ROE - Röntgenstrahlung: Braggreflexion und Röntgenfluoreszenzanalyse

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

1 Vobereitung

1.1 Physikalischer Hintergrund

1.1.1 Röntgenröhre

[?]

Röntgenstrahlung entsteht durch die Geschewindigkeitsänderung geladener Teilchen. Dabei wird die Röntgen-strahlung durch das Abbremsen energiereicher Elektronen in einer Röntgenröhre erzeugt.

Folgend ist der Aufbau einer Röntgenröhre skizzenhaft dargestellt:

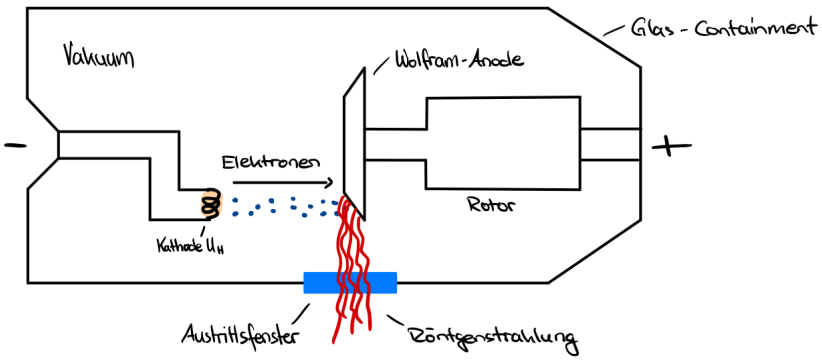

Abbildung 1: Skizzenhafte Röntgenröhre

Durch das anlegen einer Heizspannung von einigen Volt an der Kathode, treten, bedingt durch den thermoelektrischen Effekt, schwach, oder gar nicht gebundene Elektronen aus der Heizwendel aus. Durch die angelegte Anodenspannung werden die Elktronen von der Kathode zur Anode beschleunigt und treffen anschließend auf die Anode.

Durch die Bremsstrahlung und durch die Wechselwirkungen untern den einzelnen Elektronen kommt es zur Entstehung von Röntgenstrahlung. Deren Überlagerung bildt das emittierte Röntgenspektrum. Die Eigenschaften und die Entstehung von Bremsstrahlung und der charakteristischen Strahlung werden im Kapitel 1.1.3 genauer beleuchtet.

1.1.2 Linienspektrum eines Stoffe, Schalenmodell, mögliche Übergänge (Quantenmechanik)

[?] [?, 1050–1051] [?]

Ein Linienspektrum ist ein Strahlungsspektrum, welches voneinander getrennte, diskrete Linien zeigt. Das können zum Beispiel Absoroptions- oder Emissionlinien in Lichtspektren sein. Jedoch weisen auch manche Teilchenstrahlungen, wie die Alphastrahlung Lininespektren auf. Daraus kann man ableiten, dass auch Teilchen diskrete kinetische Energien haben.

Jedes Material (Atom, Molekül) hat dabei chrakteristische, diskrete Energieniveaus, auf welchen sich die Elektronen befinden können. Der Wechsel von einem Energienievau auf ein anderes erfolgt dabei durch Aufnahme/Abgabe eines Pohotons. Aus der Energiedifferenz lässt sich dann die zugehörige Wellenlänge über folgende Formel bestimmen: $\lambda = \frac{c}{v}$.

Ein angeregtes Atom oder Molekül befindet sich immer nur sehr kurz in seinem angeregten Zustand. Es fällt nach einer ehr kurzen Zeitspanne wieder in einen tieferen Energiezustand zurück. Die ausge-

sandten Photonen erscheinen dann mit einer ganz bestimmten Energie als Emissionslienien auf einem Spektrum. So entstehen chrakteristische Spektren, wie zum Beispiel das Spektrum von Wasserstoff oder Helium.

Das Schalenmodell ist ein Modell, um den Aufbau von Atomen zu beschreiben. Das Schalenmodell basiert hierbei auf dem Bohr'schen Atommodell. Im Atomkern befinden sich die Protonen und Neutronen. Die negativ geladenen Elektronen bewegen sich in Schalen um den positiv gelandenen Kern. Folgende Skizze soll das Schalenmodll verdeutlichen:

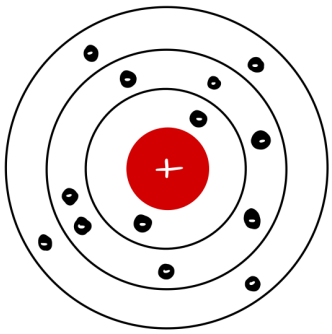

Abbildung 2: Schalenmodell

Die Schalen haben, wie in der Abbildung ersichtlich, unterschiedlich Abstände zum Kern. Die Schalen werden von innen nach außen mit unterschiedlichen Buchstaben bezeichnet, beginnend mit der "KSchalge, LSchale und MSchale. Dabei haben die Schalen unterschiedlich viel Platz für Elektronen Über die Formel $e = 2n^2$ lässt sich die maximale Anzahl an Elektronen für jede Schale bestimmen.

Jedoch hat das Schalenmodell auch Grenzen. Will man Elemente mir mehr als 20 Elektronen beschreiben, so ist das Schalenmodell zur Beschreibung nicht mehr geeignet!

Der Zusammenhang mit der Quantenmechanik ist nun leicht hergestellt. Die Quantenmechanik besagt, dass ein Elektron in einem Atom nur auf bestimmen Energieniveaus existieren kann und nicht zwischen den Niveaus! Wie oben bereits beschrieben können also folgende Aussagen getroffen werden:

1. Abstieg: ein Photon wird emittiert (von E_2 nach E_1)
2. Aufstieg: ein Photon wird absorbiert

Die Energie des Photons wird also genau durch den Energieunterschied der beiden Zustände beschrieben:

$$E_{Photon} = E_{oben} - E_{unten} = h * f = \frac{h*c}{\lambda}$$

mit h als Plancksches Wirkungsquantum, f als Frequenz, λ als Wellenlänge und c als Lichtgeschwindigkeit.

Folgende Aussagen lassen sich also zusammenfassend treffen:

1. Die Energiedifferenzen werden durch Linienspektren beschrieben
2. Die Energiedifferenzen entstehen durch Übergänge von Elektronen zwischen den unterschiedglichen Energieniveaus
3. Jedem Übergang ist eine chrakteristische Linie zugeordnet

4. Jedes Atom wird durch ein charakteristisches Linienspektrum beschrieben

Beispiel Wasserstoffatom:

- Lyman-Serie: Übergang nach $n = 1$
- Balmer-Serie: Übergang nach $n = 2$
- Paschen-Serie: Übergang nach $n = 3$
- Bracket-Serie: Übergang nach $n = 4$
- Pfund-Serie: Übergang nach $n = 5$

Hier ist wohl die Balmer-Serie die bekannteste Serie, da hier das Linienspektrum mit dem bloßen Auge sichtbar ist.

1.1.3 Röntgenstrahlung (Bremsstrahlung, chrakteristische Strahlung)

[**?**] [1010–1011; 1384] [**?**] [**?**]

Wie bereits in Kapitel 1.1.1 erwähnt, wird in diesem Kapitel die Bremsstrahlung und die charakteristische Strahlung genauer betrachtet.

Die Kathode in der Röntgenröhre beschleunigt die Elektronen auf eine Geschewindigkeit von circa $0,35c$. Die ausgetretenen Elektronen treffen nun mit dieser hohen Geschwindigkeit auf die Anode in der Röntgenröhre. Durch das extreme Abbremsen der Elektronen, entsteht elektrische Strahlung, die sogenannte Bremsstrahlung. Je stärker die Elektronen abgebremst werden, umso mehr Strahlung senden diese aus. Da nicht alle Elektronen gleich stark abgebremst werden, besitzen die Photonen der Bremsstrahlung unterschiedliche Wellenlängen. Dadurch ist das Spektrum der Bremsstrahlung ein kontinuierliches Spektrum. Die minimale Wellenlänge wird häufig als λgr bezeichnet. Die minimale Wellenlänge lässt sich durch folgende Formel bestimmen:

$$\lambda_{gr} = \frac{hc}{eU}$$

wobei h das Plancksche Wirkungsquantum, c die Lichtgeschwindigkeit, e die Elementarladung und U die Beschleunigungsspannung ist.

Aus der Formel lässt sich ableiten, dass die minimale Wellenlänge und die Beschleunigungsspannung indirekt proportional zueinander sind. Mit steigender Beschleunigungsspannung sinkt die minimale Wellenlänge.

Abschließend lässt sich also zur Bremsstrahlung folgendes zusammenfassen: das kontinuierliche Spektrum entsteht durch das Abbremsen der ausgesandeten, beschleunigten Elektronen der Kathode. Durch das unterschiedlich starke Abbremsen der Elektronen werden Photonen mit unterschiedlichen Energiewerten entsandt, wobei die minimale Wellenlänge/die maximale Photonenenergie von der Beschleunigungsspannung der Röntgenröhre abhängen.

Neben der Bremsstrahlung, tritt bei ausreichend großer Beschleunigungsspannung auch ein charakteristisches Linienspektrum auf. Diese Röntgenstralhung is jedoch nur bei höheren Ordnungszahlen auf. Das liegt daran, dass Atome mit höherer Ordnungszahl zahlreiche Atome in den äußeren Elektronenschalen haben. Diese sind nicht so stark an den Atomkern gebunden und können somit leichter austreten als Elektronen, welche sich nahe dem Kern befinden. Im folgenden soll die Entstehung der chrakteristischen Stralung stichpunktartig beschrieben werden:

1. Ein Anode-Atom wird durch ein sehr schnelles Elektron angeregt. Ein Elektron wird folgich auch ein noch freies, höheres, Energieniveau angehoben. Auf einer der unteren Schalen entsteht somit eine Lücke. Wie bekannt, gehen angeregt Atome nach kurzer Zeit in energetisch günstigere Zustände über. Folglich kann folgendes passieren:

2. Möglichkeit 1: Ein Elektron der höchsten Schale fällt nun wieder zurück auf die Lücke der untersten Schale. Dabei wird ein Photon emittiert.

3. Möglichkeit 2: Die Lücke der untersten Schale wird durch ein Elektron auf einer höheren Schale aufgefüllt. Hierbei wird ein Photon emittiert. Die so enstandene Lücke in der höheren Schale wird erneut durch ein Elektron einer Schale darüber aufgefüllt. Es wird wieder ein Photon emittiert. Dieser Vorgang läuft so lange ab, bis wieder alle Schalen aufgefüllt sind, bis auf die äußerste Schale.

Zusammenfassend lässt sich also sagen, dass das Röntgen-Spektrum durch charakteristische Strahlung identifiziert werden kann. Elektronenübergänge zwischen den unterschiedlichen Schalen der Atome sind die Ursache für das Auftreten der charakteristischen Strahlung.

1.1.4 Zustandekommen der charakteristischen Strahlung im Röntgenspektrum (K_α, K_β)

In Kapitel davor schon beschrieben. Schauen, wie man das in diesem Kapitel noch einbauen kann. Evtl Skizze und Verweis nach oben???? Übernahme der Stichpunkte nach unten?!

1.2 Aufgaben aus dem Text

1.2.1 Teile der jeweiligen Aufgaben

[...]

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 3: 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 4: Versuchsskizze Teilversuch 2

(IV) Planung der Durchführung

- eins
- zwei

(V) Vorüberlegungen zur Durchführung & Auswertung

- eins
- zwei

2.4 Teilversuch 3: Duane-Huntsches Verschiebungsgesetz

(I) Ziel: ...

(II) Versuchsmethode: ...

(III) Versuchsskizze:

Abbildung 5: Versuchsskizze Teilversuch 3

(IV) Planung der Durchführung

- eins
- zwei

(V) Vorüberlegungen zur Durchführung & Auswertung

- eins
- zwei

2.5 Teilversuch 4: Röntgenfluoreszenzanalyse

(I) Ziel: ...

(II) Versuchsmethode: ...

(III) Versuchsskizze:

Abbildung 6: Versuchsskizze Teilversuch 4

(IV) Planung der Durchführung

- eins
- zwei

(V) Vorüberlegungen zur Durchführung & Auswertung

- eins
- zwei

2.6 Teilversuch 5: Identifiaktion einer unbekannten Probe

(I) Ziel: ...

(II) Versuchsmethode: ...

(III) Versuchsskizze:

Abbildung 7: Versuchsskizze Teilversuch 5

(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

4.2 Teilversuch 2: Energiespektrum einer Röntgenröhre in Abhängigkeit der Spannung

4.3 Teilversuch 3: Duane-Huntsches Verschiebungsgesetz

4.4 Teilversuch 4: Röntgenfluoreszenzanalyse

4.5 Teilversuch 5: Identifiaktion einer unbekannten Probe

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

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, "-")

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

    @staticmethod
    def priority():
        return 0

class Mult:

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

```

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

```
120     def derivative(self, var):
121         return Mult(Mult(Const(self.value), Pow(self.child1, self.value - 1)), self.
    child1.derivative(var))
122 
123     def __str__(self):
124         useParens = not (isinstance(self.child1, Var) or isinstance(self.child1, Const))
125         baseStr = self.child1.__str__()
126         if useParens:
127             baseStr = f"({baseStr})"
128         return f"{baseStr} ^ {self.value}"
129 
130     def isEqual(self, other):
131         if not isinstance(other, Pow):
132             return False
133         return self.child1.isEqual(other.child1) and self.value == other.value
134 
135     @staticmethod
136     def priority():
137         return 1
138 
139 class Sin:
140 
141     def __init__(self, child1):
142         self.child1 = child1
143 
144     def eval(self):
145         return math.sin(self.child1.eval())
146 
147     def derivative(self, var):
148         return Mult(Cos(self.child1), self.child1.derivative(var))
149 
150     def __str__(self):
151         c1 = self.child1.__str__()
152         return f"sin({c1})"
153 
154     def isEqual(self, other):
155         if not isinstance(other, Sin):
156             return False
157         return self.child1.isEqual(other.child1)
158 
159 class Cos:
160 
161     def __init__(self, child1):
162         self.child1 = child1
163 
164     def eval(self):
165         return math.cos(self.child1.eval())
166 
167     def derivative(self, var):
168         return Mult(Mult(Const(-1), Sin(self.child1)), self.child1.derivative(var))
169 
170     def __str__(self):
171         c1 = self.child1.__str__()
172         return f"cos({c1})"
173 
174     def isEqual(self, other):
175         if not isinstance(other, Cos):
176             return False
177         return self.child1.isEqual(other.child1)
178 
179 class Log10:
```

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

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

```
                    return expr.child1
            if expr.child1.isEqual(expr.child2):
                return Pow(expr.child1, 2)
        case Div():
            if isinstance(expr.child1, Const):
                if expr.child1.value == 0:
                    return Const(0)
                if expr.child1.value == 1:
                    return Pow(expr.child2, -1)
            if isinstance(expr.child2, Const):
                if expr.child2.value == 1:
                    return expr.child1
            if expr.child1.isEqual(expr.child2):
                return Const(1)
        case Pow():
            if expr.value == 0:
                return Const(1)
            if expr.value == 1:
                return expr.child1
    return expr

# Calculate the gaussian uncertainty of expr
def gaussian(expr, params):
    total = 0
    for param in params:
        if type(param) == Const:
            continue
        der = simplify(expr.derivative(param))
        derVal = der.eval()
        delta = derVal * param.uncertainty
        total += delta ** 2
    return total ** 0.5

# Calculate the uncertainty of expr using the min-max method
def minMax(expr, params):
    minVal = expr.eval()
    maxVal = expr.eval()
    incDec = [False] * len(params)
    originalValues = []
    for param in params:
        originalValues.append(param.value)
    for j in range(2 ** len(params)):
        for i in range(len(incDec)):
            incDec[i] = not incDec[i]
            if incDec[i]:
                break
        for i in range(len(params)):
            sign = 1 if incDec[i] else -1
            params[i].value = originalValues[i] + sign * params[i].uncertainty
        newVal = expr.eval()
        minVal = min(minVal, newVal)
        maxVal = max(maxVal, newVal)
    for i in range(len(params)):
        params[i].value = originalValues[i]
    return (maxVal - minVal) / 2
```

Output:

```
-- Output --
```