmax):
        self.value = value
        self.bmin  = bmin
        self.bmax  = bmax


drudec     = Param(14.8,12,18)
drudefwhm = Param(0.5 ,0 ,1)
drudea     = Param(2e5 ,0 ,1e9)


drude = {'center'  : drudec,
            'fwhm'     : drudefwhm,
            'area'      : drudea}



""" This file contains wrappers for the different signal lines.
    These include Gaussian, Lorentzian, Voigt, and more."""

from abc import ABC
import numpy as np
import sympy
import scipy.special
import logging




dx  = '+0.01'
dx_ = '-0.01'
def prod_id(x):
    while True:
        yield chr(ord('a')+x)
        x+=1
id_ = prod_id(0)

gauss_expr           = '(area * 2 * sqrt(log(2))/(fwhm*sqrt(pi)) * exp(-4*(log(2)) * ( x -center)**2/fwhm**2 )  ) '
loren_expr           = '  (area / pi * fwhm / 2 / (( x -center)**2 + fwhm**2/4))'
# gauss_expr      = 'area * exp(-4*(log(2)) * (x-center)**2/fwhm**2 )   '
# loren_expr      = 'area * ((x-center)**2 + fwhm**2/4)'
# voigt_expr = 'area* real(Gfwhm /(4*sqrt(pi*log(2)))   * scipy.special.wofz(2*sqrt(log(2)) /Gfwhm  * (x-center+1j*Lfwhm/2) ) )  '
voigt_expr           = '(area * real(2*sqrt(log(2))/sqrt(pi)*1/Gfwhm  * wofz(2*sqrt(log(2)) /Gfwhm  * (x-center+1j*Lfwhm/2) )  ))'
# splitvoigt_expr = 'area * (((1-frac)*'+gauss_expr.replace('area','1').replace('fwhm','fwhm1') + ' + frac*'+loren_expr.replace('area','1').replace('fwhm','fwhm1')+')* (x<=center*1) ' \
#                              '+ ((1-frac)*'+gauss_expr.replace('area','1').replace('fwhm','fwhm2') + ' + frac*'+loren_expr.replace('area','1').replace('fwhm','fwhm2')+')* (x >center*1) ) '
splitvoigt_expr      = 'area * ( 1/((1-frac)*'+gauss_expr.replace('area','1').replace('fwhm','fwhm1').replace(' x ','center'+dx_) + ' + frac*'+loren_expr.replace('area','1').replace('fwhm','fwhm1').replace(' x ','center'+dx_)+') * ((1-frac)*'+gauss_expr.replace('area','1').replace('fwhm','fwhm1') + ' + frac*'+loren_expr.replace('area','1').replace('fwhm','fwhm1')+')* (x<=center*1) ' \
                          '+  1/((1-frac)*'+gauss_expr.replace('area','1').replace('fwhm','fwhm2').replace(' x ','center'+dx) + ' + frac*'+loren_expr.replace('area','1').replace('fwhm','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) ) '
volume_expr             = '(area * x * fwhm * center**2 / ((x**2-center**2)**2  + (x*fwhm)**2 ) )'
step_expr               = '(x>start*1)'
inter_expr              = '((x>start)-(x>stop)*1)'
power_expr              = 'area*(x-center)**(-k)'
# power_expr      = 'area*(x-center * ((x-0.3)>center*1))**(-k)'     #Note!
eps_1_expr              = 'area * (1-center**2 / (x**2 + fwhm**2))'
eps_2_expr              = 'area * fwhm*center**2/(x*(x**2+fwhm**2))'
experimental_expr       = '(1-area* (center**2-x**2)/((x**2-center**2)**2+x**2*fwhm**2))'
lognormal_expr          = 'area * 1/(x*sigma*sqrt(2*pi))*exp(-(log(x)-center)**2/(2*sigma**2))'     # see e.g. wikipedia
dho_expr                = 'area * sigma / (pi*(1-exp(-center/gamma)))  *  (1/((x-center)**2 +sigma**2) - 1/((x+center)**2 + sigma**2))  '
# experimental_expr = 'area*(1+ (1+1j*fwhm/x)*()   /  '
fano_expr               = '(area * abs(fwhm/2- background*exp(1j*phi)*(1j*( x -center)-fwhm/2) )**2   /   (( x -center)**2 + fwhm**2/4))'
# derivativeL_expr     = '-area * 16 * ( x - center) * fwhm  / (pi *(4*( x -center)**2+fwhm**2)**2 )'
# 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 )'
# 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)'
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)     )'

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 )       )'
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

```

```python
def step_func(x, start):
    return ((x-start) > 0)*1
def define_on_interval(x, start, stop):
    return step_func(x,start) - step_func(x,stop)


def gaussian(x,center,fwhm,area):
    sigma = fwhm/(2*np.sqrt(2*np.log(2)))
    return area * 1/(sigma*np.sqrt(2*np.pi)) * np.exp(-(x-center)**2/(2*sigma**2))



def lorentzian(x, center, fwhm, area):
    return area * 1/np.pi * fwhm/2 * 1/((x-center)**2 + (fwhm/2)**2)

def double_lorentzian(x,center,fwhm1,fwhm2,area):
    return lorentzian(x,center,fwhm1,area)*(x<=center)+lorentzian(x,center,fwhm2,area*fwhm2/fwhm1)*(x>center)

def voigt(x:np.array,c,FWHM,gamma,area):
    gamma /= 2
    sigma = FWHM/(2*np.sqrt(2*np.log(2)))
    z = (x - c + 1j * gamma) / (np.sqrt(2)*sigma)
    V = scipy.special.wofz(z)/(np.sqrt(2.0*np.pi)*sigma)
    return area*np.real(V)
def volumePlasmon(x,center,fwhm,area):
    return area * x * fwhm * center**2 / ((x**2-center**2)**2  + (x*fwhm)**2 )


class SignalLine(ABC):
    def __init__(self,parameters:dict, expr:str = None, id=None, symbols=None) -> None:
        super().__init__()
        self.parameters = parameters
        self.expr       = expr
        self.id         = id
        self.symbols    = symbols
    def append_step_func(self,start:float,stop:float = None):
        """Defines the interval the signal line is defined at"""
        if stop is None:
            self.expr += '*' + step_expr.replace('start',str(start))
        else:
            self.expr += '*' + inter_expr.replace('start',str(start)).replace('stop',str(stop))


class Lorentzian(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = loren_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('fwhm', 'fwhm'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class Gaussian(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = gauss_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('fwhm', 'fwhm'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class Voigt(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = voigt_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('Gfwhm', 'Gfwhm'+self.id).replace('Lfwhm', 'Lfwhm'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)

class SplitVoigt(SignalLine):
    """Takes in the center position, fwhm1 (<= center), fwhm2 (>= center), frac"""
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        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)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class VolumePlasmon(SignalLine):
    """Inverted Drude-Sellmeier"""

    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = volume_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('fwhm', 'fwhm'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class DHO(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = dho_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('sigma', 'sigma'+self.id).replace('gamma', 'gamma'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)

class Fano(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        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)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class DerivativeLorentzian(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = derivativeL_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('fwhm1', 'fwhm1'+self.id).replace('fwhm2', 'fwhm2'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class SplitLorentzian(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = splitLorent_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('fwhm1', 'fwhm1'+self.id).replace('fwhm2', 'fwhm2'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)

class SplitGaussian(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = splitGauss_expr.replace(’area’, ’area’+self.id).replace(’center’,’center’+self.id).replace(’fwhm1’, ’fwhm1’+self.id).replace(’fwhm2’, ’fwhm2’+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class Eps_1(SignalLine):
    """I dont think it is actually center, but just kept the name as it is the plasmon energy"""
    def __init__(self,  parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = eps_1_expr.replace(’area’, ’area’+self.id).replace(’center’,’center’+self.id).replace(’fwhm’, ’fwhm’+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)
class Eps_2(SignalLine):
    """I dont think it is actually center, but just kept the name as it is the plasmon energy"""
```

```python
    def __init__(self,  parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = eps_2_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('fwhm', 'fwhm'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)

class Experimental(SignalLine):
    """I dont think it is actually center, but just kept the name as it is the plasmon energy"""
    def __init__(self,  parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = experimental_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('fwhm', 'fwhm'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class LogNormal(SignalLine):
    def __init__(self, parameters:dict, name:str = None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        expr = lognormal_expr.replace('area', 'area'+self.id).replace('center','center'+self.id).replace('sigma', 'sigma'+self.id)
        self.symbols = {}
        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()):
            self.symbols[sym] = parameters[param]
        super().__init__(parameters, expr, self.id, self.symbols)


class PowerLaw(SignalLine):

    def __init__(self, parameters: dict, name=None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        self.symbols = {}

        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()):
            self.symbols[sym] = parameters[param]
        expr = power_expr.replace('area','area'+self.id).replace('k','k'+self.id).replace('center','center'+self.id)

        super().__init__(parameters, expr, self.id, self.symbols)


class Constant(SignalLine):
    """The parameter it takes in must be a dictionary with a key C. All other parameters are not used.

    TODO: fix the above"""
    def __init__(self, parameters: dict, name=None) -> None:
        if name is None:
            self.id = next(id_)
        else:
            self.id = name
        self.symbols = {}
        # for sym, param in zip(sympy.symbols('C'.replace('C','C'+self.id)), parameters.keys()):
```

```
        sym, param = sympy.symbols('C'.replace('C','C'+self.id))
    , parameters['C']
        self.symbols[sym] = param
        super().__init__(parameters, 'C'.replace('C','C'+self.id
    ), self.id, self.symbols)



"""Model.py"""
import numpy as np
from multiprocessing.pool import Pool
import logging
from tqdm.notebook import trange, tqdm
import MyFunc
from scipy.optimize import curve_fit
from signal_lines import *
import scripting
from skimage.segmentation import slic
from time import time
from scipy.stats import skew, kurtosis
import pickle


hbar = 6.582119569e-16     #eV s
e_0  = 8.8541878128e-12    #F/m
e    = 1.6e-19






class Model:
    """A model similar to that of hyperspy."""
    def __init__(self, data, x:np.array=None) -> None:
        self.data          = data
        self.signal_lines = {}
        self.symbols       = {}
        self.expr          = ''
        self.masks         = None
        self.N             = None
        self.x             = x



    def append(self, curve: SignalLine)-> None:
        """Appends a SignalLine to the model"""
```

```python
        if curve.id in self.signal_lines.keys():
            logging.warning('This is already in the model')
        self.signal_lines[curve.id] = curve

        for sym in curve.symbols.keys():
            self.symbols[sym] = curve.symbols[sym]
        self.expr  += '+'*(self.expr!='') +  curve.expr


    def append_labels(self, masks, N=None):
        """If N < number of labels the multifit algorithm will only fit for the first N labels. Can also use segment_model to use slic for this purpose"""
        if N is None:
            N = masks.max()
        self.masks = masks
        self.N     = N

    def segment_model(self, mask=None, n_segements=32, compactness=0.015, enforce=False, **kwargs):
        """The mask will ultimately make sure that only the TRUE pixels are curve fitted."""
        img = self.data
        segments  = slic(img, n_segments=n_segements, compactness=compactness, mask=mask, enforce_connectivity=enforce,**kwargs)
        self.append_labels(segments)

    def multifit(self, num_workers = 4, use_parallel = False,**kwargs) -> np.array:
        """Fits the appended curves to the dataset using scipy.optimize.curve_fit. Default is not parallelizing"""
        if self.masks is None:
            self.masks = np.ones(shape=(self.data.shape[0],self.data.shape[1]))
            self.N     = 1
        num_params = len(self.symbols.keys())
        for i in self.symbols.keys():
            if self.symbols[i].bmin == -1:
                num_params -=1
        results  = np.zeros(shape=(self.data.shape[0], self.data.shape[1], num_params))   #2 for skew and kurtosis
        pcov     = np.zeros(shape=(self.data.shape[0], self.data.shape[1], num_params, num_params)) # , dtype=object)
        iterable = [[self, results,self.masks==i] for i in range(1,self.N+1)]
        t = []
        if not use_parallel:

            for i in range(1,self.N+1):
                tic = time()

                fit(self,results, pcov,self.masks==i, maxfev=2000, **kwargs)
                toc = time()
                t.append((toc-tic)/60)
                # print(f’Segement {i} took {(toc-tic)/60} min’)
            return results, pcov
        res = []
        # the below works, but not with a tqdm
        with Pool(num_workers) as pool:
            for result in tqdm(pool.starmap(fit, iterable, chunksize=1)):
                res.append(result)
                pass
        res = sum([res[i][:,:] for i in range(len(res))]) #returns only one array
        return res

    def compile_function(self) -> None:
        scripting.compile_function(self)


def fit(self, result, pcov ,mask = None, maxfev:int = 2000, **kwargs):
    if ’weighted’ not in  kwargs.keys():
        weighted=False
    else:
        weighted=kwargs[’weighted’]
    if mask is None:
        mask   = np.ones(shape= (self.data.shape[0], self.data.shape[1]))
        self.N = 2
    param_values = []
    param_min    = []
    param_max    = []
    for i in self.symbols.keys():
        if self.symbols[i].bmin == -1:
            continue
        param_values.append(self.symbols[i].value)
        param_min.append(self.symbols[i].bmin)
        param_max.append(self.symbols[i].bmax)
    for i in range(self.data.shape[0]):
        for j in range(self.data.shape[1]):
```

```python
            # print(mask[i,j]*1.0, (i,j))
            if mask[i,j]*1.0 == 0:
                continue
            if np.any(np.isnan(self.data[i,j])):
                mask[i,j] = 0
                logging.info(f'Pixel {i,j} has been removed, as it was not a number')
                print(f'Pixel {i,j} has been removed, as it was not a number')
                continue
            try:
                result[i,j], _ = curve_fit(MyFunc.model_function, self.x, self.data[i,j], p0=param_values, bounds=(param_min, param_max), maxfev=maxfev)
                if weighted:
                    temp_result = MyFunc.model_function(self.x,*result[i,j])
                    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)
                param_values = result[i,j]    # uses previous output as input. Improves speed by quite much
                if np.any((np.array(param_values) == np.array(param_min)) )or np.any((np.array(param_values) == np.array(param_max))):
                    logging.warning(f'Boundary value reached for coordinate {(i,j)}')

            except RuntimeError:
                logging.warning('Couldnt find an appropriate fit. Try to imporve initialization of increase maxfev')
                continue
                return result


    return result, pcov

class Result:
    def __init__(self, model:Model, result:np.array) -> None:
        self.model = model
        self.data  = result
    def print_averages(self, data = None, label = None, mask = None):
        if data is None:
            data = self.data.copy()
        if label is not None:
            data = self.data.copy()
            data[self.model.masks != label] = np.nan
        i = 0
        for sym in self.model.symbols.keys():
            if self.model.symbols[sym].bmin == -1:
                print(f'{sym}: {self.model.symbols[sym].value}')
            else:
                print(f'{sym}: {np.nanmean(data[:,:,i])}    {np.nanstd(data[:,:,i])}')
                i+=1

    def save_model(self, fname:str, mask, skewness=None, kurtosis=None, in_correct_folder = True):
        """Skewness and kurtosis is appended after everything else. Pass it as a tuple of (mean, std)"""
        if in_correct_folder:
            fname = f'Results\{fname}'
        n_array = self.data
        # print(n_array.shape)
        results = {}
        i = 0
        for sym in self.model.symbols.keys():
            if self.model.symbols[sym].bmin == -1:
                # print(f'{sym}: is fixed')
                continue
            # print(np.nanmean(n_array[:,:,i][mask]))
            results[sym.name] = (np.nanmean(n_array[:,:,i][mask])  , np.nanstd(n_array[:,:,i][mask]))
            i+=1
        if skewness is not None:
            results['skewness'] = skewness
        if kurtosis is not None:
            results['kurtosis'] = kurtosis
        f =  open(f'{fname}.pkl', 'wb')
        pickle.dump(results,f)
        f.close()

    def get_pixel_results(self,x,y):
        r = []
        i = 0
        for sym in self.model.symbols.keys():
            if self.model.symbols[sym].bmin == -1:
                r.append(self.model.symbols[sym].value)
            else:
                r.append(self.data[x,y,i])
                i+=1
        return r

    def as_dictionary(self, label=None):
        d = {}
        if label is not None:
            data = self.data.copy()
            data[self.model.masks != label] = np.nan
        else:
            data = self.data.copy()

        i = 0
        for sym in self.model.symbols.keys():
            if self.model.symbols[sym].bmin == -1:
                d[str(sym)] = self.model.symbols[sym].value
            else:
                d[str(sym)] = data[:,:,i]
                i+=1

        # for sym,param in zip(self.model.symbols, range(data.shape[-1])):
        #     d[str(sym)] = data[:,:,param]
            # print(f'{sym}: {data[:,:,param]}    {data[:,:,param]}')
        return d

    def prod_latex_file(self,filename=r'Tables\test_table.tex', label=None, significant_digits = 2):
        """This function is not the best"""

        if label is not None:
            data = self.data.copy()
            data[self.model.masks != label] = np.nan
        else:
            data = self.data.copy()
        file = open(filename,'w')
        for sym, param in zip(self.model.symbols.keys(),range(data.shape[-1])):
            line = f"""{sym} & {np.round(np.nanmean(data[:,:,param]),significant_digits)} \pm {np.round(np.nanstd(data[:,:,param]), significant_digits)}\\ \n"""
            file.write(line)
        file.close()




"""scripting.py"""
```

```
""" 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."""
def compile_function(model, file_name = 'fitting\\MyFunc.py'):
    with open(file_name, 'w') as file:
        args = ''
        for arg in (model.symbols.keys()):
            if model.symbols[arg].bmin == -1:
                continue
            args += str(arg) +','
        file.write('from numpy import pi, sqrt, log, exp, real\n
    ')
        file.write('from scipy.special import wofz\n')
        file.write('from scipy.stats import rv_continuous\n')
        file.write(f'def model_function(x,{args[:-1]}):\n')
        line = model.expr
        for sym in model.symbols.keys():
            if model.symbols[sym].bmin == -1:
                line = line.replace(str(sym),str(model.symbols[
    sym].value))
        file.write(f'\treturn {line}')
        file.write('\n')
        file.write(f'class my_func_gen(rv_continuous):\n\tdef
    _pdf(self, x,{args[:-1]}):\n\t\t')
        file.write(f'\treturn {line}\n')
        file.write('''my_func = my_func_gen(name='my_fun')''')

    print('Function created')



# class my_func_gen(rv_continuous):
#     "Distribution for Scipy's goodness of fit"
#     def _pdf(self, x):
#         return np.exp(-x**2 / 2.) / np.sqrt(2.0 * np.pi)
# my_func = my_func_gen(name='my_func')




"""concatenate.py
The function(s) in this file takes in the lowloss and highloss
    from EELS data in the form of two Hyperspy files.

It thereby concatenates them into one single data file. The
    plots are for now merged at 11eV
```

```python

TODO: Fix the """
import numpy as np
import hyperspy.api as hs
import hyperspy
import logging
import eels_addon_hyperspy as eah

merge_energy = 11.                      # eV, where the high- and lowloss are merged
rrtol        = 1e-3



def merge(hl, ll, merge_at:float = merge_energy, rtol:float=rrtol, modifier = 'zero_loss', FWHM=0.09, cut_zlp=5.):
    """hl: Highloss, type hyperspy._signals.eels.EELSSpectrum

    ll: Lowloss, type hyperspy._signals.eels.EELSSpectrum"""
    # if modifier not in ['zero_loss', 'gaussian', 'fourier_ratio']:
    #     NotImplementedError('Only zero_loss and gaussian')
    assert hl.data.shape[0] == ll.data.shape[0] and hl.data.shape[1] == ll.data.shape[1]
    lowloss  = ll.isig[:merge_at*1.0].deepcopy()
    highloss = hl.isig[merge_at*1.0:].deepcopy()
    start = highloss.axes_manager['Energy loss'].offset
    stop  = start + highloss.axes_manager['Energy loss'].scale*highloss.axes_manager['Energy loss'].size
    x     = np.linspace(start,stop,num=highloss.axes_manager['Energy loss'].size)
    start = lowloss.axes_manager['Energy loss'].offset
    stop  = start + lowloss.axes_manager['Energy loss'].scale*lowloss.axes_manager['Energy loss'].size
    x_zlp = np.linspace(start,stop,num=lowloss.axes_manager['Energy loss'].size)
    x_tot = np.unique(np.concatenate([x_zlp, x]))
    if not np.all(x_tot == np.sort(x_tot)):
        logging.warning('x-values not sorted')
    y = np.zeros(shape=(highloss.data.shape[0],highloss.data.shape[1],x_tot.shape[0]))
    y[:,:,x_tot <= x_zlp[-1]] = lowloss                             #concatenates the counts
    y[:,:,x_tot >= x_zlp[-1]] = highloss                            #concatenates the counts
    s_ = hyperspy._signals.eels.EELSSpectrum(y)
    s_.axes_manager[2].name = 'Energy loss'
    s_.axes_manager['Energy loss'].offset = x_tot[0]
    s_.axes_manager['Energy loss'].scale = (x_tot[-1]-x_tot[0])/s_.axes_manager['Energy loss'].size
    s_.axes_manager['Energy loss'].units  = 'eV'
    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)):
        logging.warning('Scale is off!')
    if modifier=='zero_loss':
        return x_tot, s_, s_.fourier_log_deconvolution(lowloss, add_zlp=False)
    if modifier=='gaussian':
        return x_tot, s_, eah.fourier_log_deconvolution(s_,s_.isig[:cut_zlp*1.0], add_zlp=False, FWHM=FWHM)
    if modifier=='fourier_ratio':# Not really a modifier, but its convenient for now
        return x_tot,s_, s_.fourier_ratio_deconvolution(s_.isig[:cut_zlp*1.0], fwhm=FWHM, extrapolate_lowloss=False, extrapolate_coreloss=False)
    if modifier=='None':
        return x_tot,s_, s_
    NotImplementedError('modifier has to be specified. Set it to str(None) if no deconvolution')






"""Some handy functions to use in several python scripts"""
import numpy as np
temperatures = [23,80,120,160,200,240,280,320,360,400,450,500]



def create_reconvolve_func():
    raise NotImplementedError


def fourier_ratio(y:np.array, zlp:np.array,reconvolve_func:np.array=None):
    """Returns the Fourier signal of the reconvolved signal"""
    if reconvolve_func is None:
        reconvolve_func = create_reconvolve_func()
    return np.fft.fft(reconvolve_func)*np.fft.fft(y)/np.fft.fft(zlp)
```

```


def produce_filename(T,parameters):
    # print(parameters)
    components             = parameters['components']
    class_type             = str(type(components[0]))[21:-2]
    # class_type             = parameters['class_type']
    modifier               = parameters['modifier']
    if not parameters['deconvolve']:
        modifier = 'None'
    energy_interval        = parameters['energy_interval']
    curve_FWHM             = parameters['curve_FWHM']
    cut_zlp                = parameters['cut_zlp']
    add_constant           = parameters['add_constant']
    align_on_plasmon_peak = parameters['align_on_plasmon_peak']
    subtract_median        = parameters['subtract_median']
    add_pixels             = parameters['add_pixels']
    # T                      = parameters['T']
    if 'add_to_dict_name' in parameters.keys():
        addition = parameters['add_to_dict_name']
        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}'

    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'
def get_zlp_thermal():
    zlp = {}
    for T in temperatures:
        avg = []
        std = []
        val = []
        for i in range(1,10):
            try:
                f = f'ZLP/{(T,i)}.npy'
                # print(np.any(np.isnan(np.load(f))))
                val.append(np.load(f))
            except:
                print(f)
                pass
```

```python
        avg = np.mean(np.array(val))
        std = np.std(np.array(val))
        zlp[T] = (avg,std)
    return zlp

def derivative(y,x, order = 2):
    """Takes the first derivative. Available for forward
    differences (order=1) and central differences (order=2)"""
    dx = (x[-1]-x[0])/x.shape[0]
    if order==1:
        return (y[1:] - y[:-1])/dx
    elif order==2:
        return np.gradient(y,x)
    else:
        raise NotImplementedError('Implementation for higher
    order finite differences of the first derivative is not
    implemented')

def get_zlp_thermal_intensities():
    zlp = {}
    for T in temperatures:
        avg = []
        std = []
        val = []
        for i in range(1,10):
            try:
                f = f'ZLP/{(T,i)}_integral.npy'
                # print(np.any(np.isnan(np.load(f))))
                val.append(np.load(f))
            except:
                print(f)
                pass
        avg = np.mean(np.array(val))
        std = np.std(np.array(val))
        zlp[T] = (avg,std)
    return zlp


def line(x,a,b):
    return a*x+b
temp_coeff = 0.00429
R_Al       = 2.65e-8
def theoretical(T):
    return R_Al * (1+temp_coeff*(T-(20)))


mask = np.zeros(shape=(51,60)) # Shape of the images
```

```
mask[10:,10:] = 1
mask = mask > 0.5


def reshape_segments(segments, approved_files, end_shape):
    """takes the in the good files and the end-shape of the
    experimental data as well as the segment mask and fits the
    shape appropariately"""
    seg = {}
    for T in segments.keys():
        s = np.zeros(end_shape)
        count = 0
        for i in range(0,11):
            if T == 23:
                if ('RM',i+1) not in approved_files:
                    print(T,i)
                    continue
            else:
                if (T,i+1) not in approved_files:
                    continue
            s[:,:,i] = segments[T][:,:,count]
            print(count)
            count+=1
        seg[T] = s
    return seg


%matplotlib qt                # This is an .ipynnb file used for
     getting the information regarding the resolution
import numpy as np
import os
import hyperspy.api as hs
import hyperspy
import matplotlib.pyplot as plt
import scipy.ndimage as nd
from datetime import date
from tqdm import trange, tqdm
import logging
from time import time
import sys
sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
     - 2023 V r\\Master\\Data handling\\fitting')
import signal_lines
import Params
import Model
import MyFunc
```

```python
from Model import hbar, e_0         # note that hbar is taken in eV
import rcparams
rcparams.main()                     # sets the rcParams for plotting

# For automatic loading of the scripts
# This is needed in order for the MyFunc.py to be updated when needed
%load_ext autoreload
%autoreload 2

############# New cell

ZLP = signal_lines.SplitGaussian(Params.testing_splitG,'ZLP')
ZLP.append_step_func(-1,1)
m = Model.Model(np.array(0),np.array(0))
m.append(ZLP)
m.compile_function()

############ New cell

########################### Loading the data ###########################
temperatures = ['80','120','160','200','240','280','320','360','400','450','500']
temperatures = [int(T) for T in temperatures]
lowloss      = {}
tic = time()
result = {}
for T in tqdm([23,160,200,240,280,320,360,400,450,500]):
    result[T] = []
    for i in range(1,10):
        if T == 23:
            temp = hs.load(r'Cambridge_7\S1_RMTemp _ 0.01 s_58nmpixelSTEM SI'+str(i)+'.dm4')
        else:
            temp = hs.load(r'Cambridge_7\S1_'+str(T)+'CTemp _ 0.01s_58nmpixel'+str(i)+'.dm4')
        try:
            l_ = temp[-2]
            l_.align_zero_loss_peak(print_stats=False, show_progressbar=False, signal_range=(-1.,1.))
            start = l_.axes_manager['Energy loss'].offset
            # stop  = start + l_.axes_manager['Energy loss'].scale*l_.axes_manager['Energy loss'].size
```

```
                x = start+np.arange(l_.axes_manager['Energy loss'].
    size)*l_.axes_manager['Energy loss'].scale

                ZLP = signal_lines.SplitGaussian(Params.
    testing_splitG, 'ZLP')
                m = Model.Model(l_.data,x)
                m.append(ZLP)
                # m.compile_function()
                # m.append(ZLP)
                res, pcov = m.multifit(show_progressbar=False,
    iterpath='serpentine')
                result[T].append(res)
                # val = m.components.Gaussian.sigma.map['values']*2*
    np.sqrt(2*np.log(2))
                np.save('ZLP/'+f'{T,i}_SplitG_L', res[:,:,1])
                np.save('ZLP/'+f'{T,i}_SplitG_R', res[:,:,2])
                # break
            except FileNotFoundError:
                print(i, 'FileNotFoundError')
            except TypeError:
                print(i, 'TypeError')
            except ValueError:
                print(i, 'ValueError')







"""experimental.py

The compilation of the signal line may not be optimal

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"""
import numpy as np
import os
import hyperspy.api as hs
import hyperspy
import matplotlib.pyplot as plt
import scipy.ndimage as nd
from datetime import date
from tqdm import trange
# import logging
```

```python
import logging
import pickle
from scipy.stats import linregress
from time import time
from scipy.stats import skew, kurtosis
from scipy.stats import goodness_of_fit
import concatenate
import sys
sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
    - 2023 V r\\Master\\Data handling\\fitting')
import signal_lines
import Params
import Model
import utils
import MyFunc
from Model import hbar, e_0        # note that hbar is taken in
    eV
import rcparams
rcparams.main()                    # sets the rcParams for
    plotting
logging.basicConfig(level=logging.DEBUG)
# For automatic loading of the scripts
# This is needed in order for the MyFunc.py to be updated when
    needed

tic = time()


def run_experimental(T, **kwargs):
    ###
    #### Set the parameters here ####
    T                = str(T)
    if 'energy_interval' in kwargs.keys():
        energy_interval = kwargs['energy_interval']
    else:
        energy_interval = (10.,16.)
    if 'deconvolve' in kwargs.keys():
        deconvolve = kwargs['deconvolve']
    else:
        deconvolve = False
    if 'modifier' in kwargs.keys():
        modifier = kwargs['modifier']
    else:
        modifier        = 'zero_loss'
        # modifier        = 'gaussian'
        # modifier        = 'fourier_ratio' # not really a
    modifier, but it works as a quick fix
    if 'curve_FWHM' in kwargs.keys():
        curve_FWHM = kwargs['curve_FWHM']               # The FWHM for the reconvolution function
    else:
        curve_FWHM      = 0.2
    if 'cut_zlp' in kwargs.keys():
        cut_zlp = kwargs['cut_zlp']
    else:
        cut_zlp         = 2.
    if 'add_constant' in kwargs.keys():
        add_constant = kwargs['add_constant']
    else:
        add_constant    = False
    if 'subtract_median' in kwargs.keys():
        subtract_median = kwargs['subtract_median']
    else:
        subtract_median = True
    if 'align_on_plasmon_peak' in kwargs.keys():
        align_on_plasmon_peak = kwargs['align_on_plasmon_peak']
    else:
        align_on_plasmon_peak = False
    if 'weighted' not in kwargs.keys():
        kwargs['weighted'] = False
    if 'add_pixels' in kwargs.keys():
        add_pixels = kwargs['add_pixels']
    else:
        add_pixels=False
    if 'derivative' in kwargs.keys():
        derivative = kwargs['derivative']
    else:
        derivative = False
    if 'components' in kwargs.keys():
        components = kwargs['components']
    else:
        if 'signal_line' in kwargs.keys():
            plasmon = kwargs['signal_line']
        else:
            if align_on_plasmon_peak:
                para = Params.splitvoigt_PP
            else:
                para = Params.splitvoigt
            plasmon = signal_lines.SplitVoigt(para,'Plasmon1')
        components = [plasmon]
    if add_constant:
        c       = signal_lines.Constant(Params.const, 'Constant')
        components.append(c)
```

```python

    # class_type = str(type(plasmon))[21:-2]
    class_type = str(type(components[0]))[21:-2]
    # dict_name  = 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 kwargs['weighted'])+f'{T}C'
    dict_name = utils.produce_filename(T,kwargs)
    # if 'add_to_dict_name' in kwargs.keys():
    #     dict_name += kwargs['add_to_dict_name']
    ###
    ########################### Loading the data ###########################
    all_data = []
    hyper    = []
    hyper_l  = []
    ll       = []
    x        = []
    x_tot    = []
    s        = []
    for i in range(1,11):
        # print(str(i))
        if T == 'RM':
            temp = hs.load(r'Cambridge_7\S1_'+T+'Temp _ 0.01s_58nmpixelSTEM SI'+str(i)+'.dm4')
        elif T== '40':
            temp = hs.load(r'Cambridge_7\S1_'+T+'CTemp _ 0.01s_58nmpixel.060'+str(i)+'.dm4')
        else:
            temp = hs.load(r'Cambridge_7\S1_'+T+'CTemp _ 0.01s_58nmpixel'+str(i)+'.dm4')
        try:
            l_ = temp[-2]
            h_ = temp[-1]
            l_.align_zero_loss_peak(also_align=[h_], print_stats=False, show_progressbar=False, signal_range=(-1.,1.))
            l = l_.isig[:12.]

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

        # To make sure that the pixels are defined over same energy interval
            if deconvolve:
                x_test, _, s_ = concatenate.merge(hl=h_,ll=l_, modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
            else:
                x_test, s_, _ = concatenate.merge(hl=h_,ll=l_, modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
            if s_.axes_manager['Energy loss'].offset > -0.4:
                # print(f'Skipped {i} during loading due to ')
                logging.debug(f'Skipped {i} during loading due to offset value')
                continue
            x_tot.append(x_test)
            if derivative:
                s.append(s_.isig[-1.5:20.].isig[:1900].derivative(axis=-1))
                all_data.append(s_.derivative(axis=-1).data)
            else:
                s.append(s_.isig[-1.5:20.].isig[:1900])
                all_data.append(s_.data)
            hyper.append(    h)                          # if not sliced, the different scans will not have the same dimension
            hyper_l.append( l)                           # if not sliced, the different scans will not have the same dimension
            # all_data.append(s_.derivative(axis=-1).data)      # used to be h
            ll.append(      l.data)                      # if not sliced, the different scans will not have the same dimension
            # start = h.axes_manager['Energy loss'].offset
            # stop  = h.axes_manager['Energy loss'].scale*h.axes_manager['Energy loss'].size + start
            # x.append(np.linspace(start,stop, num = h.axes_manager['Energy loss'].size))
            # break
        except FileNotFoundError:
            print((T,i), 'FileNotFoundError')
        except TypeError:
            print((T,i), 'TypeError, typically EELS spectrum not subscriptable')
        except ValueError:
            print((T,i), 'ValueError')
    x      = x_tot # overriding to update for new programming
    logging.info(f'Straight after loading (and deconvolution if performed): x_high = {x[0][-1]}, x_low = {x[0][0]}')
    # print(f'Straight after loading (and deconvolution if performed): x_high = {x[0][-1]}, x_low = {x[0][0]}')
    ###

    if align_on_plasmon_peak:
        temp          = [signal.isig[10.:].inav[20:,20:].deepcopy() for signal in s]
        [temp[i].align_zero_loss_peak(print_stats=False, show_progressbar=False) for i in trange(len(temp))]
        s = temp


    ###

    ########################### Adding the pixels together ###########################

    if add_pixels:
        offset_low  = np.array([i.axes_manager['Energy loss'].offset for i in s]).max()
        adjusted    = [i.isig[offset_low:]  for i in s]
        offset_high = np.array([i.axes_manager['Energy loss'].offset +  i.axes_manager['Energy loss'].size * i.axes_manager['Energy loss'].scale for i in adjusted]).min()
        adjusted    = [i.isig[:offset_high]  for i in adjusted]
        size = adjusted[0].axes_manager['Energy loss'].size
        scale = adjusted[0].axes_manager['Energy loss'].scale
        hyper = adjusted[0]
        deleted = 0
        for i in range(1,len(adjusted)):
            try:
                hyper += adjusted[i]
            except ValueError:
                print(i,'ValueError')
                del adjusted[i-deleted]
                deleted+=1
                continue
            except IndexError:
                print(i,'IndexError')

        if subtract_median:
            all_data = np.array([i.data for i in adjusted]).mean(axis=0)
            all_data -= np.median(all_data, axis=-1).reshape((all_data.shape[0], all_data.shape[1],1))
            all_data = [all_data]
        else:
            all_data = [np.array([i.data for i in adjusted]).mean(axis=0)]

        x = [np.linspace(start=offset_low, stop=offset_high, num
=size)]
        logging.info(f'After summing over the scans: x_high={x
[0][-1]}, x_low={x[0][0]}')
        # print(f'After summing over the scans: x_high={x
[0][-1]}, x_low={x[0][0]}')

    ########### Run the curve-fitting algorithm and get the
results ###########
    # np.ones(shape=(data.shape[0],data.shape[1]))
    print('Starting curve fitting')
    for i in range(len(all_data)):
        if 'mask' in kwargs.keys():
            mask = kwargs['mask']
        else:
            mask = np.zeros((all_data[i].shape[0],all_data[i].
shape[1]))
            mask[10:,10:] = 1
            mask = mask > 0.5
        m        = Model.Model(all_data[i],x[i])
        skewness = (np.mean(skew(all_data[i][mask]    , bias=
True, axis=-1)),   np.std(skew(all_data[i][mask]    , bias=
True, axis=-1)))
        kurt     = (np.mean(kurtosis(all_data[i][mask], bias=
True, axis=-1)),   np.std(kurtosis(all_data[i][mask], bias=
True, axis=-1)))
        for comp in components:
            m.append(comp)
        if np.any(np.isnan(all_data[i])):
            mask[np.any(np.isnan(all_data[i]), axis=-1)] = False
            m.append_labels(masks=mask)
        else:
            m.segment_model(mask = mask)
        res, pcov = m.multifit(num_workers=8, use_parallel=False
, **kwargs)
                 # TODO: pcov now only for last scan!
        r = Model.Result(m,res)
        r.save_model(dict_name+str(i) ,mask, kurtosis=kurt,
skewness=skewness)
        test = r.data.copy()
        j = 0
        # print(T,'C')
        for sym in r.model.symbols.keys():
            if r.model.symbols[sym].bmin == -1:
                # print(f'{sym}: is fixed to {fwhm_gauss[i]}')
                continue
```

```python

        logging.debug(f'{T}: {sym}: {np.nanmean(test[:,:,j][mask])}     {np.nanstd(test[:,:,j][mask])}')
        j+=1
    # np.save(f'\numpy_results\{dict_name}', res)
    np.save(os.path.join('numpy_results',dict_name+f'_{i}'), res)
    np.save(os.path.join('numpy_results',dict_name+f'_{i}_std'), pcov)
    print('Saved')

###


# ################## Some plotting #################
# filter_zlp = 500*(not align_on_plasmon_peak)
# divide = 2
# i,j = -1,-2
# y = m.data[i:,j:].mean(axis=(0,1))-m.data[i:,j:].mean(axis=(0,1))[-1000:].max()/divide
# y = y[filter_zlp:].copy()
# # print(np.where(y==y.max()))
# if not modifier=='fourier_ratio':
#     max = np.where(y==y.max())[0][0]
#     # print(max[0])
#     x_low_ = np.where(np.abs(y[:max])==np.abs(y[:max]).min())[0][0]
#     x_high_= np.where(np.abs(y[max:])==np.abs(y[max:]).min())[0][0]+max

#     x_low = x[-1][x_low_+filter_zlp]
#     x_high = x[-1][x_high_+filter_zlp]

#     logging.debug(T+'C','Numerical FWHM:', (x_high-x_low),'eV')
#     np.save(os.path.join('numpy_results',dict_name+'numerical'),np.array(x_high-x_low))

# plot_s = slice(1500,-100)

# ##### Plotting the fit #####

# plt.figure()
# i,j = -3,-3
# plt.scatter(x[-1][plot_s],m.data[i,j][plot_s]-m.data[i,j][-1000:].max()/divide, s=1, alpha=1, label = '1 Pixel')
```

```
    # # plt.scatter()
    # 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')
    # 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')
    # 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')

    # if not modifier=='fourier_ratio':
    #     plt.scatter(x_low, 0)
    #     plt.scatter(x_high,0)
    # plt.plot(x[-1][plot_s], np.zeros(x[-1][plot_s].shape)
   ,'--')
    # # plt.scatter(x[-1],m.data[i:,j:].mean(axis=(0,1))-MyFunc.
   model_function(x[-1],*res[i,j]))
    # plt.title(f'{i,j}'+'_FWHM='+ str(curve_FWHM) +'_'+T+'C')
    # plt.legend()
    # plt.savefig(f'Figures\Curve_fit\{dict_name}.png')
    # # for sym,val in zip(m.symbols, res[i,j]):
    # #     print(sym,val)

    # known_params = {}
    # count = 0
    # for sym,val in zip(m.symbols, res[i,j]):
    #     # print(sym,val)
    #     known_params[str(sym)] = (val, count)
    #     count+=1

    # ###
    # # ##### Plotting the distribution of a parameter #####
    # if class_type=='Lorentzian' or class_type =='Fano':
    #     param = 'fwhmPlasmon1'
    #     param_index = known_params[param][-1]
    #     plt.figure()
    #     plt.hist(res[:,:,param_index][mask], bins=50)
    #     plt.title(param+'_FWHM='+ str(curve_FWHM) +'_'+T+'C')
    #     plt.savefig(f'Figures\{dict_name}.png')
    #     # plt.colorbar()

    # if class_type=='SplitVoigt':
    #     param = 'fwhm2Plasmon1'
    #     param_index = known_params[param][-1]
    #     plt.figure()
```

```python
#     plt.hist(res[:,:,param_index][mask], bins=50, range=(0,1))
#     plt.title(param)
#     plt.savefig(f'Figures\{dict_name}.png')
#     # plt.colorbar()
#     None
#     param = 'fwhm1Plasmon1'
#     param_index = known_params[param][-1]
#     plt.figure()
#     plt.hist(res[:,:,param_index][mask], bins=50, range=(0,1))
#     plt.title(param)
#     plt.savefig(f'Figures\{dict_name}.png')
# # plt.colorbar()
# ###
# ##### Plotting the map over the values #####
# plt.figure()
# plt.imshow(res[:,:,param_index])
# plt.title(param)



# toc = time()
# print(f'At {T}C, it took {(toc-tic)/60} min')

# index_a = 1500   #TODO: improve these
# index_b = 1700   #TODO: improve these
# assert modifier == 'zero_loss' or not deconvolve or modifier=='fourier_ratio'
# i,j=30,30
# reg = (linregress(MyFunc.model_function(x[-1][index_a:index_b],*res[i,j]),m.data[i,j,index_a:index_b]))
# reg_ = np.zeros(shape=mask.shape, dtype=object)
# for i in range(res.shape[0]):
#     for j in range(res.shape[1]):
#         if not mask[i,j]:
#             continue
#         reg_[i,j] = (linregress(MyFunc.model_function(x[-1][index_a:index_b],*res[i,j]),m.data[i,j,index_a:index_b]))
# np.save(os.path.join('Lin_results',dict_name),reg_)




```

```python

""" experimental2.py
Some parameters not availavle compared to experimental.py, but
    deals with memory better"""
import numpy as np
import os
import hyperspy.api as hs
import hyperspy
import matplotlib.pyplot as plt
import scipy.ndimage as nd
from datetime import date
from tqdm import trange
# import logging
import logging
import pickle
from scipy.stats import linregress
from time import time
from scipy.stats import skew, kurtosis
from scipy.stats import goodness_of_fit
import concatenate
import sys
sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
     - 2023 V r\\Master\\Data handling\\fitting')
import signal_lines
import Params
import Model
import utils
import MyFunc
from Model import hbar, e_0         # note that hbar is taken in
    eV
import rcparams
rcparams.main()                     # sets the rcParams for
    plotting



tic = time()
prel_mask = utils.mask      # preliminary mask, taking out vacuum
     and contamination

def run_experimental(T, **kwargs):
    ###
    #### Set the parameters here ####
    T                = str(T)
    if 'energy_interval' in kwargs.keys():
        energy_interval = kwargs['energy_interval']
    else:
        energy_interval = (10.,16.)
    if 'deconvolve' in kwargs.keys():
        deconvolve = kwargs['deconvolve']
    else:
        deconvolve = False
    if 'modifier' in kwargs.keys():
        modifier = kwargs['modifier']
    else:
        modifier        = 'zero_loss'
        # modifier        = 'gaussian'
        # modifier        = 'fourier_ratio' # not really a modifier, but it works as a quick fix
    if 'curve_FWHM' in kwargs.keys():
        curve_FWHM = kwargs['curve_FWHM']               # The FWHM for the reconvolution function
    else:
        curve_FWHM      = 0.2
    if 'cut_zlp' in kwargs.keys():
        cut_zlp = kwargs['cut_zlp']
    else:
        cut_zlp         = 2.
    if 'add_constant' in kwargs.keys():
        add_constant = kwargs['add_constant']
    else:
        add_constant    = False
    if 'subtract_median' in kwargs.keys():
        subtract_median = kwargs['subtract_median']
    else:
        subtract_median = True
    if 'align_on_plasmon_peak' in kwargs.keys():
        align_on_plasmon_peak = kwargs['align_on_plasmon_peak']
    else:
        align_on_plasmon_peak = False
    if 'weighted' not in kwargs.keys():
        kwargs['weighted'] = False
    if 'add_pixels' in kwargs.keys():
        add_pixels = kwargs['add_pixels']
    else:
        add_pixels=False
    if 'derivative' in kwargs.keys():
        derivative = kwargs['derivative']
    else:
        derivative = False
    if 'components' in kwargs.keys():
        components = kwargs['components']
    else:
        if 'signal_line' in kwargs.keys():
            plasmon = kwargs['signal_line']
        else:
            if align_on_plasmon_peak:
                para = Params.splitvoigt_PP
            else:
                para = Params.splitvoigt
            plasmon = signal_lines.SplitVoigt(para,'Plasmon1')
        components = [plasmon]
    if add_constant:
        c       = signal_lines.Constant(Params.const, 'Constant')
        components.append(c)

    # class_type = str(type(plasmon))[21:-2]
    class_type = str(type(components[0]))[21:-2]
    # dict_name  = 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 kwargs['weighted'])+f'{T}C'
    dict_name = utils.produce_filename(T,kwargs)
    x         = []
    x_tot     = []
    s         = []
    for i in range(1,11):
        # print(str(i))
        if T == 'RM':
            fname =r'Cambridge_7\S1_'+T+'Temp _ 0.01 s_58nmpixelSTEM SI'+str(i)+'.dm4'
        elif T== '40':
            fname = r'Cambridge_7\S1_'+T+'CTemp _ 0.01 s_58nmpixel.060'+str(i)+'.dm4'
        else:
            fname = r'Cambridge_7\S1_'+T+'CTemp _ 0.01 s_58nmpixel'+str(i)+'.dm4'
        try:
            temp = hs.load(fname)
            l_ = temp[-2]
            h_ = temp[-1]
            l_.align_zero_loss_peak(also_align=[h_], print_stats=False, show_progressbar=False, signal_range=(-1.,1.))
            # l = l_.isig[:12.]                # To make sure that the pixels are defined over same energy interval
            # h = h_.isig[10.:19.]
```

```
          # To make sure that the pixels are defined over same energy interval
        if deconvolve:
            x_test, _, scan = concatenate.merge(hl=h_,ll=l_, modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
        else:
            x_test, scan, _ = concatenate.merge(hl=h_,ll=l_, modifier=modifier, FWHM=curve_FWHM, cut_zlp=cut_zlp)
        if scan.axes_manager['Energy loss'].offset > -0.4:
            print(f'Skipped {(T,i)} during loading due to offset value')
            logging.debug(f'Skipped {i} during loading due to offset value')
            continue
        scan = scan.isig[-1.5:20.].isig[:1900]
        if derivative:
            scan = scan.derivative(axis=-1)

        ####### Switching to np arrays
        x = scan.axes_manager['Energy loss'].offset +scan.axes_manager['Energy loss'].scale *np.arange(scan.axes_manager['Energy loss'].size)
        data = scan.data

        ################################## Loading and deconvolving and derivative done, starting curve fit #################
        if 'mask' in kwargs.keys():
            mask = kwargs['mask']
        else:
            mask = prel_mask
        print(data.shape, x.shape)
        m = Model.Model(data,x)
        for comp in components:
            m.append(comp)
        if np.any(np.isnan(data)):
            mask[np.any(np.isnan(data), axis=-1)] = False
            m.append_labels(masks=mask)
        else:
            m.segment_model(mask = mask)
        res, pcov = m.multifit(num_workers=8, use_parallel=False, **kwargs)
        r = Model.Result(m,res)
        r.save_model(dict_name+str(i) ,mask)
        for j,sym in enumerate(r.model.symbols.keys()):
            if r.model.symbols[sym].bmin == -1:
                continue
                logging.debug(f'{T}: {sym}: {np.nanmean(r.data[:,:,j][mask])}    {np.nanstd(r.data[:,:,j][mask])}')
                j+=1
            np.save(os.path.join('numpy_results',dict_name+f'_{i}'), res)
            np.save(os.path.join('numpy_results',dict_name+f'_{i}_std'), pcov)
            print(f'Saved {(T,i)}')

        except FileNotFoundError:
            print((T,i), 'FileNotFoundError')
        except TypeError:
            print((T,i), 'TypeError, typically EELS spectrum not subscriptable')
        except ValueError as e:
            print((T,i), 'ValueError:' , e)





"""plot_class.py
Big thanks to Emil F. Christiansen for providing a notebook this .py file is heavily based on. Changed slightly to suit my purposes better"""
import matplotlib.pyplot as plt
from matplotlib.colors import SymLogNorm
import seaborn as sns
import pandas as pd

class SplitLorentzSegmented():
    """
    A class for storing Lorentz fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
        -----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, left fwhm, right fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm1':
    self.fwhm1, 'fwhm2': self.fwhm2, 'integral': self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={
    self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data
    .shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm1(self):
        return self.data[:, 1]

    @property
    def fwhm2(self):
        return self.data[:, 2]

    @property
    def integral(self):
        return self.data[:, 3]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0],
     self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', '
    integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting functions

        Returns:
        --------
        Returns the figure and axes generated by the plotting functions
        """
        if parameters is None:
            parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
```

```python
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes

class VoigtSegmented():
    """
    A class for storing Voigt fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
        -----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, Gaussian fwhm, Lorentzian fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'Gfwhm': self.Gfwhm, 'Lfwhm': self.Lfwhm, 'integral': self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def Gfwhm(self):
        return self.data[:, 1]

    @property
    def Lfwhm(self):
        return self.data[:, 2]

    @property
    def integral(self):
        return self.data[:, 3]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0], self.data.shape[1])), columns=['Ep', 'Gfwhm', 'Lfwhm', 'integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting functions

        Returns:
        --------
        Returns the figure and axes generated by the plotting functions
        """
        if parameters is None:
            parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes



class SplitVoigtSegmented():
    """
    A class for storing Lorentz fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
```

```python
        ----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, left fwhm, right fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm1': self.fwhm1, 'fwhm2': self.fwhm2, 'eta' : self.eta, 'integral': self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm1(self):
        return self.data[:, 1]

    @property
    def fwhm2(self):
        return self.data[:, 2]

    @property
    def eta(self):
        return self.data[:, 3]

    @property
    def integral(self):
        return self.data[:, 4]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0], self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', 'eta', 'integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting functions

        Returns:
        --------
        Returns the figure and axes generated by the plotting functions
        """
        if parameters is None:
            parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes


class FanoSegmented():
    """
    A class for storing Fano fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
        ----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, fwhm, integral (area), phi and background in 1st, 2nd, 3rd, 4th and 5th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm1': self.fwhm1, 'phi': self.phi, 'integral': self.integral, '
background' : self.background}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={
self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data
.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm1(self):
        return self.data[:, 1]


    @property
    def integral(self):
        return self.data[:, 2]

    @property
    def phi(self):
        return self.data[:, 3]


    @property
    def background(self):
        return self.data[:, 4]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0],
 self.data.shape[1])), columns=['Ep', 'fwhm1', 'integral', '
phi', 'background'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)
```

```

def plot(self, kind, parameters, *args, **kwargs):
    """
    Plot the fit results

    Parameters:
    -----------
    kind: str
        The kind of plot. Should be either "img" or "hist"
    parameters: str or list
        The parameter(s) to plot.
    *args: Optional arguments passed to plotting functions
    **kwargs: Optional keyword arguments passed to plotting functions

    Returns:
    --------
    Returns the figure and axes generated by the plotting functions
    """
    if parameters is None:
        parameters = ['Ep', 'fwhm1', 'phi', 'integral', 'background']
    else:
        if isinstance(parameters, str):
            parameters = [parameters]
        else:
            parameters = list(parameters)


    if kind == 'img':
        fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
        fig.suptitle(f'{self.name}')
        if len(parameters) == 1:
            axes = list([axes])
        kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
        for ax, parameter in zip(axes, parameters):
            ax.imshow(self.get_parameter(parameter), *args, **kwargs)
            ax.set_title(f'{parameter}')
    elif kind == 'hist':
        fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
        fig.suptitle(f'{self.name}')
        if len(parameters) == 1:
```

```python
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes




class VolumePlasmon():
    """
    A class for storing Voigt fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a Volume Plasmon instance

        Parameters:
        -----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, fwhm, and integral (area) in 1st, 2nd and 3re position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in C
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm': self.fwhm, 'integral': self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm(self):
        return self.data[:, 1]

    @property
    def integral(self):
        return self.data[:, 2]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0],
 self.data.shape[1])), columns=['Ep', 'fwhm', 'integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting
functions

        Returns:
        --------
        Returns the figure and axes generated by the plotting
functions
        """
        if parameters is None:
            parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']
```

```python
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes


"""plot_class.py
Big thanks to Emil F. Christiansen for providing a notebook this .py file is heavily based on. Changed slightly to suit my purposes better"""
import matplotlib.pyplot as plt
from matplotlib.colors import SymLogNorm
import seaborn as sns
import pandas as pd

class SplitLorentzSegmented():
    """
    A class for storing Lorentz fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
        ----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, left fwhm, right fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm1': self.fwhm1, 'fwhm2': self.fwhm2, 'integral': self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm1(self):
        return self.data[:, 1]

    @property
    def fwhm2(self):
        return self.data[:, 2]

    @property
    def integral(self):
        return self.data[:, 3]


```

```python

    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0], self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', 'integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting functions

        Returns:
        --------
        Returns the figure and axes generated by the plotting functions
        """
        if parameters is None:
            parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes

class VoigtSegmented():
    """
    A class for storing Voigt fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
        -----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, Gaussian fwhm, Lorentzian fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'Gfwhm': self.Gfwhm, 'Lfwhm': self.Lfwhm, 'integral': self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def Gfwhm(self):
        return self.data[:, 1]

    @property
    def Lfwhm(self):
        return self.data[:, 2]

    @property
    def integral(self):
        return self.data[:, 3]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0], self.data.shape[1])), columns=['Ep', 'Gfwhm', 'Lfwhm', 'integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
```

```python
    kind: str
        The kind of plot. Should be either "img" or "hist"
    parameters: str or list
        The parameter(s) to plot.
    *args: Optional arguments passed to plotting functions
    **kwargs: Optional keyword arguments passed to plotting functions

    Returns:
    ---------
    Returns the figure and axes generated by the plotting functions
    """
    if parameters is None:
        parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']
    else:
        if isinstance(parameters, str):
            parameters = [parameters]
        else:
            parameters = list(parameters)


    if kind == 'img':
        fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
        fig.suptitle(f'{self.name}')
        if len(parameters) == 1:
            axes = list([axes])
        kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
        for ax, parameter in zip(axes, parameters):
            ax.imshow(self.get_parameter(parameter), *args, **kwargs)
            ax.set_title(f'{parameter}')
    elif kind == 'hist':
        fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
        fig.suptitle(f'{self.name}')
        if len(parameters) == 1:
            axes = list([axes])
        df = self.dataframe
        for ax, parameter in zip(axes, parameters):
            sns.histplot(df, x=parameter, ax=ax, **kwargs)
    else:
        return NotImplementedError(f'Kind {kind} is not implemented')
    return fig, axes
```

```



class SplitVoigtSegmented():
    """
    A class for storing Lorentz fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
        ----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, left fwhm, right fwhm and integral (area) in 1st, 2nd, 3rd and 4th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm1': self.fwhm1, 'fwhm2': self.fwhm2, 'eta' : self.eta, 'integral' : self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm1(self):
        return self.data[:, 1]

    @property
    def fwhm2(self):
        return self.data[:, 2]

    @property
    def eta(self):
        return self.data[:, 3]

    @property
    def integral(self):
        return self.data[:, 4]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0],
     self.data.shape[1])), columns=['Ep', 'fwhm1', 'fwhm2', 'eta'
    , 'integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting
    functions

        Returns:
        --------
        Returns the figure and axes generated by the plotting
    functions
        """
        if parameters is None:
```

```python
            parameters = ['Ep', 'fwhm1', 'fwhm2', 'integral']
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes


class FanoSegmented():
    """
    A class for storing Fano fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a VoigtFit instance

        Parameters:
        -----------
        data: numpy.ndarray of shape (M, 4)
```

```python
            Data array with plasmon energy, fwhm, integral (area), phi and background in 1st, 2nd, 3rd, 4th and 5th position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in K
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm1': self.fwhm1, 'phi': self.phi, 'integral': self.integral, 'background' : self.background}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm1(self):
        return self.data[:, 1]


    @property
    def integral(self):
        return self.data[:, 2]

    @property
    def phi(self):
        return self.data[:, 3]


    @property
    def background(self):
        return self.data[:, 4]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0], self.data.shape[1])), columns=['Ep', 'fwhm1', 'integral', 'phi', 'background'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting functions

        Returns:
        --------
        Returns the figure and axes generated by the plotting functions
        """
        if parameters is None:
            parameters = ['Ep', 'fwhm1', 'phi', 'integral', 'background']
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes




class VolumePlasmon():
    """
    A class for storing Voigt fit parameters of a plasmon peak after segmentation
    """
    def __init__(self, data, name=None, temperature=None):
        """
        Create a Volume Plasmon instance

        Parameters:
        -----------
        data: numpy.ndarray of shape (M, 4)
            Data array with plasmon energy, fwhm, and integral (area) in 1st, 2nd and 3re position of third axis, respectively.
        name: str
            The name of the dataset
        temperature: float
            The temperature used in the experiment in C
        """
        self.data = data
        self.name = name
        self.temperature = temperature
        self._parameter_mapping = {'Ep': self.energy, 'fwhm': self.fwhm, 'integral': self.integral}

    def __repr__(self):
        return f'{self.__class__.__name__}({self.data!r}, name={self.name!r}, temperature={self.temperature!r})'

    def __str__(self):
        return f'{self.__class__.__name__} with shape {self.data.shape}:\n{self.dataframe.describe()}'

    @property
    def energy(self):
        return self.data[:, 0]

    @property
    def fwhm(self):
        return self.data[:, 1]

    @property
    def integral(self):
        return self.data[:, 2]



    @property
    def dataframe(self):
        df = pd.DataFrame(self.data.reshape((self.data.shape[0], self.data.shape[1])), columns=['Ep', 'fwhm', 'integral'])
        df.insert(0, 'T', self.temperature)
        return df

    def get_parameter(self, parameter):
        """
        Return a parameter from the fit
        """
        return self._parameter_mapping.get(parameter, None)

    def plot(self, kind, parameters, *args, **kwargs):
        """
        Plot the fit results

        Parameters:
        -----------
        kind: str
            The kind of plot. Should be either "img" or "hist"
```

```python
        parameters: str or list
            The parameter(s) to plot.
        *args: Optional arguments passed to plotting functions
        **kwargs: Optional keyword arguments passed to plotting functions

        Returns:
        ---------
        Returns the figure and axes generated by the plotting functions
        """
        if parameters is None:
            parameters = ['Ep', 'Gfwhm', 'Lfwhm', 'integral']
        else:
            if isinstance(parameters, str):
                parameters = [parameters]
            else:
                parameters = list(parameters)


        if kind == 'img':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1, sharex=True, sharey=True)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            kwargs['norm'] = kwargs.get('norm', SymLogNorm(0.01))
            for ax, parameter in zip(axes, parameters):
                ax.imshow(self.get_parameter(parameter), *args, **kwargs)
                ax.set_title(f'{parameter}')
        elif kind == 'hist':
            fig, axes = plt.subplots(ncols=len(parameters), nrows=1)
            fig.suptitle(f'{self.name}')
            if len(parameters) == 1:
                axes = list([axes])
            df = self.dataframe
            for ax, parameter in zip(axes, parameters):
                sns.histplot(df, x=parameter, ax=ax, **kwargs)
        else:
            return NotImplementedError(f'Kind {kind} is not implemented')
        return fig, axes


```

```
################### A .ipynb file used to compile and run the
    algorithm
%matplotlib qt
import numpy as np
import sys
sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
     - 2023 V r\\Master\\Data handling\\fitting')
import methods
# import voigt_methods
import splitLorentz_methods
import pickle
import pandas as pd
from Model import e_0, hbar
import experimental
import plot_class
import summation_and_deconvolve
import analyse_results
import seaborn as sns
import matplotlib.pyplot as plt
from functools import cache
from tqdm import tqdm
import MyFunc
from scipy.stats import linregress
%load_ext autoreload
%autoreload 2

########### New cell
@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
def load_data(temperatures, mask_num,approved_files):
                              # overriding the function so that
    it can be rerunned multiple times, requires input as tuple
    return summation_and_deconvolve.load_data(list(temperatures)
    ,mask_num, list(approved_files))

########### New cell
# example
import splitLorentz_methods
import lorentzian_methods
import volumeplasmon_methods
import fano_methods
# all_methods = splitLorentz_methods.all_methods[:12]

# all_methods = voigt_methods.all_methods
# all_methods = volumeplasmon_methods.all_methods
# all_methods = [all_methods[0]]
# all_methods = [splitLorentz_methods.method1000]

# all_methods = [splitLorentz_methods.methodpreSplitLorentz]
# all_methods = [splitLorentz_methods.methodpreFano        ]
# all_methods = [splitLorentz_methods.methodpreLorentz     ]
# all_methods = [splitLorentz_methods.methodpreVolume      ]
# all_methods = [splitLorentz_methods.method1000,
    splitLorentz_methods.method1001]

all_methods = splitLorentz_methods.tenseg[-4:]
all_methods[0].compile()

############ New cell    - where the algorithm is ran after a
    pixel-wise scan

delta = 0 # To be removed
narrow_linregress_interval = False
for method in all_methods:
    print(f'Starting method {method.name}')
    exclude_scans = method.part2['exclude_scans']
    _,approved_files = analyse_results.get_numpy_arrays(**method
    .part1)

    pre_segmentation_data = {}
    _[23] = _['RM']
    for T in _.keys():
        if T=='RM' :
            _[23] = _[T]
            continue
        l1 = np.array([i[0] for i in approved_files])
        l2 = np.array([i[1] for i in approved_files])
        pre_segmentation_data[int(T)] =_[T]

        pre_segmentation_data[int(T)] = _[T]
    # arrays=arrays_
    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)
    temperatures = tuple(method.part2['temperatures'])
    experimental_data, offset,scale = load_data(temperatures,
    mask_num,tuple(approved_files))
    mask_num = min(mask_num,4)
    # assert False
```

```python
    ##### Skipping linear regression for pre-segmentation data ###########
    ################## Pre-segmentation ##############
    d   = experimental_data
    val = np.array(pre_segmentation_data[23]).swapaxes(0,1).swapaxes(1,2)[:,:,:mask_num]


    sh = (val.shape[0],val.shape[1],val.shape[2])
    x = 10 + np.arange(d[23].shape[-1])*0.010    #Hardcoded

    ################################################## Running for premethods
    if method.name[:3] == 'pre':
        for T in tqdm(experimental_data.keys()):
            linreg = np.zeros(shape=sh, dtype=object)
            val = np.array(pre_segmentation_data[T]).swapaxes(0,1).swapaxes(1,2)[:,:,:mask_num]
            for i in range(sh[0]):
                for j in range(sh[1]):
                    for k in range(sh[2]):
                        if val[i,j,k,0] == 0:
                            continue
                        # linreg[i,j,k] = linregress(d[T][i,j,k],MyFunc.model_function(x, *val[i,j,k])).rvalue**2
                        if narrow_linregress_interval:
                            raise NotImplementedError('Are you sure you dont want to change this?')
                            delta = 1
                            c     = np.nanmean(val[:,:,:,0],where = (val[:,:,:,0] > 0))
                            I = ((c-delta) < x) & (x < (c+delta))
                            linreg[i,j,k] = linregress(d[T][i,j,k][I],MyFunc.model_function(x[I], *val[i,j,k])).rvalue**2
                        else:
                            curve_fitted_interval = (method.part2['energy_interval'][0] < x)  & (x < method.part2['energy_interval'][1])
                            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

            np.save(f'linregress_result\{method.name}_rvalueSquared_{T}'+narrow_linregress_interval*('narrowed'+str(delta)),linreg, allow_pickle=True )
        continue


################### Filtering out empty segments ####################
    labels = {}
    for T in temperatures:
        labels[T] = []
        for i in range(segments[T].max()+1):
            if (segments[T]==i).sum()!=0:
                labels[T].append(i)
            else:
                print(T,i)
    x = offset + np.arange(experimental_data[method.part2['temperatures']][0]].shape[-1])*scale
    new_fit = summation_and_deconvolve.sum_and_deconvolve(experimental_data,segments, pre_segmentation_data,mask_num, labels, mean_value,x, array=pre_segmentation_data, **method.part2) # Dont really know why I take in the array twice???
################ Running curve fit ##################
    popt,pcov = summation_and_deconvolve.fit_curves(new_fit,x, mean_value,labels,**method.part2)   # inclusion of only eneries within energy interval taken care of in fit_curves()

################ Linear regression analysis    ############
    d   = new_fit
    val = np.array(new_fit[23])

    # sh = (val.shape[0],val.shape[1],val.shape[2])
    x = 10 + np.arange(np.array(d[23]).shape[-1])*0.010             #Hardcoded

    for T in tqdm(experimental_data.keys()):
        linreg = np.zeros(shape=np.array(new_fit[T]).shape[0], dtype=object)
        val = np.array(new_fit[T])
        for i in range(1,len(new_fit[T])):                                 # Skipping the 0th
            en_interval = (method.part2['energy_interval'][0] < x)  & (x < method.part2['energy_interval'][1])
            linreg[i] = linregress(val[i][en_interval] ,MyFunc.model_function(x[en_interval], *popt[T][i])).rvalue**2
        # raise RunTimeError('Husk p bytte navn!')
        np.save(f'linregress_result\{method.name}_rvalueSquared_{T}'+(exclude_scans)*'_excluded_scans',linreg, allow_pickle=True )
```

```

############### Visualize a plot ############
    # plt.figure()
    # plt.plot(x,new_fit[200][3])
    # plt.plot(x,     MyFunc.model_function(x,*popt[200][3]))
    ########### Converting the results to a DataFrame to make plotting easier ############
    temp = np.array([23,80,120,160,200,240,280,320,360,400,450,500])
    class_type = method.part2['class_type']
    df = pd.DataFrame()
    df_cov = pd.DataFrame()
    data={}
    data_cov = {}
    for T in temp:
        covariance = np.diagonal(np.array(pcov[T]), axis1=1, axis2=2)
        if class_type == 'Fano':
            data[T] = plot_class.FanoSegmented(np.array(popt[T]),str(T), int(T))
            data_cov[T] = plot_class.FanoSegmented(covariance, str(T), int(T))
        if class_type == 'SplitVoigt':
            data[T] = plot_class.SplitVoigtSegmented(np.array(popt[T]),str(T), int(T))
            data_cov[T] = plot_class.SplitVoigtSegmented(covariance,str(T), int(T))
        if class_type =='DerivativeLorentzian' or class_type=='SplitLorentzian':
            data[T] = plot_class.SplitLorentzSegmented(np.array(popt[T]),str(T), int(T))
            data_cov[T] = plot_class.SplitLorentzSegmented(covariance,str(T), int(T))
        if class_type =='Voigt':                              # NOTE: Integral is not really integral as of now
            data[T] = plot_class.VoigtSegmented(np.array(popt[T]),str(T), int(T))
            data_cov[T] = plot_class.VoigtSegmented(covariance, str(T), int(T))
        if class_type =='VolumePlasmon' or class_type=='Lorentzian':
            data[T] = plot_class.VolumePlasmon(np.array(popt[T]),str(T), int(T))
            data_cov[T] = plot_class.VolumePlasmon(covariance, str(T), int(T))


```

```python
        df     = df.append(data[T].dataframe, ignore_index=True)
        df_cov = df_cov.append(data_cov[T].dataframe, ignore_index=True)
    # unfiltered_df = df.copy()


    ############ Filtering end-result ###############
    param_list = ['Ep','fwhm1','fwhm2']                # Split Lorentz
    if class_type=='Fano':
        param_list = ['Ep','fwhm1']
    if class_type == 'Voigt':
        param_list = ['Ep', 'Lfwhm']
    if class_type == 'VolumePlasmon' or class_type == 'Lorentzian':
        param_list =  ['Ep', 'fwhm']
    for param in param_list:
        for T in temp:
            q_low  = df[(df['T'] == int(T))][param].quantile(0.1)
            q_high = df[(df['T'] == int(T))][param].quantile(0.9)
            df_cov  =  df_cov[((df[param]>= q_low) & (df[param] <= q_high))    |    (df['T']!=int(T)) ]       # Has to be before the next line
            df      =  df[((df[param]>= q_low) & (df[param] <= q_high))    |    (df['T']!=int(T)) ]
            # df_cov  =  df_cov[((df[param]>= q_low) & (df[param] <= q_high))    |    (df['T']!=int(T)) ]
            # zlp = zlp[((df[param]>= q_low) & (df[param] <= q_high))    |    (df['T']!=int(T)) ]
    if  (class_type=='SplitLorentzian' ) or (class_type=='DerivativeLorentzian' ):
        df['FWHM'] = (df['fwhm1'] + df['fwhm2'])/2 # add additional column for mean FWHM      # for Lorentzian split
    elif class_type == 'Fano':
        df['FWHM'] = df['fwhm1']                                                    # for Fano
    elif class_type == 'Voigt':
        df['FWHM'] = df['Lfwhm']
    elif class_type == 'VolumePlasmon' or class_type == 'Lorentzian':
        df['FWHM'] = df['fwhm']
    else:
        raise NotImplementedError
    df['Resistivity']  = (e_0/hbar*df['Ep']**2/df['FWHM'])**-1
    df.to_pickle('dataframes_results\\'+method.name+'
    excluded_scans'*exclude_scans)                                        #
     save the method for future reference
    df_cov.to_pickle('dataframes_results\\'+method.name+'
    excluded_scans'*exclude_scans+'_cov')
          # save the method for future reference
    f =  open(f'dataframes_results\\' + method.name + '
    excluded_scans'*exclude_scans+'_part1.pkl', 'wb')
    pickle.dump(method.part1,f)
    f.close()
    f =  open(f'dataframes_results\\' + method.name + '
    excluded_scans'*exclude_scans+'_part2.pkl', 'wb')
    pickle.dump(method.part2,f)
    f.close()
############# Plot resistivity #############
    # plt.figure()
    # sns.scatterplot(df, x='T', y='Resistivity')


""" summation_and_deconvolve.py"""
import experimental
from tqdm import tqdm, trange
import numpy as np
import matplotlib.pyplot as plt
import logging

import sys
sys.path.append('c:\\Users\\krist\\OneDrive - NTNU\\Semestre\\10
     - 2023 V r\\Master\\Data handling\\fitting')
import signal_lines
import Params
import Model
# import MyFunccopy
from skimage.segmentation import slic
import utils


lowest_ev = 10.
temperatures = ['80','120','160','200','240','280','320','360','
    400','450','500']

def segment(arrays, pre_plasmon, num_params_pre_segment, **
    kwargs):
    """This function takes in the results and segments them.
    Class type is the class used pre-segmentation.

    num_params_pre_segment is the number of parameters used for the curve fitting prior to segmentation.
    """
    class_type = str(type(pre_plasmon))[21:-2]
    ###################### Getting rid of outliers ######################
    if 'temperatures' in kwargs.keys():
        temp = kwargs['temperatures']
    else:
        temp = ['80','120','160','200','240','280','320','360','400','450','500']
    if 'filter_outliers_max' in kwargs.keys():
        filter_outliers_max = kwargs['filter_outliers_max']
    else:
        filter_outliers_max = None
    if 'filter_outliers_min' in kwargs.keys():
        filter_outliers_min = kwargs['filter_outliers_min']
    else:
        filter_outliers_min = None
    if 'additional_constraints' in kwargs.keys():
        additional_constraints = kwargs['additional_constraints']
    else:
        additional_constraints = False
    if 'pixels_per_segment' in kwargs.keys():
        pixels_per_segment = kwargs['pixels_per_segment']
    else:
        pixels_per_segment = 20
    if 'num_param' in kwargs.keys():
        num_param = kwargs['num_param']
    else:
        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
    if 'compactness' in kwargs.keys():
        compactness = kwargs['compactness']
    else:
        compactness    = 1

    mask__     = arrays[temp[0]][0][:,:,0] > 10                 # To be removed
    mask_num = np.array([len(arrays[T]) for T in temp]).min()              #TODO: make the code susceptible for ragged arrays

    segments = {}

```

```python
    n_sigma   = kwargs['n_sigma']        # sigma value for including/excluding pixels.
                                    # all pixels included in the analysis. Should be same for all known values when using SplitVoigt

    mask      = np.zeros(shape=(mask__.shape[0],mask__.shape[1], mask_num), dtype=bool)       # img_sizeX, img_sizeY, number of scans
    mean_value    = {}
    sigma_value   = {}
    _             = {}
    for k,T in enumerate(temp):
        mask      = np.zeros(shape=(mask__.shape[0],mask__.shape[1],len(arrays[T])), dtype=bool)       # img_sizeX, img_sizeY, number of scans
        # print(mask_num)
        _[T] = np.array(arrays[T][:]).swapaxes(0,1).swapaxes(1,2)
        # for i in range(mask_num):
        for i in range(len(arrays[T])):
            if (int(T), i) in utils.exclude and kwargs['exclude_scans']:
                mask[:,:,i] = False
                print((T,i), 'excluded')
                continue
            if additional_constraints:
                if   class_type == 'SplitLorentzian':
                    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)
                elif class_type == 'Fano':
                    mask_    = (arrays[T][i][:,:,0] > 13 ) #& (arrays[T][i][:,:,1] > 0.3) & (arrays[T][i][:,:,1] < 0.8)
                elif class_type == 'Lorentzian':
                    mask_    = (arrays[T][i][:,:,0] > 13 ) & (arrays[T][i][:,:,1] > 0.1)
                elif class_type == 'VolumePlasmon':
                    mask_    = (arrays[T][i][:,:,0] > 13 ) & (arrays[T][i][:,:,1] > 0.1)


            else:
                mask_     = arrays[T][i][:,:,0] > 10
            mask[:,:,i] = mask_==1
            for param in range(num_params_pre_segment):
                arr = arrays[T][i][:,:,param]
                arr_ = arr[mask_]
                if (filter_outliers_min is not None) and (filter_outliers_min is not None):
                    mean_value[(T,i,param)]  = np.mean(np.sort(arr_)[filter_outliers_min:-filter_outliers_max]) # excluding extreme outliers
                    sigma_value[(T,i,param)] = np.std( np.sort(arr_)[filter_outliers_min:-filter_outliers_max]) # excluding extreme outliers
                else:
                    mean_value[(T,i,param)]  = np.nanmean(arr, where=mask_)
                    sigma_value[(T,i,param)] = np.nanstd( arr, where=mask_)
                mask[:,:,i]              = mask[:,:,i] & (np.abs((arr - mean_value[(T,i,param)]))   < n_sigma*sigma_value[(T,i,param)] )
        valid_num_pixels = mask.sum()
        n_segements         = int(valid_num_pixels/pixels_per_segment)
        segments[T]  = slic(_[T][:,:,:,:num_param], n_segments=n_segements, compactness=compactness, mask=mask, enforce_connectivity=kwargs['enforce_connectivity'], min_size_factor=0.9)#, max_size_factor=1.4)
        print(T,f'C. valid_num_pixels={valid_num_pixels}, n_segments={n_segements}; Result: n_segments={segments[T].max()}, pixels_per_segment={valid_num_pixels/segments[T].max()}')
    return segments, mean_value, sigma_value, mask_num



def load_data(temp, mask_num,approved_files):
    """Mask_num is to be removed"""
    import hyperspy.api as hs
    # mask_num = min(mask_num,4)
    T = temp
    last_index = 1000
    experimental_data = {}
    for T in tqdm(temp):
        if (T == 23) or (T=='RM'):
            lowloss  = []
            highloss = []

            for i in range(1,12):
                # print(f'Checking {(T,i)}')
                if ('RM',i) not in approved_files:
                    logging.warning(f'Skipping{(T,i)}')
                    continue
                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
                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
            # logging.warning('Note to self: tar kun med scannene fom scan 6 for romtemperatur, mulig det blir ragged da')
        else:
            lowloss = []
            highloss= []
            for i in range(1,12):
                if (str(T),i) not in approved_files:
                    logging.warning(f'Skipping{(T,i)}')
                    continue
                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)]
                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)]
        for l,h in zip(lowloss,highloss):
            l.align_zero_loss_peak(also_align=[h], show_progressbar=False, print_stats=False, signal_range=(-1.,1.))
        print(T,(highloss))
        scale   = h.axes_manager['Energy loss'].scale
        experimental_data[T] = np.array([h.isig[lowest_ev:].isig[:last_index].data  for h in highloss]).swapaxes(0,1).swapaxes(1,2)
        # logging.debug(T+ [h.isig[lowest_ev:].axes_manager['Energy loss'].offset for h in highloss])
        print(T,[h.isig[lowest_ev:].axes_manager['Energy loss'].offset for h in highloss])
        print(T,[h.isig[lowest_ev:].axes_manager['Energy loss'].scale for h in highloss])
    return experimental_data, lowest_ev,scale


def sum_and_deconvolve(experimental_data,segments, arrays, mask_num, labels, mean_value,x, array=None, class_type='SplitVoigt', **kwargs):
    ############ Summing the pixels and deconvolving #############
    from scipy.optimize import curve_fit
    import MyFunc
    import utils
    import hyperspy.api as hs
    if 'temperatures' in kwargs.keys():
        temperatures = kwargs['temperatures']
    if 'subtract_center' in kwargs.keys():
        subtract_center = kwargs['subtract_center']
        assert array is not None
    else:
        subtract_center = False
    known_params = {}
    offset = x[0]
    scale = (x[-1]-x[0])/x.shape[0]
    # class_type = str(type(plasmon))[21:-2]
    if class_type=='SplitVoigt':
        known_params = {'center': 0,
                        'fwhm1' : 1,
                        'fwhm2' : 2,
                        'frac'  : 3,
                        'area'  : 4}
    if class_type=='Fano':
        known_params = {'center': 0,
                        'fwhm' : 1,
                        'area' : 2,
                        'phi'  : 3,
                        'background'  : 4}
    if 'FWHM' in kwargs.keys():
        FWHM = kwargs['FWHM']                                 #
   eV
    else:
        FWHM = 0.08
    deconv_func = signal_lines.gaussian((np.arange(x.shape[0])-x
   .shape[0]/2)*scale,0,FWHM,1) #Keep it centered
    modifier = kwargs['modify']
    new_fit = {}
    for T in temperatures:
        # print(T)
        if T==23:
            zlp = hs.load(r'Cambridge_7\S1_RMTemp _ 0.01
   s_58nmpixelSTEM SI'+str(1)+'.dm4')[2]
        else:
            zlp = hs.load(r'Cambridge_7\S1_'+str(T)+'CTemp _
   0.01s_58nmpixel'+str(1)+'.dm4')[2]
        zlp = np.roll(zlp.data[0,0,:x.shape[0]],300)
                                    # centerig the ZLP prior to
   deconvolution, but hardcoded, taken from vacuum
```

```
        new_fit[T] = []
        arr = np.array(arrays[int(T)]).swapaxes(0,1).swapaxes(1,2)[:,:,:mask_num]
        for i in range(0,segments[int(T)].max()+1):
            if subtract_center:                                                    # Not used for end-results
                y = []
                for j in range(experimental_data[T].shape[0]):
                    for k in range(experimental_data[T].shape[1]):
                        for l in range(experimental_data[T].shape[2]):
                            if segments[T][j,k,l] != i:
                                continue
                            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
                            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

                y = np.array(y).mean(axis=0)

            else:
                y = experimental_data[T][segments[T]==i].mean(axis=0).copy()                     # Take the mean over the pixels
                y -= np.median(np.roll(y,200)[200:2*200])                                                       # Hardcoded. Median is taken for 10<E<12
            if modifier == 'fourier_ratio':                                        # deconvolve the summed regions
                s = np.fft.ifft(utils.fourier_ratio(y,zlp,deconv_func))
            elif modifier == 'zero_loss':
                y += zlp
                s = np.fft.ifft(np.fft.fft(zlp)        *np.log(np.fft.fft(y)/np.fft.fft(zlp)))
            elif modifier == 'gaussian':
                y += zlp
                s = np.fft.ifft(np.fft.fft(deconv_func)*np.log(np.fft.fft(y)/np.fft.fft(zlp)))
            else:
                s = y
            new_fit[T].append(s.real)
    return new_fit
```

```python


def fit_curves(new_fit,x_, mean_value,labels, class_type='SplitVoigt',**kwargs):
    ########### running the curve fit again ##############
    from scipy.optimize import curve_fit
    import MyFunc
    from tqdm import tqdm
    if class_type=='SplitVoigt':
        known_params = {'center': 0,
                        'fwhm1' : 1,
                        'fwhm2' : 2,
                        'frac'  : 3,
                        'area'  : 4}
        bounds     = ([13.8,0,0,0,0],[16,3,3,1,1e5])
        init_guess = [15,0.5,0.5,1,1e3]
    if class_type=='Fano':
        known_params = {'center': 0,
                        'fwhm'  : 1,
                        'area'  : 2,
                        'phi'   : 3,
                        'background'   : 4}
        bounds     = ([14,0,0,-2*np.pi,0],[16,3,1e6,2*np.pi,1])
        init_guess = [15,0.2,2e2,1,0.5]

    if (class_type=='SplitLorentzian') or (class_type=='DerivativeLorentzian'):
        known_params = {'center': 0,
                    'fwhm1' : 1,
                    'fwhm2' : 2,
                    'area'  : 3}
        bounds     = ([13.8,0,0,0],[16,3,3,1e5])
        init_guess = [15,0.5,0.5,1e3]
        if class_type=='DerivativeLorentzian':
            init_guess = [15,0.5,0.5,1e1]
    if class_type=='Voigt':
        known_params = {'center': 0,
                    'Gfwhm' : 1,
                    'Lwhm' : 2,
                    'area'  : 3}
        bounds     = ([13.8,0,0,0],[16,3,3,1e5])
        init_guess = [15,0.1,0.5,1e3]
    if (class_type=='VolumePlasmon') or (class_type=='Lorentzian'):
        known_params = {'center': 0,
                    'fwhm' : 1,
                    'area'  : 2}
        bounds      = ([13.8,0,0],[16,3,1e5])
        init_guess = [15,0.5,1e3]

    weight = kwargs['weighted']
    popt = {}
    pcov = {}
    for T in tqdm(new_fit.keys()):
        popt[T] = []
        pcov[T] = []
        for i,y_ in enumerate(new_fit[T]):

            try:
                if i not in labels[T] or i ==0:             # dont include the 0th as it is all the pixels that were excluded from the scans
                    raise RuntimeError

                if kwargs['num_bin'] != -1:
                    y   = np.histogram(x_,weights=y_, bins = kwargs['num_bin'])[0]
                    x   = np.histogram(x_,weights=y_, bins = kwargs['num_bin'])[1][:-1]

                else:
                    y = y_.copy()
                    x = x_.copy()

                en_interval = (kwargs['energy_interval'][0] < x) & (x < kwargs['energy_interval'][1])
                x = x[en_interval]                                  # Not considering what is outside of the pre-defined energy interval
                y = y[en_interval]                                  # Not considering what is outside of the pre-defined energy interval
                if True:
                    # print(kwargs['derivative'])
                    if kwargs['derivative'] or class_type=='DerivativeLorentzian': # TODO: fix all weighting specifications
                        if class_type != 'DerivativeLorentzian':
                            raise NotImplementedError
                        y = utils.derivative(y,x)
                        if weight:
                            logging.warning('Taking the weighted fit of a derivative.')
                    sigma = 1/y**2*weight+1*(not weight)
                    popt_, pcov_ = curve_fit(MyFunc.model_function, x,y, p0=init_guess, bounds=bounds, sigma=sigma, absolute_sigma=True)
                if kwargs['temp_adjusted_interval']:
                    if class_type != 'SplitLorentzian':
                        raise NotImplementedError('Only available for SplitLorentzian')
                    en_interval = ((popt_[0]-0.5)< x) & (x < popt_[0])
                    bounds_     = ([14.4,0,0,0], [popt_[0],1,1,1e5])
                    init_guess_ = [popt_[0],0.1,0.1,1e3]
                    x = x[en_interval]
                    y = y[en_interval]
                    popt_, pcov_ = curve_fit(MyFunc.model_function, x,y, p0=init_guess_, bounds=bounds_, sigma=sigma[en_interval], absolute_sigma=True)
                popt[T].append(popt_)
                pcov[T].append(pcov_)
            except RuntimeError:
                popt[T].append(np.array([np.nan]*len(known_params.keys())))
                pcov[T].append(np.ones(   (len(known_params.keys()),len(known_params.keys()))   ) *np.nan)
                print(T,i,'Runtimeerror, adding nans')
                continue
    return (popt, pcov)
```