

Name: Marius Pfeiffer

Matrikel-Nr.: 4188573

E-Mail: marius.pfeiffer@stud.uni-heidelberg.de

Betreut durch: Valentin Krems

17.02.2025

## Versuch 256: Röntgenfluoreszenz



Abbildung 1: Versuchsaufbau

## Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>2</b>
1.1 Physikalische Grundlagen . . . . .	2
1.2 Versuchsdurchführung . . . . .	4
<b>2 Messprotokoll</b>	<b>5</b>
<b>3 Auswertung</b>	<b>6</b>
<b>4 Zusammenfassung und Diskussion</b>	<b>12</b>

# 1 Einleitung

Nachdem wir in Versuch 255 das Röntgenspektrum genauer untersucht hatten, setzen wir uns in Versuch 256 mit der sogenannten Röntgenfluoreszenz auseinander. Röntgenfluoreszenz beschreibt die Abstrahlung einer sekundären Röntgenstrahlung, bei der Interaktion von Röntgenstrahlung mit Materie.

## 1.1 Physikalische Grundlagen

Trifft Röntgenstrahlung auf Atome von Materie, kann diese Elektronen aus den inneren Schalen der Atome herauslösen. Rücken daraufhin Elektronen von den äußeren Schalen in die dadurch entstandenen Fehlstellen nach, so wird die dabei frei werdende Energie in Form einer sekundären Röntgenstrahlung abgestrahlt. Für die frei werdende Energie bei einem Elektronenübergang von der Schale  $n_2$  zur Schale  $n_1$  gilt, angenähert aus dem Bohr'schen Atommodell,

$$\Delta E = E_2 - E_1 = chR_\infty \left( \frac{(Z - \sigma_{n1})^2}{n_1^2} - \frac{(Z - \sigma_{n2})^2}{n_2^2} \right). \quad (1)$$

In diese Rechnung gehen neben der Lichtgeschwindigkeit  $h$ , dem Planck'schen Wirkungsquantum  $h$  und der Rydberg-Konstante  $R_\infty$  auch die Kernladungszahl  $Z$  und die Abschirmkonstanten  $\sigma_i$  ein, welche beide Materialabhängig sind. Daher ist die Röntgenfluoreszenz charakteristisch für die bestrahlte Probe. Mit einer mittleren Abschirmkonstante  $\sigma_{12}$ , welche die  $\sigma_i$  ersetzt und der Rydberg-Energie  $E_R = chR_\infty$  ( $\approx 13.6\text{eV}$ ) können wir die obige Gleichung umschreiben zu

$$\Delta E = E_2 - E_1 = E_R(Z - \sigma_{12})^2 \left( \frac{1}{n_1^2} - \frac{1}{n_2^2} \right). \quad (2)$$

Betrachten wir speziell die  $K_\alpha$ -Strahlung der Übergänge  $n_2 = 2 \rightarrow n_1 = 1$ , lässt sich die Abschirmkonstante für nicht zu schwere Kerne mit einer Ladungszahl bis etwa  $Z \approx 30$  mit  $\sigma_{12} \approx 1$  annähern. Damit können wir die Gleichung weiter vereinfachen zu

$$\sqrt{\frac{E}{E_R}} = (Z - 1) \sqrt{\frac{3}{4}}. \quad (3)$$

### Röntgenenergiedetektor

Im Versuch werden wir einen Halbleiterdetektor verwenden, um die Energie der Fluoreszenzstrahlung zu messen. Ein solcher Detektor ist aufgebaut wie eine in Sperrrichtung betriebene Diode, zu sehen in Abbildung (2). Hierbei werden ein n-Halbleiter, also ein Halbleiter mit einer erhöhten Anzahl an beweglichen negativen Ladungsträgern und ein p-Halbleiter mit einer erhöhten Anzahl an Fehlstellen in Kontakt gebracht. Aufgrund verschiedener Ladungsträgerdichten in den beiden Halbleitern kommt es an der Kontaktstelle zur Diffusion. Hierbei können Elektronen vom n-Halbleiter in den p-Halbleiter wandern und dort eine Fehlstelle besetzen. Es bildet sich die sogenannte Verarmungszone, in welcher keine freien Ladungsträger vorhanden sind. Durch die Ladungsträgerverschiebung baut sich ein elektrisches Feld auf, welches der Diffusion entgegenwirkt. Somit kann die Verarmungszone nicht beliebig groß werden. Durch eine von außen anliegende Spannung, welche umgekehrt zur Dotierung des Halbleiters gepolt ist, werden die freien Ladungsträger zusätzlich nach außen abgesaugt, wodurch sich die Verarmungszone weiter vergrößert.

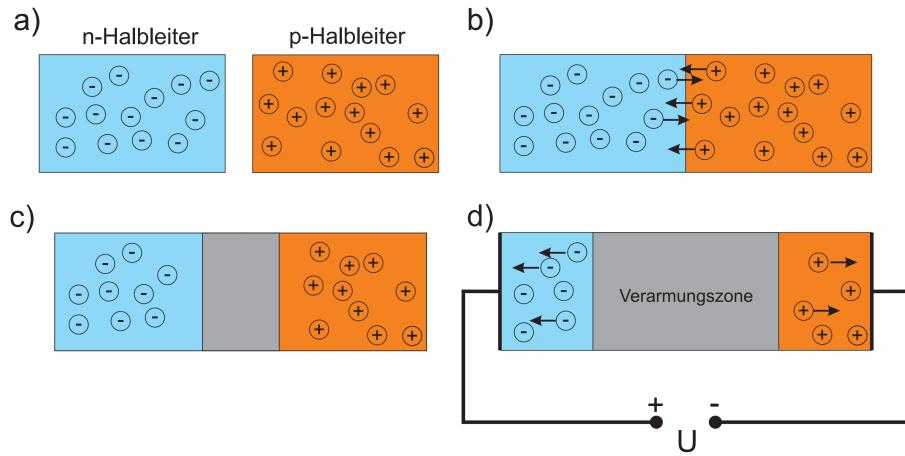


Abbildung 2: Entstehung der Verarmungszone in einem pn-Halbleiter.

Wenn ein Röntgenphoton auf die Verarmungszone trifft, wird dieses aufgrund des Photoeffekts absorbiert, wobei ein Photoelektron ausgesendet wird. Durch Stöße mit den Atomen der Halbleiter in der Verarmungszone werden Elektron-Loch-Paare zeugt, welche durch die anliegende Spannung abgesaugt werden. Dabei ist die Zahl der freigesetzten Ladungen proportional zur Energie des eingefallenen Röntgenphotons. Die ausgehenden Ladungsimpulse werden durch einen Integrator verstärkt und gemessen. Mit einem Vielkanalanalysator werden die verschiedenen Impulshöhen in Kanäle unterteilt und entsprechend der Häufigkeit, in der sie auftreten, in ein Histogramm zur Auswertung kategorisiert.

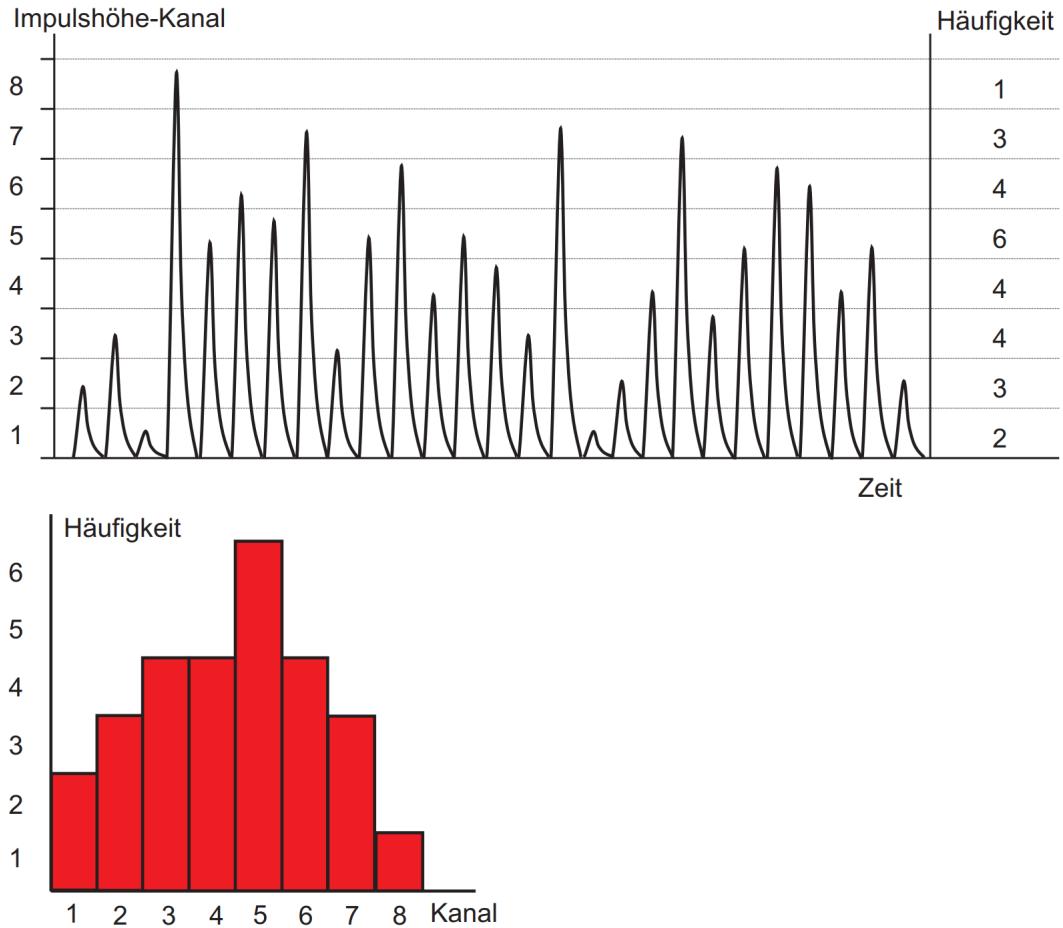


Abbildung 3: Funktionsprinzip des Vielkanalanalysators.

Entsprechend dem Energiespektrum eines bekannten Elements, in diesem Versuch sind das Eisen und Molybdän, kann die Umrechnung von der Kanalnummer zur entsprechenden Energie kalibriert werden.

## 1.2 Versuchsdurchführung

**Aufzeichnung der Impulse.** Im Röntgengerät ist der Targethalter in einem Winkel von  $45^\circ$  zur Strahlebene angebracht. Auf diesem positionieren wir nacheinander die verschiedenen Probenplättchen. Die Röntgenröhre wird mit einer Spannung von 35kV und einem Strom von 1mA betrieben. Wir zeichnen die Impulse über 512 Kanäle über eine Messzeit von 180s pro Probe auf. Die Proben umfassen acht verschiedene reine Metalle und fünf verschiedene Legierungen.

**Energiekalibrierung.** Wir bestimmen zunächst die in der Analysesoftware im gemessenen Eisen- und Molybdänspektrum jeweils den Impulskanal des  $K_\alpha$ -Peaks. Wir kalibrieren die Energieskala, indem wir dann die bestimmten Impulskanäle mit den entsprechenden bekannten Energiewerten der  $K_\alpha$ -Strahlung der beiden jeweiligen Elemente assoziieren.

**Bestimmung der Lage der  $K_\alpha$ - und  $K_\beta$ -Peaks.** Anhand der nun kalibrierten Skala ermitteln wir Peakschwerpunkt ( $\mu$ ) und Peakbreite ( $\sigma$ ) der  $K_\alpha$ - und  $K_\beta$ -Peaks für alle aufgezeichneten Spektren der reinen Metalle.

**Untersuchung der Legierungen.** Anhand eines in der Analysesoftware integrierten Periodensystems, lassen sich die jeweiligen Energien in den aufgezeichneten Spektren ein- und ausblenden. Über die Zuordnung der dieser Energien zu den Peaks in den aufgezeichneten Spektren analysieren wir die Zusammensetzung der Legierungen.

## 2 Messprotokoll

Messprotokoll 256 Marius Pfeiffer  
Robert Grossch 17.02.2025

Reihenfolge Spektren

#	Element	Farbe	$\mu_{\alpha}$		$\mu_{\beta}$	
			$\mu$	$\sigma$	$\mu$	$\sigma$
1	Molybdän	schwarz	17.46	0.18	19.57	0.17
2	Eisen	rot	6.38	0.17	7.03	0.42
3	Nickel	blau	7.46	0.18	8.26	0.21
4	Zink	lila	8.64	0.18	9.59	0.16
5	Zirconium	cyan	15.78	0.18	17.67	0.19
6	Titan	ultramarin	4.44	0.19	4.44	0.19
7	Kupfer	pink	8.04	0.17	8.91	0.14
8	Silber	rostbraun	21.89	0.21	24.59	0.18

Kalibrierung

$$\text{Fe} : \mu = 111, \sigma = 3, 6.40 \text{ keV}$$

$$\text{Mo} : \mu = 279, \sigma = 3, 17.46 \text{ keV}$$

Lesierung

Probe 1, schwarz Fe, Cr

Probe 2, schwarz Cu, Zn

Probe 3, rot Cu, Zn

Probe 4, blau Fe, Ni

Probe 5, lila Cu, Ga, Ge

U.K

### 3 Auswertung

Tabelle (1) zeigt noch einmal zusammengefasst die durch die Analysesoftware ermittelten Energiewerte der  $K_{\alpha}$ - und  $K_{\beta}$ -Linien der untersuchten Elemente. Die Linien sind auch in Abbildung (4) in den jeweils zugeordneten Elementsymbolen beziehungsweise Farben zu finden. Zur Kalibrierung der Energieskala assoziierten wir die Energie der  $K_{\alpha}$ -Linie von Eisen von 6.40keV mit dem Kanal 111 ± 3 und die Energie 17.48keV der  $K_{\alpha}$ -Linie von Molybdän mit dem Kanal 279 ± 3.

#	Element	Farbe	$K_{\alpha}$ [keV]		$K_{\beta}$ [keV]	
			$E_{\alpha}$	$\Delta E_{\alpha}$	$E_{\beta}$	$\Delta E_{\beta}$
1	Molybdän	schwarz	17.46	0.18	19.57	0.17
2	Eisen	rot	6.38	0.17	7.03	0.42
3	Nickel	blau	7.46	0.18	8.26	0.21
4	Zink	lila	8.64	0.18	9.59	0.16
5	Zirconium	cyan	15.78	0.18	17.67	0.19
6	Titan	ultramarin	4.44	0.19	4.44	0.19
7	Kupfer	pink	8.04	0.17	8.91	0.14
8	Silber	rostbraun	21.89	0.21	24.59	0.18

Tabelle 1: Energien der  $K_{\alpha}$ - und  $K_{\beta}$ -Linien der untersuchten Elemente.

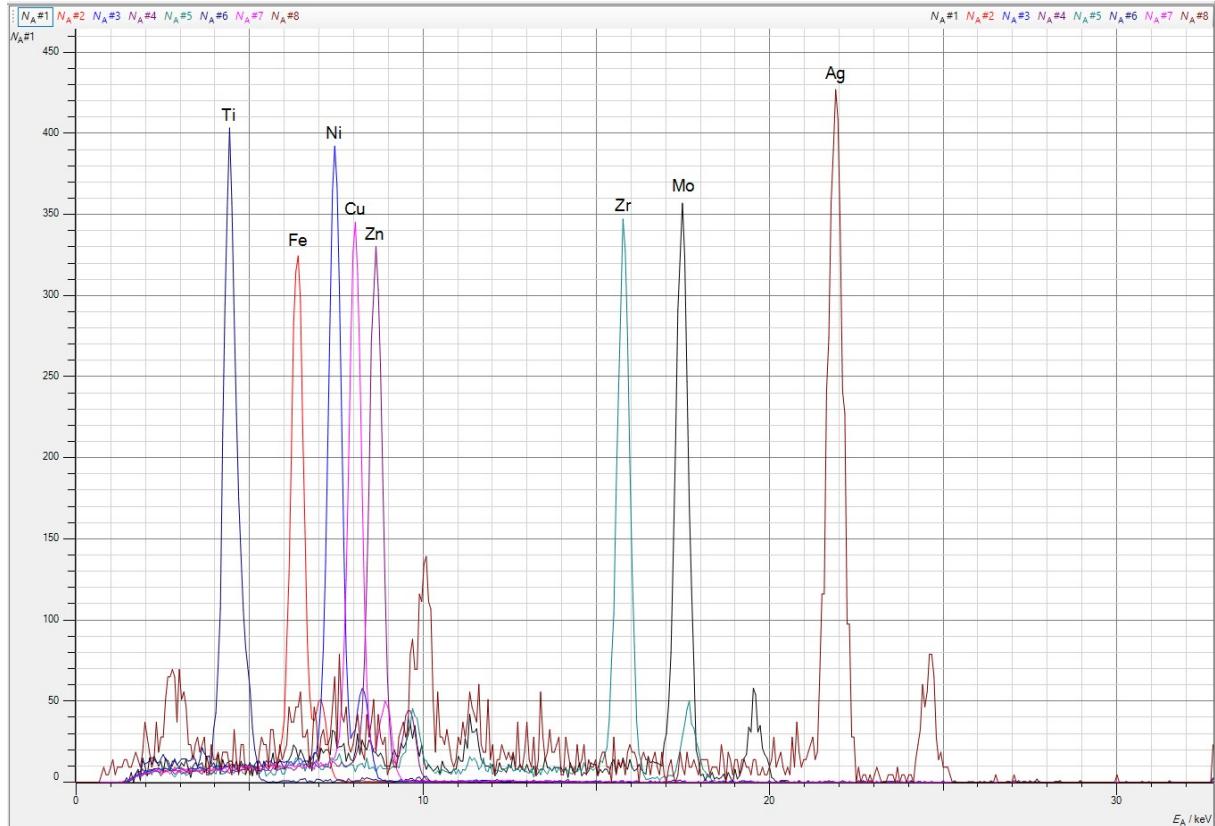


Abbildung 4: Röntgenfluoreszenzspektren der untersuchten Elemente. Zu sehen sind die Häufigkeiten, zu welcher die Energien gemessen wurden.

Wir möchten nun die gemessenen Energiewerte je für die  $K_\alpha$ - und die  $K_\beta$ -Linien der Elemente als Funktion der Kernladungszahl betrachten und an diese die Funktion

$$\sqrt{E_{\alpha(\beta)}} = \sqrt{E_R}(Z - \sigma_{n_1, n_2}) \sqrt{\left(\frac{1}{n_1^2} - \frac{1}{n_2^2}\right)} = f(Z; \sqrt{E_R}, \sigma_{n_1, n_2}) \quad (4)$$

mit den Parametern  $\sqrt{E_R}$  und  $\sigma_{n_1, n_2}$  anpassen. Hierzu berechnen wir zunächst die Wurzeln der Energiewerte aus Tabelle (1) nach der Formel

$$\sqrt{E_{\alpha(\beta)}} \pm \Delta \sqrt{E_{\alpha(\beta)}} = \sqrt{E_{\alpha(\beta)}} \pm \frac{1}{\sqrt{E_{\alpha(\beta)}}} \Delta E_{\alpha(\beta)}. \quad (5)$$

Wir betrachten zunächst die  $K_\alpha$ -Linie. Diese entspricht dem Übergang  $2 \rightarrow 1$ , somit gilt  $n_2 = 2$ ,  $n_1 = 1$  und wir suchen  $\sigma_{12}$ , sowie  $\sqrt{E_R}$ . Abbildung (5) zeigt die Wurzeln ermittelten Energiewerte mit Fehlern aufgetragen über der Kernladungszahl der Elemente, sowie die optimierte Funktion (4) mit den entsprechenden Parametern.

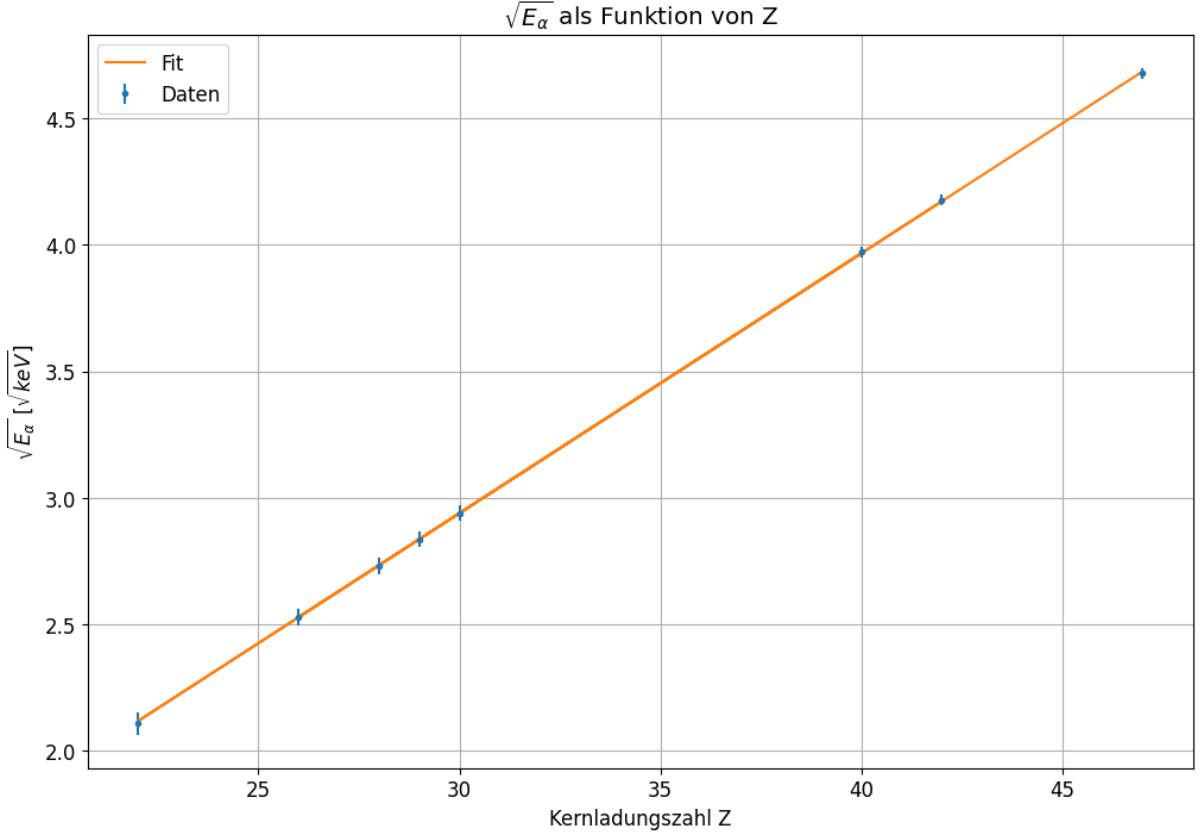


Abbildung 5:  $\sqrt{E_\alpha}$  als Funktion der Kernladungszahl  $Z$  der untersuchten Elemente mit Fit der Funktion (4).

Die optimierten Werte der Parameter lauten

$$\sqrt{E_R} = (0.1188 \pm 0.0015)\text{keV}, \quad (6)$$

$$\sigma_{12} = 1.4 \pm 0.5. \quad (7)$$

Wir quadrieren  $\sqrt{E_R}$  und bestimmen so einen ersten Wert für die Rydberg-Energie von

$$E_R = (14.1 \pm 0.4)\text{eV}. \quad (8)$$

Die gleiche Prozedur führen wir nun noch einmal für die Energien der  $K_\beta$ -Linien durch. Hierbei betrachten wir Übergang  $3 \rightarrow 1$ , es gilt also  $n_2 = 3$ ,  $n_1 = 1$  und wir suchen  $\sigma_{13}$ , sowie erneut  $\sqrt{E_R}$ . Abbildung (6) zeigt erneut die Wurzeln ermittelten Energiewerte mit Fehlern über der Kernladungszahl, sowie die optimierte Funktion (4) mit den entsprechenden Parametern.

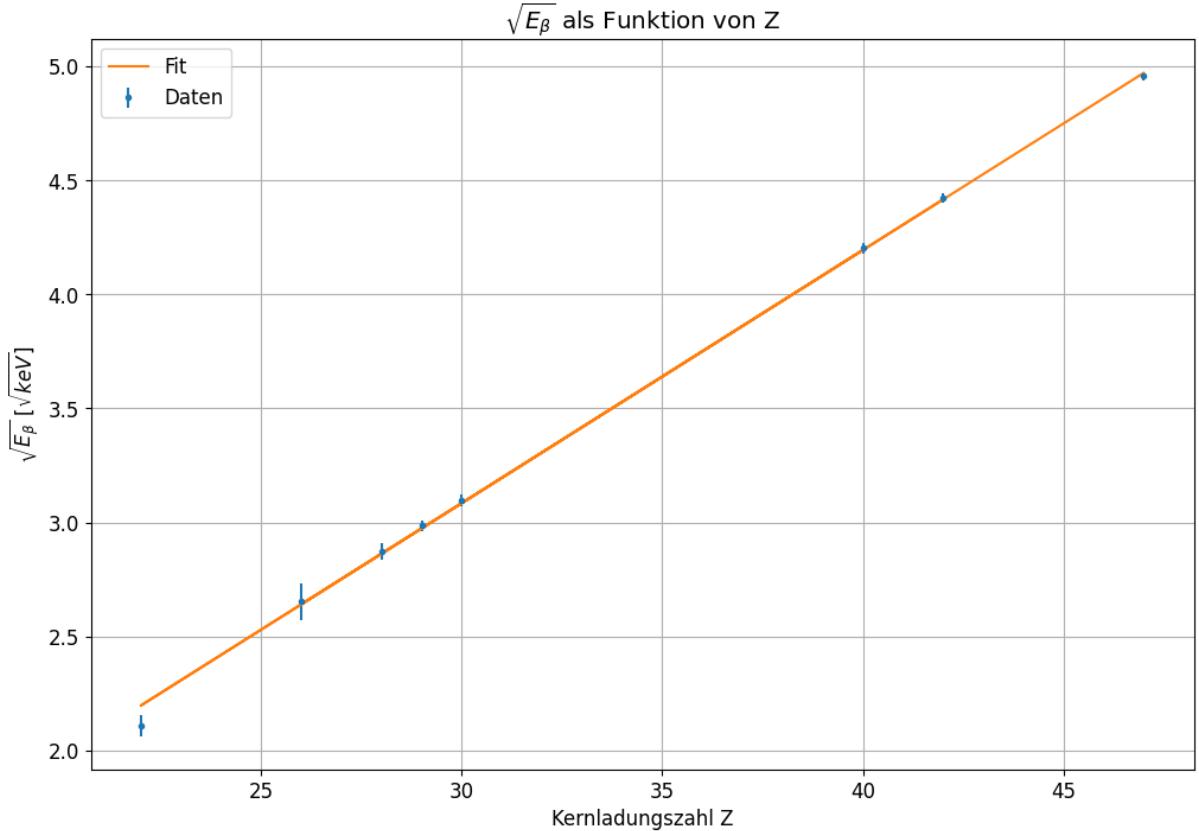


Abbildung 6:  $\sqrt{E_\beta}$  als Funktion der Kernladungszahl  $Z$  der untersuchten Elemente mit Fit der Funktion (4).

Aus dem Fit erhalten wir hierbei die optimierten Parameter

$$\sqrt{E_R} = (0.1178 \pm 0.0013)\text{keV}, \quad (9)$$

$$\sigma_{13} = 2.2 \pm 0.4. \quad (10)$$

Wir quadrieren erneut  $\sqrt{E_R}$  und erhalten so einen weiteren Wert für die Rydberg-Energie von

$$E_R = (13.87 \pm 0.21)\text{eV}. \quad (11)$$

Für den zweiten Versuchsteil haben wir die Röntgenfluoreszenzspektren verschiedener Metalllegierungen aufgezeichnet. Anhand der in den Spektren sichtbaren  $K_{\alpha}$ - und  $K_{\beta}$ -Peaks haben wir diesen die entsprechenden reinen Metalle zugeordnet, welcher in der Legierung enthalten sind. Die Spektren sind in den folgenden Abbildungen (7) bis (11) zu sehen.

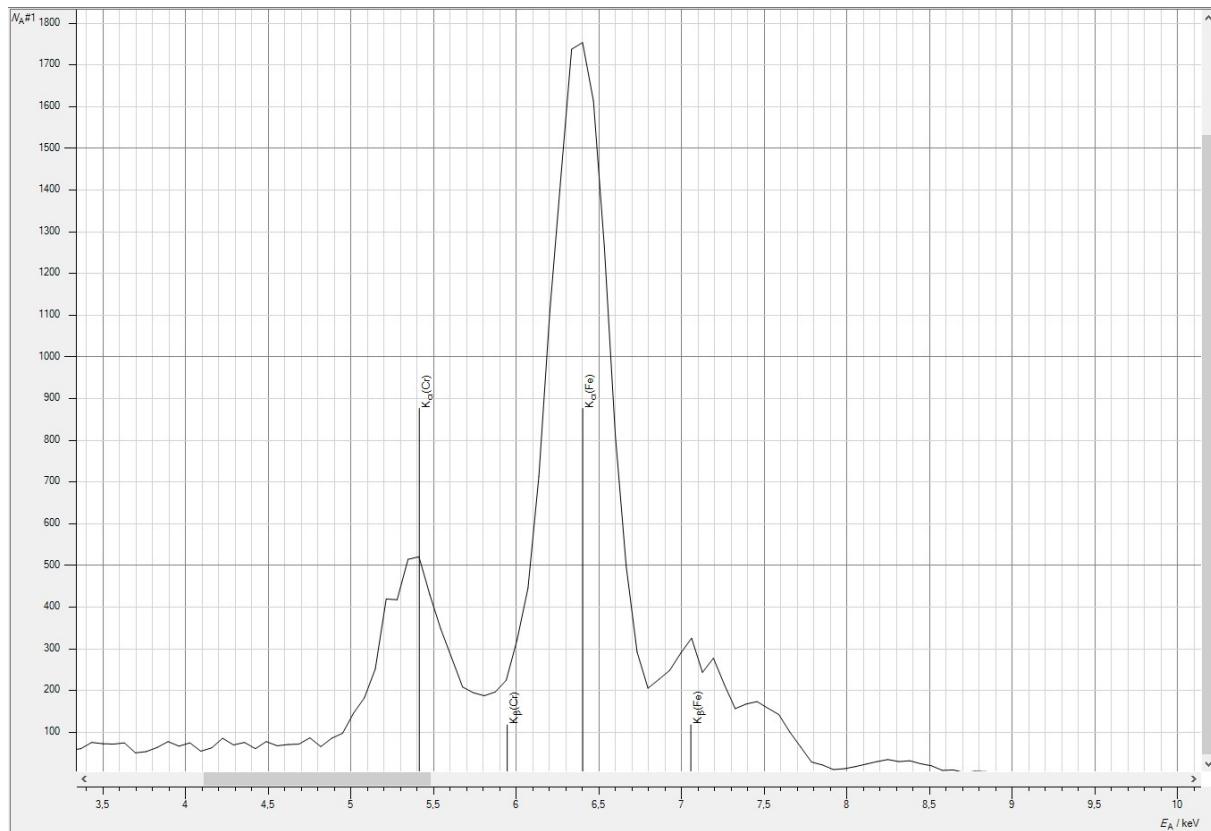


Abbildung 7: Probe 1, Legierung: Eisen, Chrom = Ferrochrom(?)

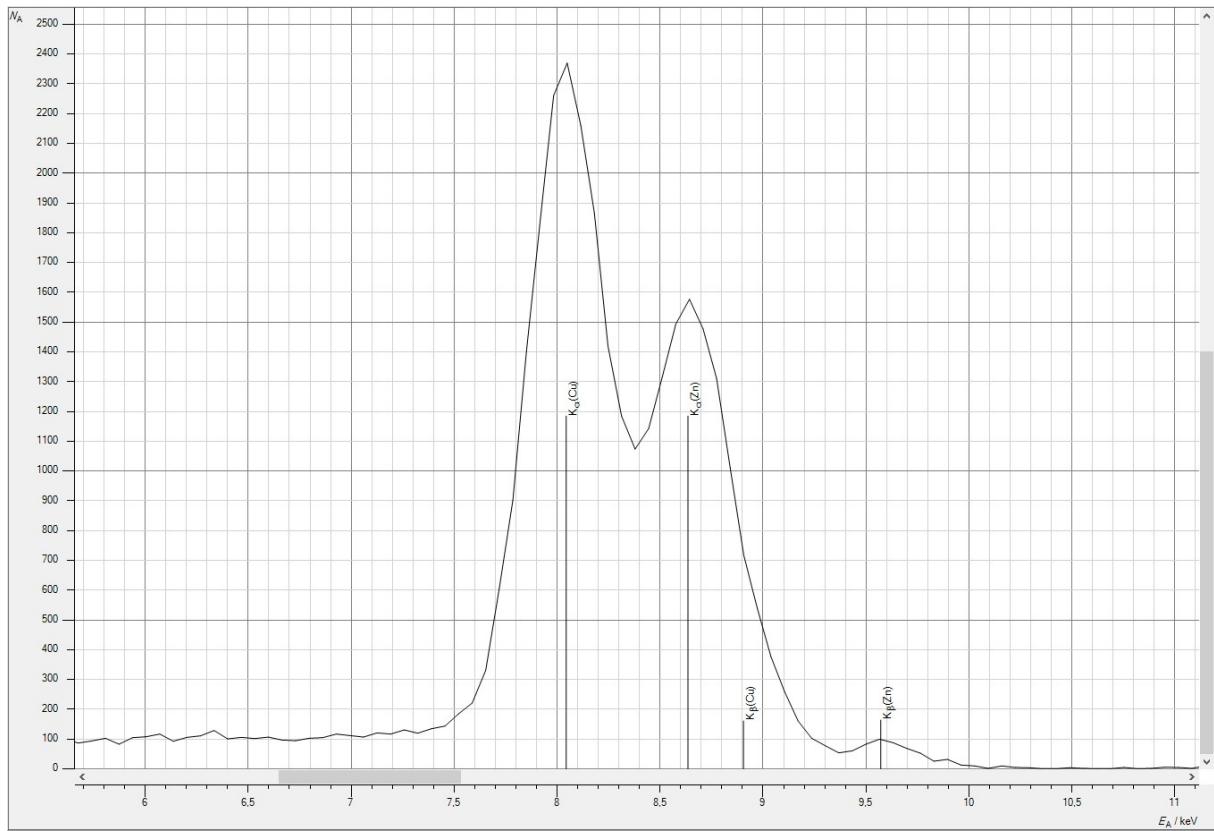


Abbildung 8: Probe 2, Legierung: Kupfer, Zink = Messing

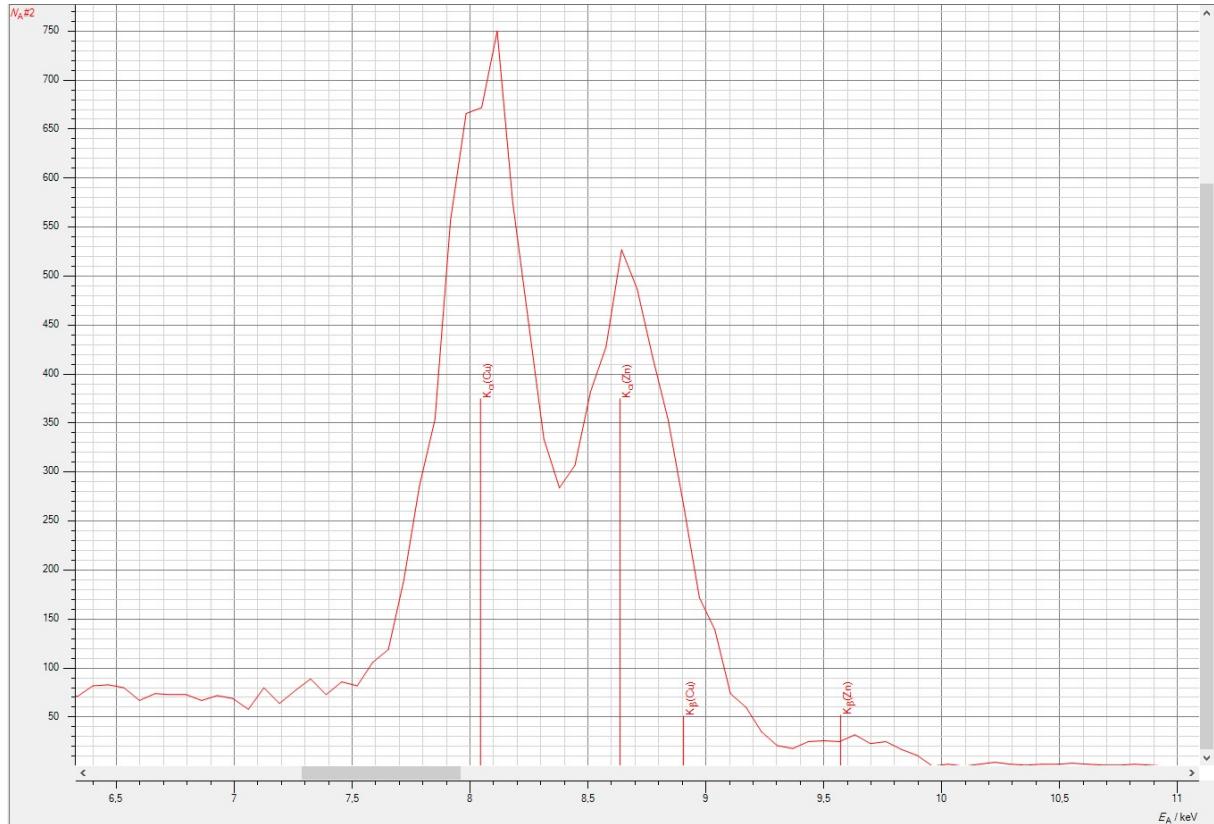


Abbildung 9: Probe 3, Legierung: Kupfer, Zink = Messing

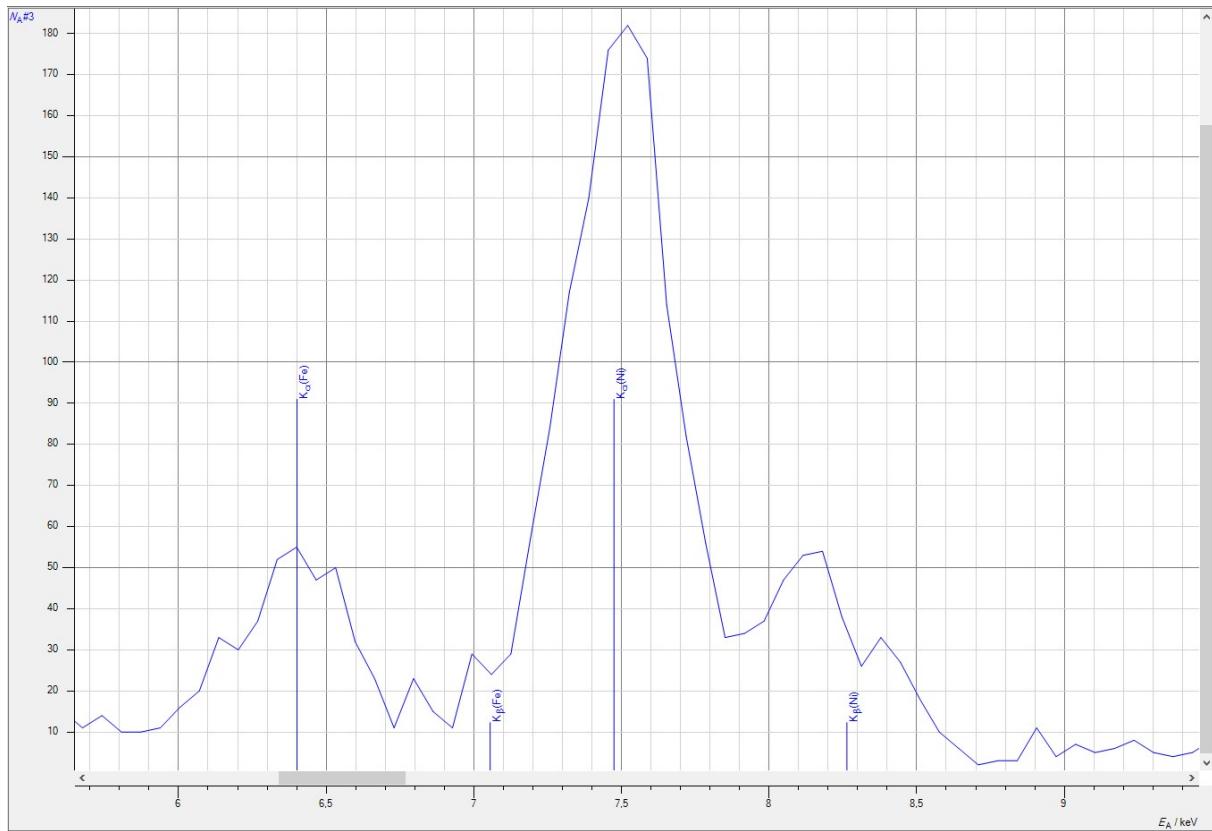


Abbildung 10: Probe 4, Legierung: Eisen, Nickel = Invar(?)

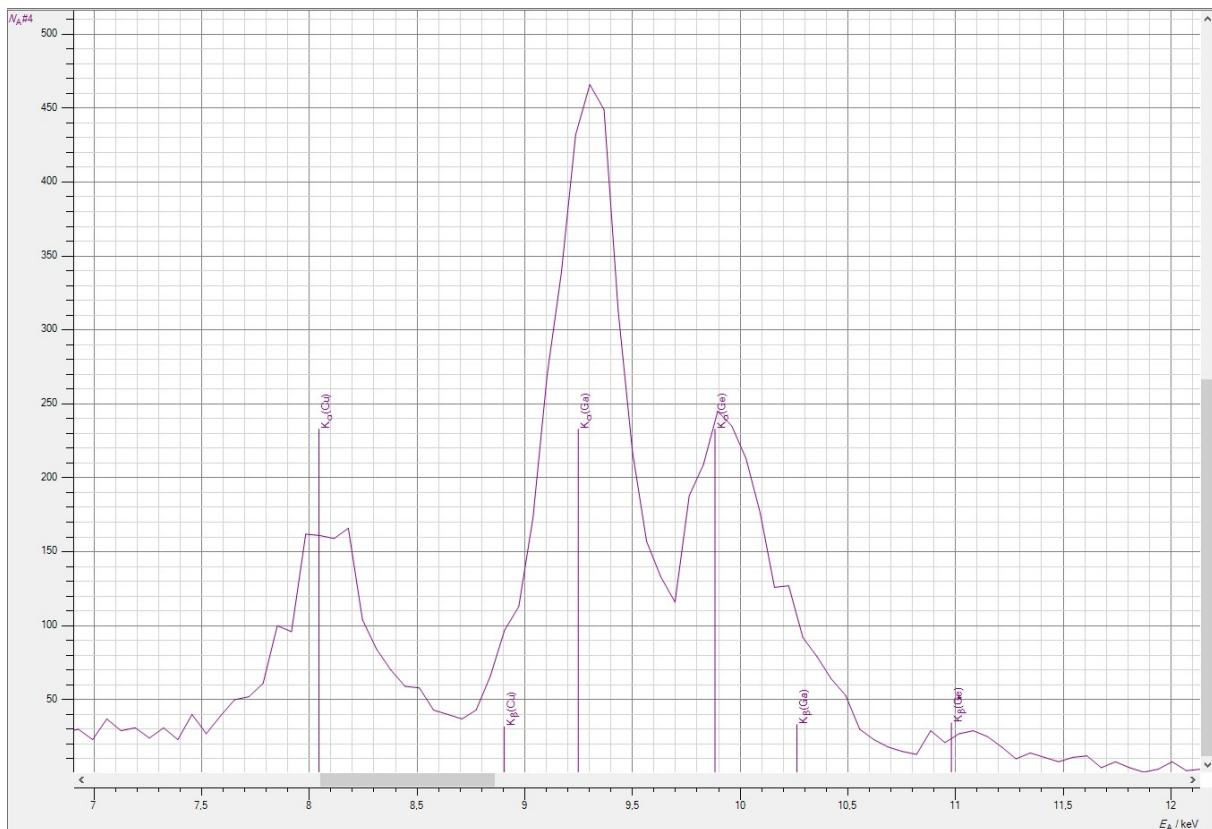


Abbildung 11: Probe 5, Legierung: Kupfer, Gallium, Germanium

## 4 Zusammenfassung und Diskussion

In Versuch 256 untersuchten wir das Phänomen der Röntgenfluoreszenz. Dies ist eine sekundäre Röntgenstrahlung welche abgestrahlt wird, wenn durch einfallende Röntgenstrahlung in einem Material Elektronenübergänge hervorgerufen werden. Das abgestrahlte Spektrum zeigt die aus dem vorherigen Versuch bekannten  $K_\alpha$ - und  $K_\beta$ -Linien auf, welche charakteristisch für das bestrahlte Material sind. Angenähert folgt die bei einem Elektronenübergang auf den Schalen  $n_2 \rightarrow n_1$  abgestrahlte Energie dem Moseleyschen Gesetz

$$\sqrt{E} = \sqrt{E_R}(Z - \sigma_{n_1, n_2})\left(\frac{1}{n_1^2} - \frac{1}{n_2^2}\right). \quad (12)$$

Im ersten Versuchsteil bestimmten wir die Rydberg-Energie  $E_R$ , sowie die Abschirmkonstante  $\sigma_{12}$  für Übergänge  $2 \rightarrow 1$ , also  $K_\alpha$ -Übergänge, und  $\sigma_{13}$  für Übergänge  $3 \rightarrow 1$ , also  $K_\beta$ -Übergänge. Hierzu nahmen wir zunächst die Röntgenfluoreszenzspektren verschiedener reiner Metalle auf. Durch die Analysesoftware wurden die gemessenen abgestrahlten Energieimpulse entsprechend ihrer Stärke in Kanälen kategorisiert. Um damit rechnen zu können, mussten wir die Energieskala kalibrieren, indem wir die bekannten Energien der  $K_\alpha$ -Übergänge von Eisen und Molybdän mit den entsprechenden Kanälen, in welchen wir die Peaks fanden assoziieren. In Tabelle (2) sind die ermittelten Energiewerte der Linien zusammen mit den Literaturwerten<sup>1</sup> zum Vergleich zu finden.

#	Element	$E_\alpha$ [keV]	$E_{\alpha,\text{Lit}}$ [keV]	Abw.	$E_\beta$ [keV]	$E_{\beta,\text{Lit}}$ [keV]	Abw.
1	Mo	$17.46 \pm 0.18$	—	—	$19.57 \pm 0.17$	19.61	$0.24\sigma$
2	Fe	$6.38 \pm 0.17$	—	—	$7.03 \pm 0.42$	7.06	$0.08\sigma$
3	Ni	$7.46 \pm 0.18$	7.48	$0.12\sigma$	$8.26 \pm 0.21$	8.27	$0.05\sigma$
4	Zn	$8.64 \pm 0.18$	8.64	0	$9.59 \pm 0.16$	9.57	$0.13\sigma$
5	Zr	$15.78 \pm 0.18$	15.77	$0.06\sigma$	$17.67 \pm 0.19$	17.67	0
6	Ti	$4.44 \pm 0.19$	4.51	$0.37\sigma$	$4.44 \pm 0.19$	4.93	$2.58\sigma$
7	Cu	$8.04 \pm 0.17$	8.05	$0.06\sigma$	$8.91 \pm 0.14$	8.91	0
8	Ag	$21.89 \pm 0.21$	22.16	$1.29\sigma$	$24.59 \pm 0.18$	24.94	$1.95\sigma$

Tabelle 2: Energien der  $K_\alpha$ - und  $K_\beta$ -Linien der untersuchten Elemente und Vergleich mit den entsprechenden Literaturwerten.

Nacheinander trugen wir einem die  $K_\alpha$ -Energien und einmal die  $K_\beta$ -Energien über der Kernladungszahl  $Z$  der Elemente in einem Diagramm auf und passten die oben genannte Funktion an diese Daten an. Aus der Anpassung an die  $K_\alpha$ -Energie ermittelten wir so die Werte

$$\sigma_{12} = 1.4 \pm 0.5 \quad (13)$$

und

$$E_R = (14.1 \pm 0.4)\text{eV}. \quad (14)$$

Im Vergleich mit dem Wert von etwa 13.6eV aus der Praktikumsanleitung weicht der von uns berechnete Wert um etwa  $1.54\sigma$  ab. Der ermittelte Wert von  $\sigma_{12}$  weicht um etwa  $1.03\sigma$  vom Literaturwert<sup>2</sup> von ungefähr 1.0 ab.

<sup>1</sup>Quelle Literaturwerte Energien: <https://physics.nist.gov/PhysRefData/XrayTrans/Html/search.html>

<sup>2</sup>Quelle Literaturwerte  $\sigma_{12}$ ,  $\sigma_{13}$ : [https://de.wikipedia.org/wiki/Moseleysches\\_Gesetz](https://de.wikipedia.org/wiki/Moseleysches_Gesetz)

Aus der Anpassung der Funktion an die  $K_\beta$ -Energien konnten wir die Werte

$$\sigma_{13} = 2.2 \pm 0.4 \quad (15)$$

und

$$E_R = (13.87 \pm 0.21)\text{eV}. \quad (16)$$

bestimmen.

Der hier für  $E_R$  berechnete Wert weicht ebenfalls um etwa  $1.54\sigma$  vom Literaturwert ab. Die Abweichung des von uns ermittelten Wertes von  $\sigma_{13}$  zum Literaturwert von etwa 1.8 beträgt ungefähr  $1.13\sigma$ .

Die bekannten Positionen der  $K_\alpha$ - und  $K_\beta$ -Linien im Röntgenfluoreszenzspektrum eines Metalls lassen sich außerdem verwenden, um die Bestandteile von Metalllegierungen herauszufinden. Um dies zu erproben, untersuchten wir im abschließenden Versuchsteil die Röntgenfluoreszenzspektren fünf verschiedener Metalllegierungen. Hierunter fanden wir die folgenden Legierungen:

- Eisen + Chrom = Ferrochrom(?)
- Kupfer + Zink = Messing (zwei Mal)
- Eisen + Nickel = Invar(?)
- Kupfer + Gallium + Germanium = ?

Die aufgezeichneten Spektren sind noch einmal in den Abbildungen Abbildung (12) und Abbildung (13) zusammengefasst zu sehen.

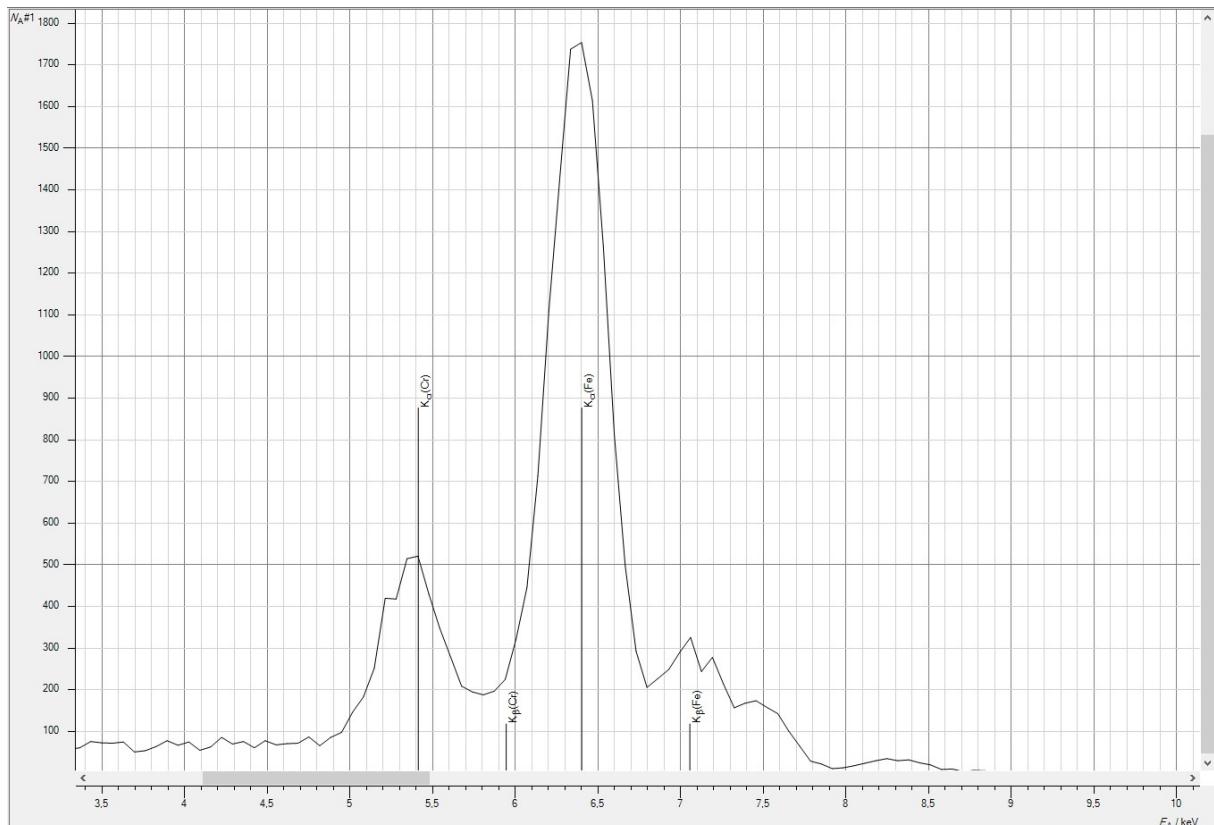


Abbildung 12: Legierung 1: Eisen + Chrom

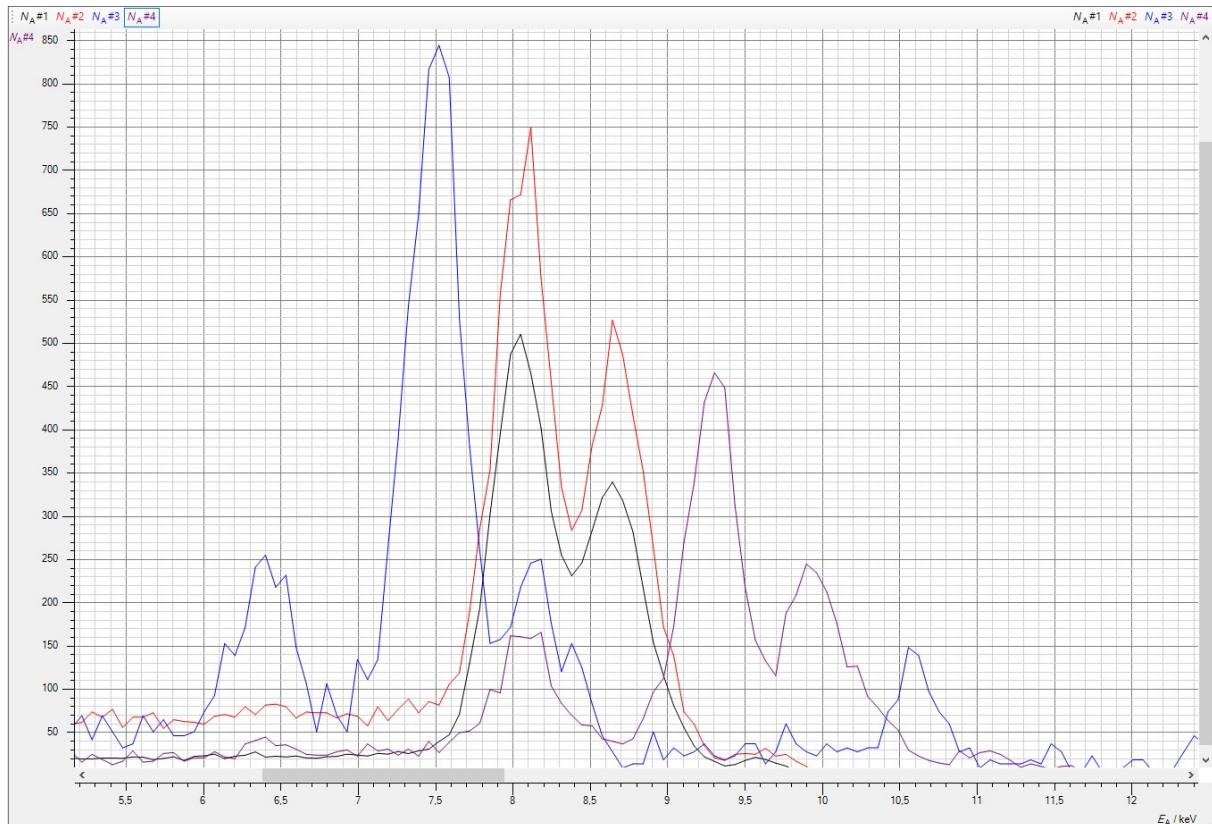


Abbildung 13: Legierungen 2 - 5

# Python Code, Hauptprogramm

auswertung256

April 9, 2025

```
[6]: import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import numpy as np
from scipy.signal import argrelextrema
from scipy.optimize import curve_fit
from scipy.stats import chi2

plt.rcParams.update({'font.size': 12})

%run ../lib.ipynb
LibFormatter.OutputType = 'latex'

class Literaturwerte:
    E_R = ValErr(13.6, 0) # eV - Praktikumsanleitung
    sig12 = ValErr(1, 0) # https://de.wikipedia.org/wiki/Moseleysches_Gesetz
    sig13 = ValErr(1.8, 0) # https://de.wikipedia.org/wiki/Moseleysches_Gesetz

class Element:
    name = ""
    color = ""
    z = 0
    K_alpha = ValErr(0,0)
    K_beta = ValErr(0,0)

    def __init__(self, name, color, z, ka_val, ka_err, kb_val, kb_err):
        self.name = name
        self.color = color
        self.z = z
        self.K_alpha = ValErr(ka_val, ka_err)
        self.K_beta = ValErr(kb_val, kb_err)

    def __repr__(self) -> str:
        return f"{self.name} ({self.z}): K_a: {self.K_alpha.strfmtf2(2, 0)} /_"
        "K_b: {self.K_beta.strfmtf2(2, 0)}"
```

```
[7]: elements = [
    Element("Molybdän", "schwarz", 42, 17.46, 0.18, 19.57, 0.17),
    Element("Eisen", "rot", 26, 6.38, 0.17, 7.03, 0.42),
```

```

Element("Nickel", "blau", 28, 7.46, 0.18, 8.26, 0.21),
Element("Zink", "lila", 30, 8.64, 0.18, 9.59, 0.16),
Element("Zirconium", "cyan", 40, 15.78, 0.18, 17.67, 0.19),
Element("Titan", "ultramarin", 22, 4.44, 0.19, 4.44, 0.19),
Element("Kupfer", "pink", 29, 8.04, 0.17, 8.91, 0.14),
Element("Silber", "rostbraun", 47, 21.89, 0.21, 24.59, 0.18),
]

for i in range(0, len(elements)):
    print(elements[i])

```

```

Molybdän (42): K_a: 17.46 \pm 0.18 / K_b: 19.57 \pm 0.17
Eisen (26): K_a: 6.38 \pm 0.17 / K_b: 7.03 \pm 0.42
Nickel (28): K_a: 7.46 \pm 0.18 / K_b: 8.26 \pm 0.21
Zink (30): K_a: 8.64 \pm 0.18 / K_b: 9.59 \pm 0.16
Zirconium (40): K_a: 15.78 \pm 0.18 / K_b: 17.67 \pm 0.19
Titan (22): K_a: 4.44 \pm 0.19 / K_b: 4.44 \pm 0.19
Kupfer (29): K_a: 8.04 \pm 0.17 / K_b: 8.91 \pm 0.14
Silber (47): K_a: 21.89 \pm 0.21 / K_b: 24.59 \pm 0.18

```

```

[8]: def get_fit_func(n1, n2):
    def fit_func(x, sqrt_Er, sig12):
        return sqrt_Er * (x - sig12) * np.sqrt((1 / n1**2) - (1 / n2**2))
    return fit_func

Zs = np.array([x.z for x in elements])

```

### Auswertung $K_\alpha$ -Linien

```

[9]: K_alpha_vals = np.array([x.K_alpha.val for x in elements])
K_alpha_errs = np.array([x.K_alpha.err for x in elements])

sqrt_K_alpha_vals = np.sqrt(K_alpha_vals)
sqrt_K_alpha_errs = (1 / (2 * sqrt_K_alpha_vals)) * K_alpha_errs

plt.figure(figsize=(12,8))
plt.errorbar(Zs, sqrt_K_alpha_vals, sqrt_K_alpha_errs, fmt=".",
             label="Daten")
plt.xlabel('Kernladungszahl Z')
plt.ylabel(r'$\sqrt{E_\alpha}$ [$\sqrt{\text{eV}}$]')
plt.title(r'$\sqrt{E_\alpha}$ als Funktion von Z')
plt.grid()
plt.legend()
plt.savefig("K_alpha_vs_Z.png", format="png", bbox_inches='tight')

fitfunc12 = get_fit_func(1, 2)

popt_K_alpha, pcov_K_alpha = curve_fit(fitfunc12, Zs, sqrt_K_alpha_vals,
                                         sigma=sqrt_K_alpha_errs, absolute_sigma=True)

```

```

plt.plot(Zs, fitfunc12(Zs, *popt_K_alpha), label="Fit")
plt.legend()
plt.savefig("K_alpha_vs_Z_with_fit.png", format="png", bbox_inches='tight')

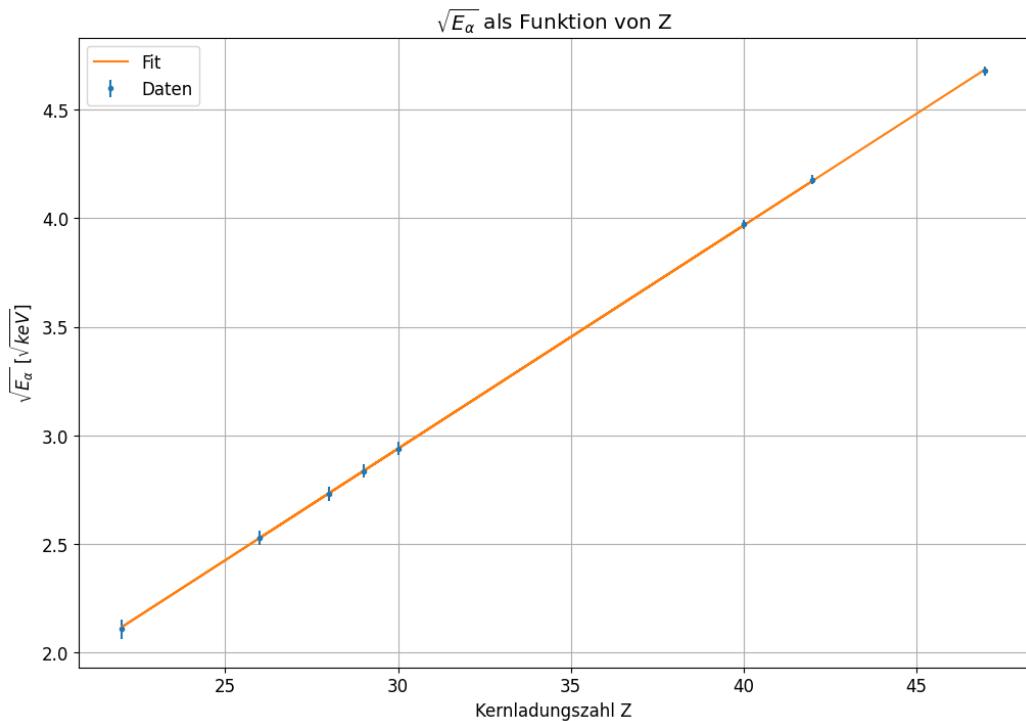
sqrt_Er_K_alpha = ValErr.fromFit(popt_K_alpha, pcov_K_alpha, 0)
sig12_K_alpha = ValErr.fromFit(popt_K_alpha, pcov_K_alpha, 1)

Er_K_alpha = sqrt_Er_K_alpha.pow(2) * 10**3

print_all(
    sqrt_Er_K_alpha.strfmtf2(5, 0, "sqrt(E_R)"),
    sig12_K_alpha.strfmtf2(5, 0, "_12"),
    sig12_K_alpha.sigmadiff_fmt(Literaturwerte.sig12),
    Er_K_alpha.strfmtf2(5, 0, "E_R"),
    Er_K_alpha.sigmadiff_fmt(Literaturwerte.E_R))

sqrt(E_R) = 0.11878 \pm 0.00140
_12 = 1.43499 \pm 0.42318
1.03\sigma
E_R = 14.10930 \pm 0.33193
1.54\sigma

```



## Auswertung $K_\beta$ -Linien

```
[10]: K_beta_vals = np.array([x.K_beta.val for x in elements])
K_beta_errs = np.array([x.K_beta.err for x in elements])

sqrt_K_beta_vals = np.sqrt(K_beta_vals)
sqrt_K_beta_errs = (1 / (2 * sqrt_K_beta_vals)) * K_beta_errs

plt.figure(figsize=(12,8))
plt.errorbar(Zs, sqrt_K_beta_vals, sqrt_K_beta_errs, fmt=". ", label="Daten")
plt.xlabel('Kernladungszahl Z')
plt.ylabel(r'$\sqrt{E_\beta}$ [keV]')
plt.title(r'$\sqrt{E_\beta}$ als Funktion von Z')
plt.grid()
plt.legend()
plt.savefig("K_beta_vs_Z.png", format="png", bbox_inches='tight')

fitfunc13 = get_fit_func(1, 3)

popt_K_beta, pcov_K_beta = curve_fit(fitfunc13, Zs, sqrt_K_beta_vals, u
                                      sigma=sqrt_K_beta_errs, absolute_sigma=True)

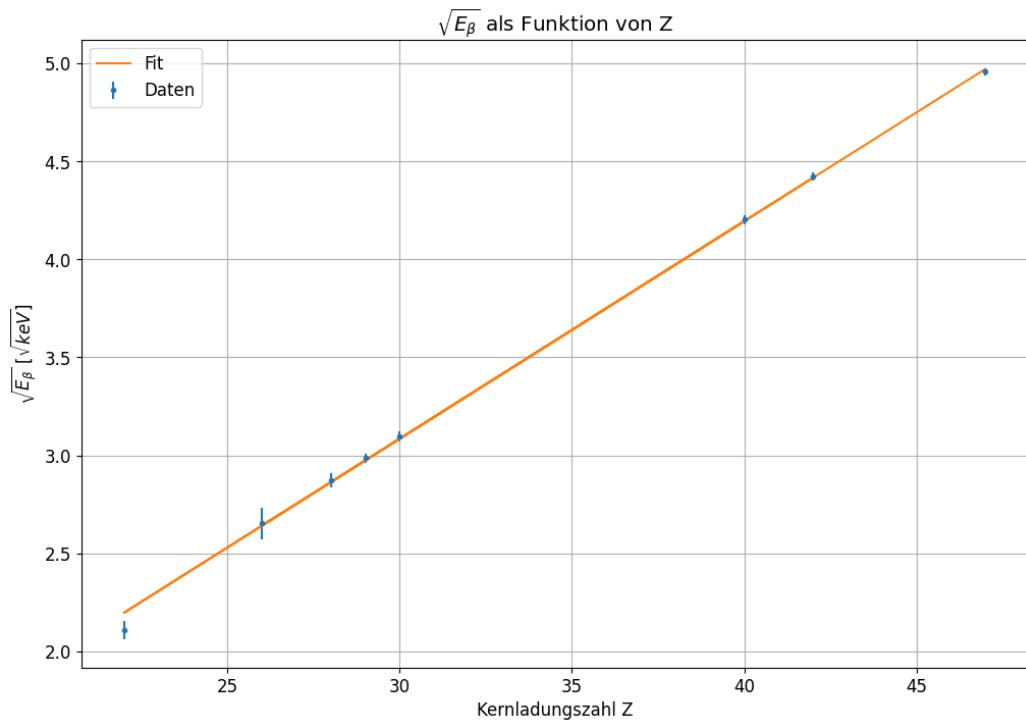
plt.plot(Zs, fitfunc13(Zs, *popt_K_beta), label="Fit")
plt.legend()
plt.savefig("K_beta_vs_Z_with_fit.png", format="png", bbox_inches='tight')

sqrt_Er_K_beta = ValErr.fromFit(popt_K_beta, pcov_K_beta, 0)
sig13_K_beta = ValErr.fromFit(popt_K_beta, pcov_K_beta, 1)

Er_K_beta = sqrt_Er_K_alpha.pow(2) * 10**3

print_all(
    sqrt_Er_K_beta.strfmtf2(5, 0, "sqrt(E_R)"),
    sig13_K_beta.strfmtf2(5, 0, "_13"),
    sig13_K_beta.sigmadiff_fmt(Literaturwerte.sig13),
    Er_K_beta.strfmtf2(5, 0, "E_R"),
    Er_K_beta.sigmadiff_fmt(Literaturwerte.E_R))

sqrt(E_R) = 0.11778 \pm 0.00121
_13 = 2.22144 \pm 0.37472
1.13\sigma
E_R = 14.10930 \pm 0.33193
1.54\sigma
```



```
[11]: Er_mean = (Er_K_alpha + Er_K_beta) / 2
      print_all(
          Er_mean.strfmtf2(5, 0, "E_R"),
          Er_mean.sigmadiff_fmt(Literaturwerte.E_R))
```

E\_R = 14.10930 \pm 0.23471  
2.17\sigma

[ ]:

# Python Code, Bibliothek

lib

April 9, 2025

```
[51]: def floatfmt(v, prec, exp):
    return f"{v/10**exp}:0={prec}f}{LibFormatter.exp10(exp) if exp != 0 else"
        ↵' '}""

def prec_ceil(v, prec=0):
    return np.true_divide(np.ceil(v * 10**prec), 10**prec)

def prec_floor(v, prec=0):
    return np.true_divide(np.floor(v * 10**prec), 10**prec)

[57]: class LibFormatter:
    OutputType = 'text'

    @classmethod
    def exp10(self, exp):
        if LibFormatter.OutputType == 'latex':
            return f' \cdot 10^{{exp}}'
        elif LibFormatter.OutputType == 'text':
            return f'e{exp}'
        else:
            raise ValueError(f"Unsupported OutputType: '{LibFormatter.
        ↵OutputType}'")

    @classmethod
    def pm(self):
        if LibFormatter.OutputType == 'latex':
            return f'\pm'
        elif LibFormatter.OutputType == 'text':
            return f'±'
        else:
            raise ValueError(f"Unsupported OutputType: '{LibFormatter.
        ↵OutputType}'")

    @classmethod
    def sigma(self):
        if LibFormatter.OutputType == 'latex':
            return f'\sigma'
```

```

    elif LibFormatter.OutputType == 'text':
        return f' '
    else:
        raise ValueError(f"Unsupported OutputType: '{LibFormatter.
        ↪OutputType}'")

```

```

[2]: import math
import numpy as np

class ValErr:
    val: float = 0
    err: float = 0
    err_set = False

    def __init__(self, val, err=0):
        self.val = val
        if err != 0:
            self.err_set = True
            self.err = err

    def getTuple(self):
        return (self.val, self.err)

    def setErr(self, err_value):
        self.err_set = True
        self.err = err_value

    @classmethod
    def fromMeasurements(self, measurements):
        return ValErr(np.mean(measurements), (1 / math.sqrt(len(measurements))) ↪
        ↪* np.std(measurements, ddof=1))

    @classmethod
    def fromTuple(self, tup):
        return ValErr(tup[0], tup[1])

    @classmethod
    def fromFit(self, popt, pcov, i):
        return ValErr(popt[i], np.sqrt(pcov[i][i]))

    @classmethod
    def fromFitAll(self, popt, pcov):
        for i in range(0, len(popt)):
            yield ValErr(popt[i], np.sqrt(pcov[i][i]))

    @classmethod
    def fromValPerc(self, v, perc):

```

```

    return ValErr(v, v * perc/100)

def strfmt(self, prec=2):
    if self.err != 0:
        return fr"{{self.val:.{prec}e} {LibFormatter.pm()} {self.err:.{prec}e}}"
    else:
        return f"{{self.val:.{prec}e}}"

def strfmtf(self, prec, exp, name = ""):
    prefix = ""
    if name != "":
        prefix = f"{{name}} = "

    if self.err != 0:
        return prefix + fr"{{floatfmt(self.val, prec, exp)} {LibFormatter.
pm()} {{floatfmt(self.err, prec, exp)}}}"
    else:
        return prefix + f"{{floatfmt(self.val, prec, exp)}}"

def strfmtf2(self, prec, exp, name = ""):
    prefix = ""
    if name != "":
        prefix = f"{{name}} = "

    if self.err != 0:
        return prefix + fr"{{f'({if exp != 0 else ''}{self.val/10**(exp):
0=1.{prec}f} {LibFormatter.pm()} {self.err/10**(exp):0=1.
{prec}f}{f')}{LibFormatter.exp10(exp)}' if exp != 0 else ''}}"
    else:
        return prefix + f"{{floatfmt(self.val, prec, exp)}}"

def strltx(self, prec=2):
    if self.err != 0:
        return fr"{{self.val:.{prec}e} \pm {{self.err:.{prec}e}}}"
    else:
        return f"{{self.val}}"

def relerr(self):
    return self.err / self.val

def sigmadiff(self, other):
    return np.abs(self.val - other.val) / np.sqrt(self.err**2 + other.
err**2)

def sigmadiff_fmt(self, other, prec=2):
    return f"{{prec_ceil(sigmadiff(other), prec)}{LibFormatter.sigma()}}"

```

```

def pow(self, p):
    return ValErr(self.val**2, 2 * self.val * self.err)

def __repr__(self):
    return f"ValErr({self.val}, {self.err})"

def __radd__(self, other):
    return self.__add__(other)

def __add__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(self.val + other.val, math.sqrt(self.err**2 + other.
        ↪err**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(self.val + other, self.err)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
        ↪__class__}' and '{type(other)}'")

def __rsub__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(other.val - self.val, math.sqrt(other.err**2 + self.
        ↪err**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(other - self.val, self.err)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
        ↪__class__}' and '{type(other)}'")

def __sub__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(self.val - other.val, math.sqrt(self.err**2 + other.
        ↪err**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(self.val - other, self.err)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
        ↪__class__}' and '{type(other)}'")

def __rmul__(self, other):
    return self.__mul__(other)

def __mul__(self, other):
    if isinstance(other, self.__class__):

```

```

        return ValErr(self.val * other.val, math.sqrt((other.val * self.
↳err)**2 + (self.val * other.err)**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(self.val * other, self.err * np.abs(other))
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
↳__class__}' and '{type(other)}'")

def __rtruediv__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(other.val / self.val, math.sqrt((other.err / self.
↳val)**2 + (other.val * self.err / self.val**2)**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(other / self.val, np.abs(other / self.val**2) * self.
↳err)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
↳__class__}' and '{type(other)}'")

def __truediv__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(self.val / other.val, math.sqrt((self.err / other.
↳val)**2 + (self.val * other.err / other.val**2)**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(self.val / other, self.err / other)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
↳__class__}' and '{type(other)}'")

```

```
[54]: def spacearound(dat, add):
       return np.linspace(dat[0] - add, dat[len(dat)-1] + add)
```

```
[55]: def div_with_err(a, a_err, b, b_err):
       err = (1 / b) * np.sqrt(a_err**2 + (a * b_err / b)**2)
       return (a / b, err)
```

```
[56]: def print_all(*args):
       for e in args:
           print(e)
```

```
[ ]:
```