```
# Sample Code zur Vorlesung Informatik für Nanowissenschaftler an der Universität Hamburg WS 20/21
# Autor: Prof. Dr. Rübhausen
# Dieser Code dient zu Illustration der Vorlesung. Schreiben Sie ihren eigenen Code für ihre Projekte - bitte kein "Copy & Paste" sie reduzieren
# sonst ihren eigenen Lerneffekt und schaden sich selbst


import matplotlib.pyplot as plt # Erzeugt eine Instanz von Pyplot
import numpy as np # Erzeugt eine Instanz von Numpy


# numerische Lösungs-Prozeduren
def Euler_DGL(K,U0,Z0,Delta_X,N):# Definiere eine Prozedur um mit dem Euler-Verfahren eine Differentialgleichung 2ter Ordnung auszurechnen. Die Parameter der Prozedur ... K = omega^2
    Werte_U = []#Liste mit Werten der Lösung der DGL
    Werte_Z = []# List mit Werten der Ableitung der Lösung z.b. bei einem Federpendel Ort und Geschwindigkeit
    Werte_U.append(U0) # Trägt den Start von U in den Listenplatz "Null" ein
    Werte_Z.append(Z0) # Trägt den Start von Z in den Listenplatz "Null ein
    h = Delta_X/N # Schrittweite
    print("Schrittweite")#Ausgabe
    print(h)
    for i in range(1,N):#Schleife zur Berechnung
        Werte_U.append(Werte_Z[i-1]*h+Werte_U[i-1])#Euler DGL1 für DGL 1 Ordnung
        Werte_Z.append((-K)*Werte_U[i-1]*h+Werte_Z[i-1])#Euler DGL 2 für DGL 1 Ordnung
    return Werte_U#Rückgabe der Lösung

def RK2_DGL(K,U0,Z0,Delta_X,N):# Definiere eine Prozedur um mit dem Runge-Kutta 2O-Verfahren eine Differentialgleichung 2ter Ordnung auszurechnen. Die Parameter der Prozedur ... K = omega^2
    Werte_U = []#Analog zu Euler
    Werte_Z = []#s.o.
    Werte_U.append(U0) # Trägt den Start von U in den Listenplatz "Null" ein
    Werte_Z.append(Z0)#s.o.
    h = Delta_X/N#s.o.
    print("Schrittweite")
    print(h)
    for i in range(1,N):
        Werte_U.append((Werte_Z[i-1]+0.5*h*Werte_Z[i-1])*h+Werte_U[i-1])#s.o. allerdings mit Zusatzfunktion um die Steigung im Mittelpunkt zu berücksichtigen - daher Korrektur zu Euler
        Werte_Z.append((-K)*(Werte_U[i-1]+0.5*h*Werte_U[i-1])*h+Werte_Z[i-1])#s.o. allerdings mit Zusatzfunktion um die Steigung im Mittelpunkt zu berücksichtigen - daher Korrektur zu Euler
    return Werte_U#s.o.

def Num_DGL(K,U0,U1,Delta_X,N):# Definiere eine Prozedur um mit dem Numerov - Verfahren eine Differentialgleichung 2ter Ordnung auszurechnen. Die Parameter der Prozedur ... K = omega^2
    Werte_U = []#s.o.
    Werte_U.append(U0) # Trägt den Start von U in den Listenplatz "Null" ein
    Werte_U.append(U1) # Wir benötign den Wert x-h und x um x+h abzuschätzen
    h = Delta_X/N
    print("Schrittweite")
    print(h)
    for i in range(2,N):
        Werte_U.append(((2 * Werte_U[i - 1] * (1 - 5 / 12 * h * h * K)) - Werte_U[i - 2] * (
```

```
                    1 + h * h * K / 12)) / (1 + 1 / 12 * h * h * K))  # Numerov
Approximation mit konstantem K
    return Werte_U

# Definition einer Potentialwandlandschaft - Tiefe V_0, (eV) Position x_0 (nm) und
Breite L (nm) - Delta_X ist der Bereich indem gerechnet wird (nm), N = Anzahl der
Schritte in x

def Potential_Landschaft(V0,X0,L,Delta_X,N):
    Werte_Potential = []
    X_Achse = []
    h = Delta_X / N # Auflösung in nm /schritt
    print("Auflösung in nm")
    print(h)
    Schritte_bis_X0 = round(X0/h)#Ganze Zahl für Schleife
    print("Schritte bis X0")
    print(Schritte_bis_X0)
    Schritte_ohne_Potential = round(L/h)#Ganze Zahl für Schleife
    print("Schritte zwischen den Wänden")
    print(Schritte_ohne_Potential)
    Schritte_nach_Potential = N-Schritte_bis_X0 - Schritte_ohne_Potential #
Brauchen wir nicht

    for i in range(N):
        X_Achse.append(i*h)#Erzeugung der x-Achse in nm

    for i in range(Schritte_bis_X0):#Bis X0 gibt es eine Potentialwand
        Werte_Potential.append(V0)
    for i in range(Schritte_bis_X0,(Schritte_bis_X0 +
Schritte_ohne_Potential)):#Der Bereich ab X0 im Bereich L ist potentialfrei
        Werte_Potential.append(0)
    for i in range((Schritte_bis_X0 + Schritte_ohne_Potential),N):#Danach gibt es
die rechte Potentialwand
        Werte_Potential.append(V0)

    return Werte_Potential, X_Achse#Rückgabe des Potentials (eV) und der X-Achse
(nm)

# jetzt müssen wir noch die richtige K_List berechnen - dass ist der Omega^2
Vorfaktor und er hängt von der Energie des Teilchens ab

def K_list_gen(Potential,Energie,N):
    K_List = []
    Faktor = 26.27#Umrechnungsfaktor für eV und nm
    for i in range(N):
        K_List.append(Faktor * (Energie-Potential[i]))#Die Energie des Teilchens
ist fest aber die Potential-Landschaft ändert sich mit der Position in X
    return K_List # Rückgabe des ortsabhängigen K

def Wavefunction(K,U0,U1,Delta_X,N):#Berechnung der Wellenfunktion mit
ortsabhängigem K
    Werte_U = []
    Werte_U.append(U0)  # Trägt den Start von U in den Listenplatz "Null" ein
    Werte_U.append(U1)  # Wir benötign den Wert x-h und x um x+h abzuschätzen
    h = Delta_X / N
   # print("Schrittweite")
   # print(h)
    for i in range(2, N):
        Werte_U.append(((2 * Werte_U[i - 1] * (1 - 5 / 12 * h * h * K[i-1])) -
Werte_U[i - 2] * (
                1 + h * h * K[i-2] / 12)) / (1 + 1 / 12 * h * h * K[i]))  #
```

```
Numerov mit K(x) vergleiche mit Numerov oben

    return Werte_U

def Nullstellensuche_Simple(U0,U1,Delta_X,N,Potential,E_0,E_max,Delta_E): #
Anmerkung - zunächst die Numerov-relevanten Variablen, dann die notwendige
Information für das Potential und die aus der Energie zu berechnende K_List
    Energie_Eigenwerte=[]
    Last_Val = []
    N_E = round((E_max-E_0)/Delta_E)# Energie-Schritte bestimmen
    print("Eigenwertsuche in äquidistanten Energieschritten von (eV)")
    print(Delta_E)
    print("Anzahl der Schritte: " , N_E)
    for i in range(N_E):
        Energie = i*Delta_E # jeder Energiewert wird gesetzt
        K=K_list_gen(Potential,Energie,N) # für jeden Energiewert wird mit dem für
alle Werte konstaten Potential die K_list berechnet (unser omega^2(x))
        wave = Wavefunction(K,U0,U1,Delta_X,N) # jetzt wird damit die
Wellenfunktion berechnet
        Last_Val.append(wave[N-1]) # die letzten Werte der jeweiligen
Wellenfunktion werden in die Liste Last_Val[i] geschrieben - diese wird dann auf
Vorzeichenwechsel untersucht
    for i in range(1,N_E): # wir durchsuchen jetzt die liste nach
Vorzeichenwechsel
        if np.sign(Last_Val[i-1]) != np.sign(Last_Val[i]): # gibt es einen
Vorzeichenwechsel dann
            print("Nullstelle gefunden bei :") # informiere mich
            E_est = i*Delta_E-0.5*Delta_E # Schätze den Wert aus den bekannten
Werte ab
            print(E_est) # Ausgeben
            Energie_Eigenwerte.append(E_est) # Eintragen
    return Energie_Eigenwerte, Last_Val  # Liste mit Energieeigenwerten zurück
geben

def Nullstellensuche_NR(Nullstelle,dN_Start,U0,U1,Potential,Delta_X,N,Max_Number):
    Energie = Nullstelle
    dE = dN_Start
    for i in range(Max_Number):  # hier kommt jetzt das Optimierungsverfahren
(Newton-Raphson) - allerdings mit einer maximalen Anzahl an Schritten oder einem
Abbruch bei Limit = Genauigkeit
        K = K_list_gen(Potential, Energie,N)  # für jeden Energiewert wird mit dem
für alle Werte konstaten Potential die K_list berechnet (unser omega^2(x)), da der
Energiewertgeändert wird muss die Berechnung in der Optimierungsschleife angepasst
werden
        wave_1 = Wavefunction(K, U0, U1, Delta_X, N)  # wir rechnen die
Wellenfunktion für E aus
        wave_max_E = wave_1[N - 1]  # wir bestimmen den Wert psi(xmax,E)
        print("Ergebnis von Wave_max_E", wave_max_E)
        K = K_list_gen(Potential, (Energie + dE), N)  # wir berechnen die K_list
für E+dE
        wave_2 = Wavefunction(K, U0, U1, Delta_X, N)  # Wir berechen die
Wellenfunktion für E+dE
        wave_max_E_dE = wave_2[N - 1]  # wir berechnen den Wert psi(xmax,E+dE)
        print("Ergebnis von Wave_max_E+dE", wave_max_E_dE)
        dE_neu = - wave_max_E / ((wave_max_E_dE - wave_max_E) / dE)  # Berechnung
der Steigung
        print("Neues dE", dE_neu)
        Energie_neu = Energie + dE_neu  # Berechnung des neuen Energie_Eigenwerts
        print("Energie_neu", Energie_neu)
        dE = dE_neu  # Anpassung des Schritts in dE
        ratio = Energie_neu/Energie
```

```
        print("ratio", round(ratio,5))
        Energie = Energie_neu
        if (1.00001 > ratio > 0.999999):#Hier überprüfen wir auf die Genauigkeit
und fangen ein Teilen durch Null ab
            print("Auflösung nach Schritten erreicht", (i+1))
            break



    print("Energieeigenwert nach Optimierung ist : ", Energie_neu)  # das beste
Ergebnis wird ausgegeben
    Nullstelle_Opt = Energie_neu
    return Nullstelle_Opt

def Nullstellensuche_NR_Liste(Energies,U0,U1,Delta_X,N,Potential,Max_Number): #
Anmerkung wir haben die Energieeigenwerte z.B. über die Nullstellensuche_Simple
abgeschätzt jetzt wollen wir diese Werte verfeinern
    Anzahl_Energiewert = len(Energies)#Also wieviele Energieeigenwerte gibt es ?
    print("Anzahl der Eigenwerte", Anzahl_Energiewert)
    Eigenwerte_NR = [] # Liste mit den Energieeigenwerten nach Optimierung

    for n in range(Anzahl_Energiewert):#Schleife über die Energie-Eigenwerte
        print("Optimierung des geschätzten Energie-Eigenwerts")
        Energie = Energies[n] # abgeschätzter Energieeigenwert der Input
        print("Energie: ", Energie)
        dE = Energie/100 #kleiner initialer Schritt um den abgeschätzten Energie-
Eigenwert
        #print("dE: ", dE)
        Energie_neu =
Nullstellensuche_NR(Energie,dE,U0,U1,Potential,Delta_X,N,Max_Number)
        Eigenwerte_NR.append(Energie)  # das beste Ergebnis wird in die
Ergebnisliste eingetragen

    return Eigenwerte_NR

def Berechnung_der_Normierten_Wellenfunktion(Eigenwert,U0,U1,Potential,Delta_X,N):
    Norm_Wave =[]
    frequency = K_list_gen(Potential,Eigenwert,N)
    Eigen_Wave = Wavefunction(frequency,U0,U1,Delta_X,N)
    # Wir haben jetzt eine nicht-normierte Wellenfunktion zum Energie-Eigenwert
ausgerechnet
    # Wir müssen jetzt die Normierung ausrechnen
    Summe = 0

    for i in range(N):#Summe = Integral
        Summe = Eigen_Wave[i]**2+Summe
    print("Normierung",Summe)
    for i in range(N):
        Norm_Wave.append((Eigen_Wave[i]/np.sqrt(Summe))) #Normierung der Werte

    return Norm_Wave




# Parameter - global
Num_0 = 0
Num_1 = 0.000001
Range = 15 # in x von 0 bis 15 nm
Position = 2.5 # Position linke Wand
```

```
Length = 10 # Ohne Potential
Steps = 1000 # Schritte
Potential_Wert = 1 #  Potential in eV
E_min = 0  #  Minimum  Energie (electron)
E_max = 1 # Maximum Energie
Delta_E = 0.001 # Energieauflösung
# 1 eV (compare to 0.00375637 (inf) / 0.0034764667 (n=1) / 0.0139058668 n=2 (nom.)
0.013904041
# 2 eV 0.0035542320205
#10 eV 0.003661893614476034774021795392684452963294461369514 48

# 1eV , 5 nm l^2= 0.0139058668, tat.: E = 0.012949909855417594297


Pot, X =Potential_Landschaft(Potential_Wert,Position,Length,Range,Steps)
#Nullstellensuche_NR(0.0035,0.000035,Num_0,Num_1,Pot,Range,Steps,10)

Eigenwerte, Werte_Wellenfunktion =
Nullstellensuche_Simple(Num_0,Num_1,Range,Steps,Pot,E_min,E_max,Delta_E)
Bessere_Eigenwerte =
Nullstellensuche_NR_Liste(Eigenwerte,Num_0,Num_1,Range,Steps,Pot,10)

Wave_1 =
Berechnung_der_Normierten_Wellenfunktion(0.00347646668802934,Num_0,Num_1,Pot,Range
,Steps)
Wave_2 =
Berechnung_der_Normierten_Wellenfunktion(0.013904041005392142,Num_0,Num_1,Pot,Rang
e,Steps)
Wave_3 =
Berechnung_der_Normierten_Wellenfunktion(0.03127716244972,Num_0,Num_1,Pot,Range,St
eps)

#Frequency = K_list_gen(Pot,0.12495122725906137,Steps)
#Wave = Wavefunction(Frequency,0,0.000001,Range,Steps)

#plt.plot(Werte_Wellenfunktion)
#plt.show()

# Darstellung Potential und K_list / Wellenfunktion auf 2 Y-Achsen und einer X-
Achse # ihr könnt das aber auch ganz anders lösen

fig, ax1 = plt.subplots()

color = "tab:red"
ax1.set_xlabel("X-Axis (nm)")
ax1.set_ylabel("Potential (eV)")
ax1.plot(X, Pot, color = color)
ax1.tick_params(axis='y', labelcolor = color)


ax2 = ax1.twinx()

color = 'tab:blue'
ax2.set_ylabel("Psi", color = color)
#ax2.plot(X, Wave, color = color)
ax2.plot(X, Wave_1,Color = color)
ax2.plot(X, Wave_2, color=color)
ax2.plot(X, Wave_3, color = color)
ax2.tick_params(axis='y', labelcolor =color)

fig.tight_layout()
```

```
#plt.xlim(1,14) #Achsen einschränken
#plt.ylim(-50,50)

plt.show()
```