BAS - Balmer-Serie

Simon Weinzierl, Yannic Werner

14. Juli 2025

Physikalisches Fortgeschrittenenpraktikum P3B nach der Studienordnung für Studienbeginn bis WS 2022/23

Alle Teile dieses Dokuments (Vorbereitung, Protokoll, Auswertung) wurden von beiden Teilnehmern in gleichen Teilen und ohne fremde Hilfe bearbeitet. Sofern fremde Quellen verwendet wurden, sind diese angegeben.

Der LaTeX-Code ist auf GitHub unter https://github.com/WeinSim/P3B/BAS verfügbar.

Inhaltsverzeichnis

1 Vobereitung **4**
1.1 Physikalischer Hintergrund 4
1.1.1 Bahndrehimpulsoperator L und Eigenwerte 4
1.1.2 Wellenansatz zur Lösung der Schrödingergleichung 4
1.1.3 Laplace-Operator in Kugelkoordinaten 5
1.1.4 Deutung des Wasserstoffspektrums 6
1.1.5 Prismen- und Gitterspektrometer, Auflösung 6
1.1.6 Beugung, Brechung 7
1.1.7 Geometrische Optik, Abbildungsfehler an Linsen 7
1.1.8 Bildkonstruktion an Sammellinsen 8
1.2 Aufgabe zur Vorbereitung 9

2 Veruschsablaufplan **10**
2.1 Benötigte Materialien 10
2.2 Teilversuch 1: Bragg-Reflexion von Röntgenstrahlung des Molybdän an einem NaCl-Eiskristall 11
2.3 Teilversuch 2: Energiespektrum einer Röntgenröhre in Abhängigkeit der Spannung 12

3 Versuchsprotokoll **13**

4 Auswertung **14**
4.1 Teilversuch 1: Bragg-Reflexion von Röntgenstrahlung des Molybdän an einem NaCl-Eiskristall 14

5 Anmerkung: Graphische Auswertung und Fehlerfortpflanzung mit Python-Code **15**

Literatur

[Gri13] David J. Griffiths. *Introduction to electrodynamics*. Pearson, 2013.

[Gri18] David J. Griffiths. *Introduction to Quantum Mechanics*. Cambridge University Press, 2018.

[Pol25] Mikhail Polyanskiy. Refractiveindex.info - refractive index database. https://refractiveindex.info, 2025.

[Zin18] Wolfgang Zinth. *Optik. Lichtstrahlen - Wellen - Photonen*. De Gruyter Studium, 5th edition, 2018.

1 Vobereitung

1.1 Physikalischer Hintergrund

1.1.1 Bahndrehimpulsoperator L und Eigenwerte

Genau wie andere Observablen, wie z.B. Position, Impuls oder Energie hat auch der Drehimpuls $\vec{L}$ einen zugehörigen Operator in der Quantenmechanik. Analog zur klassischen Gleichung $\vec{L} = \vec{r} \times \vec{p}$ gilt in der Quantenmechanik $\hat{\vec{L}} = \hat{\vec{r}} \times \hat{\vec{p}}$. Beispielsweise gilt also für die z-Komponente des Drehimpulsoperators $\hat{L}_z = x\hat{p}_y - y\hat{p}_x$. (Die Reihenfolge der Faktoren spielt dabei keine Rolle, weil r_i und $\hat{p}_j$ für $i \neq j$ kommutieren.)

Die verschiedenen Drehimpulsoperatoren kommutieren nicht untereinander, stattdessen erfüllen sie die Drehimpulsalgebra

$$[\hat{L}_x, \hat{L}_y] = i\hbar L_z, \quad [\hat{L}_y, \hat{L}_z] = i\hbar L_x, \quad [\hat{L}_z, \hat{L}_x] = i\hbar L_y.$$

Jedoch kommutiert der Operator $\hat{L}^2$, der das Betragsquadrat des Gesamtdrehimpulses misst, mit jedem der Operatoren $\hat{L}_x, \hat{L}_y, \hat{L}_z$.

Im Wasserstoffatom können die (gemeinsamen) Eigenwerte von $\hat{L}^2$ und $\hat{L}_z$ durch die Leiteroperatoren $\hat{L}_\pm = \hat{L}_x \pm i\hat{L}_y$ bestimmt werden. Sei ψ sowohl eine Eigenfunktion von $\hat{L}^2$ zum Eigenwert λ als auch eine Eigenfunktion von $\hat{L}_z$ zum Eigenwert μ. Dann ist $\hat{L}_\pm\psi$ eine Eigenfunktion von $\hat{L}^2$ zum gleichen Eigenwert λ und eine Eigenfunktion von $\hat{L}_z$ zum Eigenwert $\mu \pm \hbar$:

$$\begin{aligned} \hat{L}^2(\hat{L}_\pm\psi) &= \hat{L}_\pm(\hat{L}^2\psi) = \hat{L}_\pm(\lambda\psi) = \lambda(\hat{L}_\pm\psi) \\ \hat{L}_z(\hat{L}_\pm\psi) &= [\hat{L}_z, L_\pm]\psi + \hat{L}_\pm\hat{L}_z\psi = (\pm\hbar\hat{L}_\pm)\psi + \hat{L}_\pm(\mu\psi) = (\mu \pm \hbar)(\hat{L}_\pm\psi). \end{aligned}$$

Wegen $L_z^2 < L^2$, muss es eine oberste und eine unterste Stufe der "Leiter" an Eigenzuständen geben. Es folgt, dass der Eigenwert von $\hat{L}^2$ genau $\hbar^2 l(l+1)$ ist, und dass die verschiedenen Eigenwerte von $\hat{L}_z$ genau $\hbar m$ sind, wobei l und m die Nebenquantenzahlen des Zustands des Elektrons sind.

Die z-Achse war bei den Rechnungen beliebig gewählt. Also sind $\hbar m$ für $-l \leq m \leq l$ die Eigenzustände des Drehimpulses um jede beliebige Achse.

[Gri18, 157–161]

1.1.2 Wellenansatz zur Lösung der Schrödingergleichung

Die allgemeine (zeitabhängige) Schrödingergleichung lautet

$$i\hbar\frac{\partial\Psi}{\partial t} = \hat{H}\Psi,$$

wobei der Hamiltonoperator $\hat{H}$ durch die Summe der kinetischen und potentiellen Energie ausgedrückt werden kann:

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V.$$

In einem zeitunabhängigen Potential kann diese Gleichung vereinfacht werden. Ist ψ eine Lösung der zeitunabhängigen Schrödingergleichung

$$E\psi = \hat{H}\psi,$$

wobei E die Energie des Systems darstellt, dann ist Ψ, definiert durch

$$\Psi(x,t) = \psi(x)e^{-iEt/\hbar}, \qquad (*)$$

eine Lösung der allgemeinen Schrödingergleichung. Für ein freies Teilchen, d.h. ein Teilchen mit $V(x) = 0$, lautet die zeitunabhängige Schrödingergleichung

$$E\psi = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2}.$$

Diese kann einfach gelöst werden: weil die zweite Ableitung von ψ proportional zu sich selbst ist, ist die allgemeine Lösung der Gleichung eine Exponentialfunktion:

$$\psi(x) = Ae^{ix\sqrt{2mE}/\hbar} + Be^{-ix\sqrt{2mE}/\hbar},$$

wobei die Amplituden A und B durch Rand- und Normierungsbedingungen bestimmt werden. Zusammen mit (*) sowie den Bezeichnungen $\omega = E/\hbar$ und $k = \pm\sqrt{2mE}/\hbar$ ergibt sich damit die Lösung

$$\Psi(x,t) = e^{i(kx-\omega t)}.$$

Dies ist die Gleichung für eine ebene Welle mit einer Winkelgeschwindigkeit ω und einem Wellenvektor k. Freie Teilchen breiten sich in der Quantenmechanik also als ebene Wellen aus. Weil diese Lösung allerdings nicht normierbar ist, gibt es keine freien quantenmechanischen Teilchen mit wohldefinierter Energie. Stattdessen treten sie immer als Linearkombinationen über einen (kontinuierlichen) Energiebereich auf.

Die Wellennatur quantenmechanischer Teilchen ist nicht nur bei freien Teilchen beobachtbar. Beispielsweise werden die gebundenen Zustände eines kastenförmigen Potentials durch stehende Wellen beschrieben. Für einen unendlich tiefen Potentialtopf der Breite a gilt für den n-ten gebundenen Zustand

$$\Psi(x,t) = \sqrt{\frac{2}{a}}\sin\left(\frac{n\pi}{a}x\right)e^{-iE_nt/\hbar},$$

mit $E_n = (n\pi\hbar)^2/(2ma^2)$.

[Gri18, 25–32, 55–57]

1.1.3 Laplace-Operator in Kugelkoordinaten

Der Laplace-Operator $\Delta = \vec{\nabla}^2$ ist eine Verallgemeinerung der zweiten Ableitung in drei Dimensionen. In kartesischen Koordinaten hat er eine einfach Form:

$$\Delta = \vec{\nabla}^2 = \begin{pmatrix}\partial_x\\ \partial_y\\ \partial_z\end{pmatrix}^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}.$$

In den krummlinigen Kugelkoordinaten hat bereits der Nabla-Operator $\vec{\nabla}$ eine kompliziertere Form:

$$\vec{\nabla} = \frac{\partial}{\partial r}\hat{e}_r + \frac{1}{r}\frac{\partial}{\partial\theta}\hat{e}_\theta + \frac{1}{r\sin\theta}\frac{\partial}{\partial\varphi}\hat{e}_\varphi.$$

$\hat{e}_r, \hat{e}_\theta$ und $\hat{e}_\varphi$ stehen dabei jeweils für die lokalen, orthonormierten Einheitsvektoren in Richtung der r-, θ- bzw. φ-Koordinaten. Für den Laplace-Operator folgt daraus

$$\Delta = \vec{\nabla}^2 = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\sin\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2}{\partial\varphi^2}.$$

[Gri13, 42]

1.1.4 Deutung des Wasserstoffspektrums

Das Elektron eines Wasserstoffatoms kann nur bestimmte, diskrete Energieniveaus einnehmen. Wechselt das Elektron von einem Ausgangszustand n in einen Endzustand m niedrigerer Energie, so wird die Energie in Form eines Photons freigesetzt. Die Wellenlänge λ des Photons ist durch die Energiedifferenz der beiden Zustände vollständig bestimmt. Es gilt

$$\frac{1}{\lambda} = R_\infty \left(\frac{1}{n^2} - \frac{1}{m^2} \right),$$

wobei $R_\infty = (m_e e^4)/(8\varepsilon_0^2 h^3 c)$ die Rydberg-Konstante ist. Die exakte Wellenlänge hängt jedoch auch von der Masse des Kerns ab, welche in der obigen Formel durch R_∞ als unendlich angenommen wurde. Um die korrekten Wellenlängen für einen Kern mit endlicher Masse m_k zu bestimmen, muss die Rydberg-Konstante R_∞ durch R_{m_k} ausgetauscht werden:

$$R_{m_k} = \frac{m_e m_k e^4}{8\varepsilon_0^2 h^3 c(m_e + m_k)}.$$

Daraus folgt, dass das Spektrum eines Wasserstoffatoms (mit einer Kernmasse von ca. 1u) leicht vom dem eines Deuteriumatoms (Kernmasse ca. 2u) abweicht.

Gruppiert man die so entstehenden Wellenlängen nach dem Endzustand m, so erhält man verschiedene "Reihen" an möglichen Wellenlängen. Übergänge mit dem Endzustand $m = 1$ werden der Lyman-Serie zugeordnet, Übergänge mit dem Endzustand $m = 2$ werden der Balmer-Serie zugeordnet und Übergänge mit dem Endzustand $m = 3$ werden der Paschen-Serie zugeordnet.

Die auf diese Weise emittierten Wellenlängen sind charakteristisch für das Wasserstoffatom. Weil Elektronen auch in anderen Atomen und Molekülen nur diskrete Energieniveaus annehmen können, diese sich aber von denen des Wasserstoffatoms unterscheiden, hat jedes Atom bzw. Molekül ein eigenes charakteristisches Spektrum - sozusagen einen "Fingerabdruck". Aus den emittierten Wellenlängen einer Substanz kann also auf dessen chemische Zusammensetzung geschlossen werden.

1.1.5 Prismen- und Gitterspektrometer, Auflösung

Um eine unbekannte Lichtquelle (beispielsweise das Emissionsspektrum eines Wasserstofatoms) nach dessen Wellenlängenzusammensetzung zu untersuchen, müssen die einzelnen Wellenlängen räumlich voneinander getrennt werden. Dies kann beispielsweise mithilfe eienes Prismas oder eines optischen Gitters gemacht werden.

Trifft Licht aus Luft auf eine Glasoberfläche, so wird es in der Einfallsebene in Richtung der Normale abgelenkt. Der Ablenkungswinkel ist dabei abhängig vom Einfallswinkel und vom Brechungsindex des Glases. Der Brechungsindex ist wiederum nicht konstant, sondern nimmt für unterschiedliche Wellenlängen einen leicht unterschiedlichen Wert an. Insgesamt wird Licht durch ein Prisma in seine verschiedenen Wellenlängen getrennt. Die Winkeldifferenz zwischen zwei Wellenlängen ist jedoch abhängig von der Differenz in den Brechungsindizes, welche in der Regel sehr klein ist. Beispielsweise hat Glas N-BK7 hat im blauen Bereich bei $\lambda = 350\,\text{nm}$ einen Brechungsindex von $1,53924$, während der Brechungsindex im roten Bereich bei $\lambda = 750\,\text{nm}$ einen Wert von $1,5118$ annimmt ([Pol25]).

Auch mithilfe eines Gitters kann Licht in seine Wellenlängen zerlegt werden. Das zugrundeliegende Prinzip ist nicht wie beim Prisma die Brechung, sondern die Beugung. Durch konstruktive bzw. destruktive Interferenz entstehen auf einem Schirm Maxima zu verschiedenen Ordnungen. Eine wichtige Rolle spielt dabei die Gitterkonstante g, die die Anzahl an Strichen pro Längeneinheit angibt. Trifft Licht unter einem Einfallswinkel δ (gemessen zum Lot) auf ein Reflexionsgitter, dann lautet die Bedingung für den Winkel

α, unter dem das Maximum der Ordnung m auftritt

$$\sin\alpha \;=\; m\frac{\lambda}{g} - \sin\delta.$$

Anders als beim Prismenspektrometer ist der Winkel nun proportional zur Wellenlänge λ. Rotes Licht ($\lambda = 350\,\text{nm}$) wir nun also mehr als doppelt so stark abgelenkt als blaues Licht $\lambda = 750\,\text{nm}$). Dadurch können einzelne Wellenlängen genauer bestimmt werden und nahe beieinanderliegende Wellenlängen besser unterschieden werden. Weil es aber nun mehrer Intensitätsmaxima gibt, kann es passieren, dass ein Maximum einer Wellenlänge zu einer Ordnung sicht mit dem Maximum einer anderen Wellenlänge zu einer anderen Ordnung überlagert. Dieser Effekt wird mit zunehmender Ordnung stärker. Dadurch wird der tatsächlich nutzbare Spektralbereich ($\delta\lambda$) eingeschränkt.

Will man zwei nahe beieinanderliegende Maxima unterscheiden, so spielt das Auflösungsvermögen A des Spektrometers eine wichtige Rolle. Das Maximum einer Wellenlänge sollte auf dem Minumum der anderen Wellenlänge liegen. Daraus ergibt sich folgende Bedingung:

$$A \;=\; \frac{\lambda}{\Delta\lambda} \;=\; mN,$$

wobei N die Anzahl der beleuchteten Spalten ist.

1.1.6 Beugung, Brechung

Licht kann klassisch als Wellenphänomen betrachtet werden und zeigt damit typische Welleneigenschaften wie zum Beispiel Brechung und Beugung. Die Wechselwirkung von Licht, einer Welle im elektrischen und magnetischen Feld, mit Materie, die aus geladenen Teilchen besteht, bewirkt, dass sich Licht in einem Medium langsamer ausbreitet als im Vakuum. Das Verhältnis der Lichtgeschwindigkeit im Vakuum zur Ausbreitungsgeschwindigkeit in einem Medium ist der Brechungsindex dieses Mediums. Er hängt primär von den Materialeigenschaften ab, wird jedoch auch von Temperatur und Wellenlänge leicht beeinflusst.

Beim Übergang von einem Medium mit Brechungsindex n_1 in ein Medium mit Brechungsindex n_2 wird eine einfallende Lichtwelle abgelenkt. Dies passiert in der Ebene, die vom Wellenvektor $\vec{k}$ und dem Normalenvektor $\vec{n}$ aufgespannt wird. Beim Übgergang in ein optisch dichteres Medium, d.h. im Fall $n_2 > n_1$, passiert die Beugung zum Lot hin, andernfalls wird der Lichtstrahl vom Lot weg gebeugt. Der genaue Zusammenhang zwischen Einfallswinkel α und Ausfallswinkel β wird durch das Snellius'sche Brechungsgesetzt beschrieben:

$$\frac{\sin\alpha}{\sin\beta} \;=\; \frac{n_2}{n_1}.$$

Das Huygens'sche Prinzip besagt, dass jeder Punkt einer Wellenfront als Ausgangspunkt einer neuen Kugelwelle betrachtet werden kann. Wendet man dieses Prinzip auf eine ebene (Licht-)Welle an, die sich durch einen Spalt oder an einer Kante vorbei bewegt, erhält man als Ergebnis, dass die Lichtwelle am Spalt oder Hindernis gebeugt wird. Dadurch erreicht das Licht Bereiche, die nach der geometrischen Optik im "Schatten" liegen müssten. Wesentliche Beugungseffekte treten nur dann auf, wenn die Größe des Hindernisses in etwa in der Größenordnung der Wellenlänge des Lichts liegt.

[Zin18, 31–34, 140–142]

1.1.7 Geometrische Optik, Abbildungsfehler an Linsen

In der geometrischen Optik werden Wellenphänome des Lichts, wie beispielsweise Beugung oder Interferenz, vernachlässigt. Stattdessen interessiert man sich für "Lichtstrahlen", die sich in einem homogenen

Medium geradlinig ausbreiten und an Grenzflächen gebrochen und reflektiert werden. Im Bezug auf praktische Anwendungen der geometrischen Optik spielt die Bildkonstruktion eine wichtige Rolle. Ein Bild eines Gegenstandes G im Punkt B entsteht dann, wenn es für einen Betrachter so aussieht, als würde sich der Gegenstand im Punkt B befinden. Dabei werden zwischen reellen und virtuellen Bildern unterschieden: bei einem reellen Bild laufen alle Strahlen durch den Bildpunkt B und breiten sich von dort aus geradlinig aus. Dadurch kann das Bild mit einem Schirm beboachtet werden. Bei einem virtuellen Bild treffen die Lichtstrahlen nie an einem Punkt zusammen, folglich kann ein virtuelles Bild nicht mit einem Schirm bebobachtet werden.

In der Praxis ist es unmöglich, eine mathematisch perfekte Abbildung zu konstruieren, es sei denn, sie besteht nur aus Reflexionen an ebenen Flächen. Also muss man immer bestimme Abbildungsfehler in Kauf nehmen. Zu diesen Abbildungsfehlern zählen beispielsweise:

- *Chromatische Aberrationen.* Wie bereits oben ewähnt ist der Brechungsindex eines Materials von der Wellenlänge des darauffallenden Lichts abhängig. Dies hat zur Folge, dass die verschiedenen Wellenlängen eines weißen Lichtstrahls von Linsen und ähnlichen optischen Elementen unterschiedlich stark gebrochen werden (dies ist genau die Funktionsweise des Prismenspektrometers!). Die Brennweite einer Linse ist damit keine Konstante mehr, sondern abhängig von der Farbe des Lichts.

- *Sphärische Aberrationen.* Linsen mit kugelförmigen Oberflächen werden in der Praxis oft eingesetzt. Die Brennweite einer solchen Linse ist jedoch nicht konstant, sondern hängt von der Entfernung der eintreffenden Strahlen von der optischen Achse ab: achsferne Strahlen werden früher gebündelt (d.h. sie haben eine kleinere Brennweite) als achsnahe Strahlen. Dieser Abbildungsfehler kann zwar für einzelne Gegenstandsweiten korrigiert werden, jedoch ist es unmöglich, ihn für alle Gegenstandsweiten gleichzeitig zu beheben.

- *Astigmatismus.* Fällt ein Lichtbündel in einem großen Winkel zur optischen Achse auf eine Linse, dann wird es nicht in einem einzigen Punkt gesammelt. Stattdessen tritt die Bündelung zunächst entlang einer Achse auf, und in einem größeren Abstand zur Linse entlang einer dazu orthogonalen Achse. Der Brennpunkt hat sich also auf einen Brennbereich ausgeweitet.

Abbildungsfehler führen allgemein zu unscharfen oder verschwommenen optischen Abbildungen. Jede Art von Fehler kann auf eine gewisse Art und Weise minimiert werden, allerdings sind diese Minimierungen sehr spezifisch auf bestimmte Winkel, Abstände oder Wellenlängen ausgerichtet. Häufig bringt die Verringerung eines Fehlers auch die Vergrößerung eines anderen Fehlers mit sich.

[Zin18, 69–116]

1.1.8 Bildkonstruktion an Sammellinsen

Die Brennweite einer Sammellinse gibt die Entfernung von der Linse an, in der parallel einfallende Lichtstrahlen gebündelt werden. Bei einem (beidseitigen) Krümmungsradius r und einem Brechungsindex n berechnet sie sich durch die Formel

$$\frac{1}{f} = (n-1)\frac{2}{r}.$$

Sei ein Gegenstand in einer Entfernung g links von einer Sammellinse mit Brennweite f platziert. Für die Entfernung b des Bildpunktes zur Linse gilt

$$\frac{1}{g} + \frac{1}{b} = \frac{1}{f}.$$

Für $g > f$ entsteht ein reelles Bild rechts von der Linse, während für $g < f$ ein virtuelles Bild links von der Linse entsteht.

Bei der Strahlenkonstruktion an einer Sammellinse können immer folgende Regeln berücksichtigt werden:

(I) Parallel einfallende Strahlen gehen durch den Brennpunkt.

(II) Strahlen, die durch den Mittelpunkt der Linse verlaufen, werden nicht abgelenkt.

(III) Strahlen, die durch den Brennpunkt gekommen sind, werden zu parallelen Strahlen.

Daraus ergibt sich folgende Konstruktion für die Bildentstehung eines Gegenstandes:

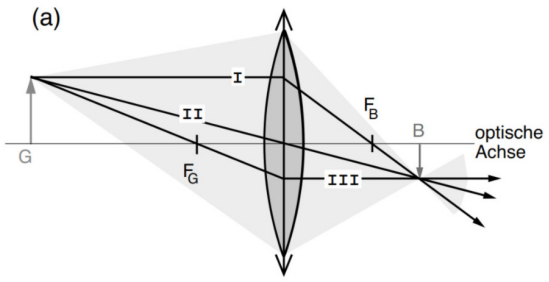

Abbildung 1: Bildkonstruktion an einer Sammellinse. Quelle: [Zin18, 95]

Die Strahlen (I), (II) und (III) wurden jeweils durch Anwendung der entsprechenden Regel konstruiert.

[Zin18, 86–97]

1.2 Aufgabe zur Vorbereitung

In der gesamten Versuchsdurchführung wird jeweils die H_α-Linie betrachtet. Diese entspricht dem Übgergang vom 3. in den 2. angeregten Zustand, also dem Übergang von $n = 2$ zu $m = 3$.

Sei λ_H die Wellenlänge der H_α-Linie des Wasserstoffatoms und λ_D die der H_α-Linie des Deuteriumatoms. Für das Wasserstoffatom gilt

$$\lambda_H = \left(R_H\left(\frac{1}{2^2}-\frac{1}{3^2}\right)\right)^{-1} = \frac{m_e+m_H}{m_e m_H}\frac{8\varepsilon_0^2 h^3 c}{e^4}(1/4-1/9)^{-1} = \frac{288(m_e+m_H)\varepsilon_0^2 h^3 c}{5 m_e m_H e^4}$$

und analog für das Deuteriumatom

$$\lambda_D = \frac{288(m_e+m_D)\varepsilon_0^2 h^3 c}{5 m_e m_D e^4}.$$

Daraus berechnet sich eine absolute Wellenlängendifferenz von

$$\Delta\lambda = \frac{288\varepsilon_0^2 h^3 c}{5e^4}\left(\frac{m_e+m_H}{m_e m_H}-\frac{m_e+m_D}{m_e m_D}\right) = 0,1849\,\text{nm} \quad \texttt{vorbereitung()}$$

sowie eine relative Differenz von

$$\frac{\Delta\lambda}{\lambda_H} = 2,872\cdot 10^{-4}. \quad \texttt{vorbereitung()}$$

2 Veruschsablaufplan

2.1 Benötigte Materialien

1. Eins
2. Zwei

2.2 Teilversuch 1: Bragg-Reflexion von Röntgenstrahlung des Molybdän an einem NaCl-Eiskristall

(I) Ziel: ...

(II) Versuchsmethode: ...

(III) Versuchsskizze:

Abbildung 2: Versuchsskizze Teilversuch 1

(IV) Planung der Durchführung

- eins
- zwei

(V) Vorüberlegungen zur Durchführung & Auswertung

- eins
- zwei

2.3 Teilversuch 2: Energiespektrum einer Röntgenröhre in Abhängigkeit der Spannung

(I) Ziel:

(II) Versuchsmethode:

(III) Versuchsskizze:

Abbildung 3: Versuchsskizze Teilversuch 2

(IV) Planung der Durchführung

- eins
- zwei

(V) Vorüberlegungen zur Durchführung & Auswertung

- eins
- zwei

3 Versuchsprotokoll

Auf den folgenden Seiten befindet sich das eingescannte Versuchsprotokoll. Alle Daten wurden selbst gemessen. Sofern fremde Hilfe benutzt wurde, wurde sie klar gekennzeichnet.

Messunsicherheiten wurden angegeben und folgend in der Auswertung verwendet. Alle weiteren Rechnungen und Analysen finden in der Versuchsasuwertung statt.

..... width=!,height=!,pages=..., pagecommand=, frame=true

4 Auswertung

4.1 Teilversuch 1: Bragg-Reflexion von Röntgenstrahlung des Molybdän an einem NaCl-Eiskristall

5 Anmerkung: Graphische Auswertung und Fehlerfortpflanzung mit Python-Code

Alle Berechnungen inkl. Fehlerbestimmung wurden mit einem selbstgeschriebenen Python-Skript durchgeführt, um uns die Arbeit zu erleichtern und Fehler zu vermeiden. Alle Ergebnisse, die auf diese Weise zustande gekommen sind, sind entsprechend mit einem blauen Hintergrund gekennzeichnet; s. folgendes Beispiel:

$$F \;=\; ma \;=\; 20\,\mathrm{kg} \cdot 9,81\,\frac{\mathrm{m}}{\mathrm{s}^2} \;=\; (19,62 \pm 0,5)\,\mathrm{N} \quad \texttt{tv1()}$$

Dies soll bedeuten, dass die Berechnung des Wertes und der Unsicherheit von der Python-Funktion namens `tv1` durchgeführt wird. Die Unsicherheit wird mithilfe der Gauß'schen Fehlerfortpflanzung berechnet. Außerdem wird das Python-Package `matplotlib` zum Erstellen von Graphen verwendet.

Der verwendete Code ist sowohl auf GitHub verfügbar (https://github.com/WeinSim/P3B/BAS) als auch auf den folgenden Seiten zu finden und kann mit dem Befehl `python Main.py` ausgeführt werden. Für eine genauere Beschreibung des Codes siehe die README-Datei auf GitHub sowie die Kommentare im Code. (Manche Sonderzeichen im Code (ä, ö, ü, Δ, etc.) werden von LaTeXnicht richtig erkannt, deswegen kann der Code auf den nachfolgenden Seiten an einigen Stellen unvollständig erscheinen. Auf GitHub wird aber alles richtig angezeigt.)

Main.py:

```
import math
import random as rd
import numpy as np

import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

from Expressions import *

# markers - linestyle - color

def tv1():
    print("--- Teilversuch 1 ---")

    # transmittiert
    # mm
    # r = [0.0, 5.5, 7.5, 8.5, 10.0, 10.5, 11.0, 12.0, 12.5, 13.0, 13.5]
    r = [4.5, 5.5, 7.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.5, 12.0]
    evalTV1("Transmittiert", r)

    # transmittiert
    # mm
    r = [5.0, 6.0, 7.0, 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0]
    evalTV1("Reflektiert", r)

def evalTV1(name, r):
    n = np.arange(1, len(r) + 1)
    r2 = []
    for ri in r:
        r2.append(ri * ri)
    coefs = np.polyfit(n, r2, 1)

    # deltaCoefs = linRegUncertainty(n, r2, coefs)
    deltaCoefs = [0.1, 0.1]

    varM = Var(coefs[0], deltaCoefs[0], "m")
    varT = Var(coefs[1], deltaCoefs[1], "t")

    params = [varM]

    cR = Const(12.141e3)
    cL = Const(635e-6)

    r = Div(varM, cL)
    l = Div(varM, cR)

    print(f"{name}:")

    printVar(varM)
    printVar(varT)

    rVal = r.eval()
    rUnc = gaussian(r, params)
    printExpr("R", rVal * 1e-3, rUnc * 1e-3)
    printDiff(rVal, cR.eval(), rUnc)

    lVal = l.eval()
    lUnc = gaussian(l, params)
    printExpr("lambda", lVal * 1e3, lUnc * 1e3)
    printDiff(lVal, cL.eval(), lUnc)
```

```
61
62      print()
63
64      fit = np.polyval(coefs, n)
65
66      pp = PdfPages(f"../Abbildungen/GraphTV1_{name}.pdf")
67
68      plt.figure()
69      plt.clf()
70
71      # plt.plot(d, p, "-o")
72      plt.plot(n, r2, "o", label=f"Messwerte")
73      plt.plot(n, fit, "-", label=f"Ausgleichsgerade")
74
75      plt.title(f"Radius^2 vs. Interfernzordnung")
76      plt.xlabel('Interferenzordnung')
77      plt.ylabel('Radius^2 (m^2)')
78      plt.legend()
79
80      pp.savefig()
81      pp.close()
82
83  def printExpr(name, value, unc):
84      print(f"{name}  = {value}")
85      print(f"   {name} = {unc}")
86
87  def printVar(var):
88      printExpr(var.name, var.value, var.uncertainty)
89
90  def printDiff(val, theo, unc):
91      u = abs(val - theo) / unc
92      print(f"Abweichung vom theoretischen Wert: {u} * Unsicherheit")
93
94  def tv2():
95      print("--- Teilversuch 2 ---")
96
97      print("Fresnelbiprisma 1:")
98      s = Var(385e-3, 3e-3, "s")
99      bB = Var(18e-3, 1e-3, "B")
100     b = Var(2180e-3, 5e-3, "b")
101     f = Const(300e-3)
102     dm = Var(30e-3, 1e-3, "delta_m")
103     evalSpiegel(s, bB, b, f, dm, 54)
104
105     print("Fresnelbiprisma 2:")
106     s = Var(385e-3, 3e-3, "s")
107     bB = Var(12e-3, 1e-3, "B")
108     b = Var(2115e-3, 5e-3, "b")
109     f = Const(300e-3)
110     dm = Var(20e-3, 1e-3, "delta_m")
111     evalSpiegel(s, bB, b, f, dm, 25)
112
113     print("Fresnelbiprisma 3:")
114     s = Var(385e-3, 3e-3, "s")
115     bB = Var(9e-3, 1e-3, "B")
116     b = Var(2115e-3, 5e-3, "b")
117     f = Const(300e-3)
118     dm = Var(15e-3, 1e-3, "delta_m")
119     evalSpiegel(s, bB, b, f, dm, 13)
120
121     print("Fresnelbiprisma:")
```

```
122     s = Var(380e-3, 3e-3, "s")
123     bB = Var(21e-3, 1e-3, "B")
124     b = Var(2510e-3, 5e-3, "b")
125     f = Const(300e-3)
126     dm = Var(25e-3, 1e-3, "delta_m")
127     evalSpiegel(s, bB, b, f, dm, 35)
128 
129 def evalSpiegel(s, bB, b, f, dm, m):
130     params = [s, bB, b, f, dm]
131 
132     a = Div(Mult(bB, f), Sub(b, f))
133     printExpr("a", a.eval() * 1e3, gaussian(a, params) * 1e3)
134 
135     delta = Div(dm, Const(m))
136 
137     lam = Div(Mult(a, delta), Add(s, b))
138     lamUnc = gaussian(lam, params)
139     lamVal = lam.eval()
140     printExpr("lambda", lamVal * 1e6, lamUnc * 1e6)
141 
142     theo = 635e-9
143 
144     unc = abs(lamVal - theo) / lamUnc
145     print("Abweichung vom theoretischen Wert: %.3f * Unsicherheit" % (unc))
146     print()
147 
148 tv1()
149 tv2()
```

Expressions.py:

```
import math

# Diese Datei enth lt verschieden Klassen um arithmetische Operationen zu
# repr sentieren. Zu den Operationen z hlen Addition (Add), Subtraktion (Sub),
# Multiplikation (Mul), Division (Div), Exponentiation (Pow), Sinus (Sin),
# Cosinus (Cos) und der Zehnerlogarithmus (Log10).
# Diese Operationen bilden die Knoten der entstehenden Syntaxb ume.
# Jede Instanz besitzt ein bzw. zwei arithmetische Ausdr cke als "Kinder".
# Die Bl tter bilden Konstanten (Const) und Variablen (Var).
# Konstanten besitzen einen festen Wert, w hrend Variablen einen Wert
# und eine Unsicherheit haben.
# Ein arithmetischer Ausdruck kann mit der Funktion eval() ausgewertet werden.
# Mit derivative() wird die Ableitung nach der angegebenen Variable gebildet,
# welche wieder ein arithmetischer Ausdruck ist.
# Mit den Funktionen gaussian und minMax wird die gau 'sche Unsicherheit
# bzw. die Min-Max-Unsicherheit eines arithmetischen Ausdrucks
# mit den gegebenen Parametern berechnet.

class Add:

    def __init__(self, child1, child2):
        self.child1 = child1
        self.child2 = child2

    def eval(self):
        return self.child1.eval() + self.child2.eval()

    def derivative(self, var):
        return Add(self.child1.derivative(var), self.child2.derivative(var))

    def __str__(self):
        return toStr(self, "+")

    def isEqual(self, other):
        if not isinstance(other, Add):
            return False
        if self.child1.isEqual(other.child1) and self.child2.isEqual(other.child2):
            return True
        if self.child1.isEqual(other.child2) and self.child2.isEqual(other.child1):
            return True
        return False

    @staticmethod
    def priority():
        return 0

class Sub:

    def __init__(self, child1, child2):
        self.child1 = child1
        self.child2 = child2

    def eval(self):
        return self.child1.eval() - self.child2.eval()

    def derivative(self, var):
        return Sub(self.child1.derivative(var), self.child2.derivative(var))

    def __str__(self):
        return toStr(self, "-")
```

```
61
62      def isEqual(self, other):
63          if not isinstance(other, Sub):
64              return False
65          return self.child1.isEqual(other.child1) and self.child2.isEqual(other.child2)
66
67      @staticmethod
68      def priority():
69          return 0
70
71  class Mult:
72
73      def __init__(self, child1, child2):
74          self.child1 = child1
75          self.child2 = child2
76
77      def eval(self):
78          return self.child1.eval() * self.child2.eval()
79
80      def derivative(self, var):
81          return Add(Mult(self.child1, self.child2.derivative(var)), Mult(self.child1.
    derivative(var), self.child2))
82
83      def __str__(self):
84          return toStr(self, "*")
85
86      def isEqual(self, other):
87          if not isinstance(other, Mult):
88              return False
89          if self.child1.isEqual(other.child1) and self.child2.isEqual(other.child2):
90              return True
91          if self.child1.isEqual(other.child2) and self.child2.isEqual(other.child1):
92              return True
93          return False
94
95      @staticmethod
96      def priority():
97          return 1
98
99  class Div:
100
101     def __init__(self, child1, child2):
102         self.child1 = child1
103         self.child2 = child2
104
105     def eval(self):
106         c2 = self.child2.eval()
107         if c2 == 0.0 or c2 == -0.0:
108             return float('NaN')
109         return self.child1.eval() / c2
110
111     def derivative(self, var):
112         num = Sub(Mult(self.child2, self.child1.derivative(var)), Mult(self.child2.
    derivative(var), self.child1))
113         return Div(num, Pow(self.child2, 2))
114
115     def __str__(self):
116         return toStr(self, "/")
117
118     def isEqual(self, other):
119         if not isinstance(other, Div):
```

```
            return False
        return self.child1.isEqual(other.child1) and self.child2.isEqual(other.child2)

    @staticmethod
    def priority():
        return 1

class Pow:

    def __init__(self, child1, value):
        self.child1 = child1
        self.value = value

    def eval(self):
        return math.pow(self.child1.eval(), self.value)

    def derivative(self, var):
        return Mult(Mult(Const(self.value), Pow(self.child1, self.value - 1)), self.
    child1.derivative(var))

    def __str__(self):
        useParens = not (isinstance(self.child1, Var) or isinstance(self.child1, Const))
        baseStr = self.child1.__str__()
        if useParens:
            baseStr = f"({baseStr})"
        return f"{baseStr} ^ {self.value}"

    def isEqual(self, other):
        if not isinstance(other, Pow):
            return False
        return self.child1.isEqual(other.child1) and self.value == other.value

    @staticmethod
    def priority():
        return 1

class Sin:

    def __init__(self, child1):
        self.child1 = child1

    def eval(self):
        return math.sin(self.child1.eval())

    def derivative(self, var):
        return Mult(Cos(self.child1), self.child1.derivative(var))

    def __str__(self):
        c1 = self.child1.__str__()
        return f"sin({c1})"

    def isEqual(self, other):
        if not isinstance(other, Sin):
            return False
        return self.child1.isEqual(other.child1)

class Cos:

    def __init__(self, child1):
        self.child1 = child1

```

```
180     def eval(self):
181         return math.cos(self.child1.eval())
182 
183     def derivative(self, var):
184         return Mult(Mult(Const(-1), Sin(self.child1)), self.child1.derivative(var))
185 
186     def __str__(self):
187         c1 = self.child1.__str__()
188         return f"cos({c1})"
189 
190     def isEqual(self, other):
191         if not isinstance(other, Cos):
192             return False
193         return self.child1.isEqual(other.child1)
194 
195 class Log10:
196 
197     def __init__(self, child1):
198         self.child1 = child1
199 
200     def eval(self):
201         return math.log(self.child1.eval()) / math.log(10)
202 
203     def derivative(self, var):
204         return Pow(Mult(self.child1, Const(math.log(10))), -1)
205 
206     def __str__(self):
207         c1 = self.child1.__str__()
208         return f"log_10({c1})"
209 
210     def isEqual(self, other):
211         if not isinstance(other, Log10):
212             return False
213         return self.child1.isEqual(other.child1)
214 
215 class Const:
216 
217     def __init__(self, value):
218         self.value = value
219 
220     def eval(self):
221         return self.value
222 
223     def derivative(self, var):
224         return Const(1) if self is var else Const(0)
225 
226     def __str__(self):
227         return f"{self.value}"
228 
229     @staticmethod
230     def priority():
231         return 3
232 
233     def isEqual(self, other):
234         if not isinstance(other, Const):
235             return False
236         return self.value == other.value
237 
238 class Var:
239 
240     def __init__(self, value, uncertainty, name):
```

```
241         self.value = value
242         self.name = name
243         self.uncertainty = uncertainty
244 
245     def eval(self):
246         return self.value
247 
248     def derivative(self, var):
249         return Const(1) if self is var else Const(0)
250 
251     def __str__(self):
252         return self.name
253 
254     def isEqual(self, other):
255         return self is other
256 
257     @staticmethod
258     def priority():
259         return 3
260 
261 # Hilfsfunktion zur Darstellung eines arithmetischen Ausdrucks als String.
262 def toStr(expr, infix):
263     c1Parens = expr.child1.priority() <= expr.priority()
264     c2Parens = expr.child2.priority() <= expr.priority()
265     c1 = expr.child1.__str__()
266     c2 = expr.child2.__str__()
267     if c1Parens:
268         c1 = f"({c1})"
269     if c2Parens:
270         c2 = f"({c2})"
271     return f"{c1} {infix} {c2}"
272     # return f"({self.child1.__str__()} + {self.child2.__str__()})"
273 
274 # Vereinfachung eines arithmetischen Ausdrucks.
275 # Haupts chlich werden Konstanten zusammengefasst bzw. entfernt
276 # (z.B. x + 0 = x, 1 * x = x, 0 * x = 0)
277 def simplify(expr):
278     match expr:
279         case Add() | Sub() | Mult() | Div():
280             expr.child1 = simplify(expr.child1)
281             expr.child2 = simplify(expr.child2)
282             if isinstance(expr.child1, Const) and isinstance(expr.child2, Const):
283                 return Const(expr.eval())
284         case Pow() | Sin() | Cos() | Log10():
285             expr.child1 = simplify(expr.child1)
286             if isinstance(expr.child1, Const):
287                 return Const(expr.eval())
288 
289     match expr:
290         case Add():
291             if isinstance(expr.child1, Const):
292                 if expr.child1.value == 0:
293                     return expr.child2
294             if isinstance(expr.child2, Const):
295                 if expr.child2.value == 0:
296                     return expr.child1
297                 if expr.child2.value < 0:
298                     return Sub(expr.child1, Const(-expr.child2.value))
299             if expr.child1.isEqual(expr.child2):
300                 return Mult(Const(2), expr.child1)
301         case Sub():
```

```
302             if isinstance(expr.child1, Const):
303                 if expr.child1.value == 0:
304                     return Mult(Const(-1), expr.child2)
305             if isinstance(expr.child2, Const):
306                 if expr.child2.value == 0:
307                     return expr.child1
308                 if expr.child2.value < 0:
309                     return Add(expr.child1, Const(-expr.child2.value))
310             if expr.child1.isEqual(expr.child2):
311                 return Const(0)
312         case Mult():
313             if isinstance(expr.child1, Const):
314                 if expr.child1.value == 0:
315                     return Const(0)
316                 if expr.child1.value == 1:
317                     return expr.child2
318             if isinstance(expr.child2, Const):
319                 if expr.child2.value == 0:
320                     return Const(0)
321                 if expr.child2.value == 1:
322                     return expr.child1
323             if expr.child1.isEqual(expr.child2):
324                 return Pow(expr.child1, 2)
325         case Div():
326             if isinstance(expr.child1, Const):
327                 if expr.child1.value == 0:
328                     return Const(0)
329                 if expr.child1.value == 1:
330                     return Pow(expr.child2, -1)
331             if isinstance(expr.child2, Const):
332                 if expr.child2.value == 1:
333                     return expr.child1
334             if expr.child1.isEqual(expr.child2):
335                 return Const(1)
336         case Pow():
337             if expr.value == 0:
338                 return Const(1)
339             if expr.value == 1:
340                 return expr.child1
341     return expr
342 
343 # Berechnung der gau 'schen Unsicherheit des Ausdrucks expr mit den Variablen
344 # params
345 def gaussian(expr, params):
346     total = 0
347     for param in params:
348         if type(param) == Const:
349             continue
350         der = simplify(expr.derivative(param))
351         derVal = der.eval()
352         delta = derVal * param.uncertainty
353         total += delta ** 2
354     return total ** 0.5
355 
356 # Berechnung der Unsicherheit des Ausdrucks expr mit der Min-Max-Methode
357 # und den Variablen params
358 def minMax(expr, params):
359     minVal = expr.eval()
360     maxVal = expr.eval()
361     incDec = [False] * len(params)
362     originalValues = []
```

```
363     for param in params:
364         originalValues.append(param.value)
365     for j in range(2 ** len(params)):
366         for i in range(len(incDec)):
367             incDec[i] = not incDec[i]
368             if incDec[i]:
369                 break
370         for i in range(len(params)):
371             sign = 1 if incDec[i] else -1
372             params[i].value = originalValues[i] + sign * params[i].uncertainty
373         newVal = expr.eval()
374         minVal = min(minVal, newVal)
375         maxVal = max(maxVal, newVal)
376     for i in range(len(params)):
377         params[i].value = originalValues[i]
378     return (maxVal - minVal) / 2
```

Output:

```
-- Output --
```