FHV - Franck-Hertz-Versuch

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/FHV verfügbar.

Inhaltsverzeichnis

Literatur

[Dem16] Wolfgang Demtröder. *Experimentalphysik 3: Atome, Moleküle und Festkörper*. Springer, 5th edition, 2016.

[Kou16] Benjamin Koubek. Entwicklung und untersuchung eines pulsgenerators zur ansteuerung dielektrisch behinderter entladungen, 2016.

1 Vobereitung

1.1 Physikalischer Hintergrund

1.1.1 Elektronenkonfiguration

Elektronen werden in einem Atom (im Schrödinger-Bild ohne Spin) durch die drei Quantenzahlen n, l und m beschrieben: n ist die Hauptquantenzahl und hat die größte Auswirkung auf die Energie des Elektrons. l wird als Drehimpulsquantenzahl bezeichnet und gibt den Bahndrehimpuls an ($L^2 = l(l+1)\hbar^2$). m ist die magnetische Quantenzahl und gibt die Komponente des Drehimpulses entlang einer beliebigen Achse (i.d.R. der z-Achse) an. Für die Wertebereiche von l und m gelten die Regeln $0 \leq l < n$ und $-l \leq m \leq l$.

Das Pauli-Prinzip besagt, dass zwei Elektronen in einem Atom nie den gleichen Satz an Quantenzahlen besitzen können. Hier muss jedoch der Spin mit berücksichtigt werden. Weil die Spin-Quantenzahl s für Elektronen stets die Werte $1/2$ oder $-1/2$ annimmt, kann also jede Kombination von Quantenzahlen (n, l, m) von höchstens zwei Elektronen besetzt werden. Diese haben dann einen entgegengesetzten Spin. In einem Atom mit Kernladungszahl $Z > 2$ befinden sich also selbst im Grundzustand des Atoms nur zwei Elektronen im niedrigsten Energiezustand. Die übrigen $Z - 2$ Elektronen verteilen sich auf höhere Energieniveaus, wobei niedrigere Energienievaus stets vor höhreren aufgefüllt werden. Die genaue Besetzung dieser Energienviveaus bezeichnet man als die Elektronenkonfiguration. In der üblichen Schreibweise wird für jede Kombination an Haupt- und Drehimpulsquantenzahl die Anzahl der Elektronen angegeben, die diese Quantenzahlen besitzen. Die Drehimpulsquantenzahl l wird dabei als Buchstabe kodiert: $0 \rightarrow$ s, $1 \rightarrow$ p, $2 \rightarrow$ d, $3 \rightarrow$ f, $4 \rightarrow$ g. Die Hauptquantenzahl n wird vor diesen Buchstaben geschrieben und die Anzahl der Elektronen als Exponent. So werden alle Elektronen in aufsteigender Energie aufgeschrieben.

1.1.2 Termschema und Grotrian-Diagramm

Für die Erklärung einiger quantenmechanischen Phänomene sind genauere Informationen über die Elektronen in einem Atom notwendig als nur die Elektronenkonfiguration. Insbesondere sind der Gesamtdrehimpuls J, welcher die Summe aus Bahn- und Spindrehimpuls ist, und die sog. Multiplizität, die die Anzahl an möglichen Spin-Projektionen auf die z-Achse angibt, relevant. Die Multiplizität berechnet sich als $(2S + 1)$, wobei S die Summe der Spin-Porjektionen der einzelnen Elektronen im Atom ist. Diese Informationen werden kompakt in der sog. Termschreibweise angegeben:

$$^{2S+1}L_J.$$

Zustände mit Multiplizität 1 werden als Singulett-Zustände bezeichnet, Zustände mit Multiplizität 2 als Dublett, mit Multiplizität 3 als Triplett, etc. Durch die Feinstruktur (s. 1.1.3) werden Zustände mit gleichen Termen aufgespalten; die Anzahl an entstehenden Linien ist durch die Multiplizität gegeben. Die Multiplizität ist außerdem deswegen relevant, weil nach den Hund'schen Regeln Terme mit höherer Multiplizität eine geringere Energie besitzen. Dadurch wird die Reihenfolge bestimmt, mit der einzelne Subschalen aufgefüllt werden: Zuerst wird jeder Wert von m von einem Elektron besetzt. Wenn alle $2l+1$ Drehimpulsprojektionen besetzt sind (und damit $2l + 1$ ungepaarte Elektronen mit gleichgeritetem Spin vorliegen) werden die magnetischen Quantenzahlen doppelt besetzt (von Elektronen mit entgegengesetztem Spin, um das Pauli-Prinzip nicht zu verletzen).

Vollständig gefüllte Subschalen müssen für die Bestimmung eines Terms nicht berücksichtigt werden, weil sie nichts zum Gesamtdrehimpuls und -Spin beitragen. Für jedes Elektron mit Quantenzahlen (n, l, m) gibt es ein Elektron mit Quantenzahlen $(n, l, -m)$ und je zwei Elektronen mit den selben Quantenzahlen haben entgegengesetzten Spin.

Verschiedene Zustände in einem Atom werden also von verschiedenen Termen beschrieben. Somit kann jedem Term ein Energieniveau zugeordnet werden. Diese Energieniveaus können in einem Grotrian-Diagramm (s. Abbildung 1) aufgetragen werden. Dabei werden Zustände mit unterschiedlicher Multiplizität (z.B. Singulett- und Triplett-Zustände) meist voneinander getrennt aufgetragen, weil insbesondere bei leichten Atomen Übergänge zwischen Zuständen unterschiedlicher Multiplizität verboten sind. Jeder Übergang entspricht einer charakteristischen Energie (und damit einer charakteristischen beobachtbaren Wellenlänge), die mithilfe von Spektroskopie gemessen werden können, wodurch Rückschlüsse über die Eigenschaften der untersuchten Probe gezogen werden können.

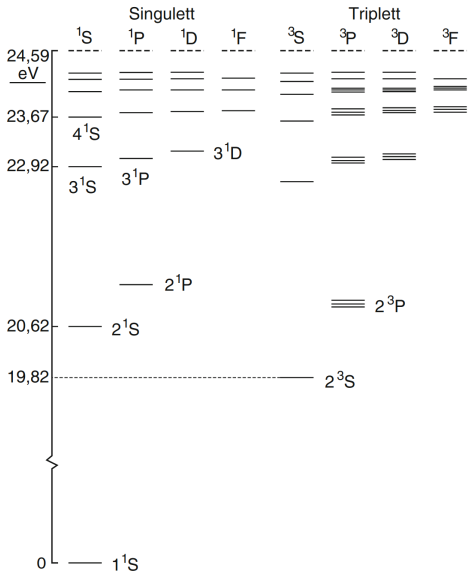

Abbildung 1: Grotrian-Diagramm für Helium. Aus [Dem16, 183]

1.1.3 LS-Kopplung

Der Spin **s** (welcher in der Schrödingergleichung nicht auftritt, sondern erst durch Berücksichtigung relativistischer Effekte aus der Dirac-Theorie stammt) verleiht den Elektronen ein kleines intrinsisches magnetisches Dipolmoment $\boldsymbol{\mu}_S = g_s(\mu_B/\hbar)\mathbf{s}$. Dabei hat g_s einen Wert von ≈ 2 und wird Landé-Faktor genannt. Dieses magnetische Moment wechselwirkt mit dem magnetischen Moment des Atomkerns, welches aus Sicht des Elektrons durch dessen Bahndrehimpuls erzeugt wird. Diese Interaktion wird als LS-Kopplung bezeichnet. Es ergibt sich eine Energiekorrektur der Terme, die proportional zum Skalarprodukt $\mathbf{L} \cdot \mathbf{s}$ des Bahn- und Spindrehimpuls ist. Weil in einem Mehrelektronensystem mit Gesamtspin S die Projektion des Spins auf die z-Achse (und damit auch das Skalaprodukt $\mathbf{L} \cdot \mathbf{S}$) genau $2S + 1$ verschiedene Werte annehmen kann, spaltet sich das Energieniveau eines Terms mit Multiplizität M in M verschiedene voneinander getrennte Niveaus auf. Terme mit gleichgerichtetem Bahndrehimpuls und Spin besitzen dabei die niedrigste Energie. Diese Aufspaltung wird als Feinstruktur bezeichnet.

[Dem16, 155–161]

1.1.4 Stoßionisation

Stößt ein Elektron mit genügend kinetischer Energie auf ein (neutrales) Atom A, so kann es passieren, dass aus A ein weiteres Elektron herausgeschlagen wird. Nach dem Stoßprozess liegt das positiv geladene Ion A^+ vor, sowie zwei freie Elektronen. Dieser Prozess kann als Reaktionsgleichung geschrieben werden:

$$A + e^- \quad \longrightarrow \quad A^+ + 2e^-.$$

Für die Energiebilanz dieser Reaktion muss gelten

$$E_{\text{kin}} + E_B \;=\; E_1 + E_2,$$

wobei E_{kin} die kinetische Energie des Elektrons vor dem Stoß ist, $E_{1/2}$ die kinetischen Energien der beiden Elektronen nach dem Stoß und E_B die Bindungsenergie des ursprünglich gebundenen Elektrons im Atom.

Die Wahrscheinlichkeit, dass ein neutrales Atom bei einem Stoß mit einem Elektron tatsächlich ionisiert wird, hängt von der Atomsorte, der kinetischen Energie des freien Elektrons und der Bindungsenergie des gebundenen Elektrons ab. Sie wird mithilfe des Ionisierungsquerschnitts σ beschrieben. σ ist abhängig von der kinetischen Energie des freien Elektrons und gibt die Querschnittsfläche um das Atom A an, durch die das freie Elektron fliegen muss.

[Dem16, 34–35]

1.1.5 Unselbstständige Dunkel- bzw. Townsend-Entladung

Die Townsend-Entladung ist eine Art der Gasentladung, bei der sich ein geladener Plattenkondensator über ein Gas, welches sich zwischen dessen Platten befindet, entladen wird. Im Regime der unselbstständigen Townsend-Entladung wird eine externe Elektronenquelle benötigt (etwa eine UV-Lampe, die durch den Photoelektrischen Effekt Elektronen an der Kathode freisetzt). Die Elektronen werden im elektrischen Feld des Kondensator richtung Anode beschleunigt. Dabei können sie auf neutrale Gasteilchen treffen und diese ionisieren, wodurch weitere Elektronen freigesetzt werden. Dadurch entsteht ein Lawinen-Effekt, weil entlang des Kondensators immer mehr Elektronen freigesetzt werden. Die dabei entstehenden positiv geladenen Ionen werden zur Kathode hin beschleunigt und können beim Auftreffen auf dieser weitere Elektronen freisetzen. Wenn auf diese Art und Weise genügend freie Elektronen an der Kathode entstehen, um den Prozess von alleine aufrecht zu erhalten, spricht man von der selbstständigen Townsend-Entladung.

[Kou16, 18–21]

1.1.6 Erklärung des 1. und 2. Townsend-Koeffizienten

Der Strom, der bei der Townsend-Entladung zwischen den Platten des Kondensators fließt, kann mit der Formel

$$\frac{I}{I_0} \;=\; \frac{e^{\alpha d}}{1 - \lambda(e^{\alpha d} - 1)}$$

berechnet werden.

[Kou16, 18–21]

1.2 Aufgabe zur 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/FHV) 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 --
```