Abgabedatum: 19. Juli 2017 Erik Teichmann

Computational Physics - Schrödinger Gleichung

1 Einleitung

Das Ziel dieses Versuches ist es, die Schrödinger-Gleichung in einem Doppelmulden-Potential zu simulieren. Dazu müssen die Eigenwerte der stationären Gleichung numerisch gefunden werden. Anschließend kann die zeitliche Entwicklung der Gleichung untersucht werden und die Richtigkeit anhand der Stationarität der Eigenfunktionen verifiziert werden.

2 Methodik

Die zeitabhängige, dimensionslose Schrödinger-Gleichung ist durch die folgende Gleichung gegeben.

$$\hat{H}\Psi(x,t) = i\frac{\partial\Psi(x,t)}{\partial t} = \left(-\frac{\partial^2}{\partial x^2} + V(x,t)\right)\Psi(x,t) \tag{1}$$

Dabei ist $V(x,t)$ das Potential und $\Psi(x,t)$ die Wellenfunktion. Das verwendete Doppelmulden-Potential hat die Form $V(x,t) = A/2(1-x^2)^2$. Anhand der Gleichung erkennt man schnell, dass es genau zwei Nulstellen bei ± 1 gibt. Diese sind unabhängig von dem Skalenparameter A. Im folgenden wurde $A = 1$ gewählt. Der Parameter bestimmt, wie stark das Potential ansteigt.

2.1 Eigenwerte und Eigenfunktionen der stationären Schrödingergleichung

Zuerst wird die stationäre Schrödinger-Gleichung numerisch gelöst. Sie hat die Form

$$E\Psi(x) = \left(-\frac{\partial^2}{\partial x^2} + V(x)\right)\Psi(x) \tag{2}$$

E ist der Eigenwert des Hamilton-Operators $\hat{H}$ und $\Psi(x)$ die zugehörige Eigenfunktion. Die Schwierigkeit bei der Lösung liegt darin, dass die Gleichung durch die Randbedingungen $\Psi(\pm\infty) = 0$ bestimmt wird.
Die Randbedingungen können durch geeigente Annahmen in zwei Anfangsbedingungen zerlegt werden: durch das hohe Potential kann für große $|x|$ ein Verhalten von $\Psi \propto e^{-\sqrt{E}|x|}$ angenommen werden. Es gibt also zwei Funktionen, eine linke Ψ_- und eine recht Ψ_+. Das heißt die Lösungen sollten für ein großes x_0 die folgende Form haben:

$$\Psi_-(x_0) = Ce^{\sqrt{E}x_0} \tag{3}$$

$$\Psi_+(x_0) = Ce^{-\sqrt{E}x_0} \tag{4}$$

Mit zwei Stetigkeitsbedingung

$$\Psi_+(0) = \Psi_-(0) \tag{5}$$

$$\Psi'_+(0) = \Psi'_-(0) \tag{6}$$

Um die Differentialgleichung numerisch zu lösen werden zwei Differentialgleichungen, eine von links zu 0 kommende mit Ψ_- und eine von rechts zu 0 kommende mit Ψ_+ gesucht, die die

Stetigkeitsbedingung erfüllen. Um die Suche nach den Eigenwerten zu vereinfachen kann die Funktion

$$w(E) = \Psi_+(0)\Psi'_-(0) - \Psi_-(0)\Psi'_+(0) \stackrel{!}{=} 0 \tag{7}$$

verwendet werden. Die Nullstellen dieser Funktion entsprechen den Eigenwerten. Es muss darauf geachtet werden, dass die Wahl der Anfangsbedingungen immer zu einer symmetrischen Lösung führt, um antisymmetrische Lösung zu erhalten muss für die entsprechnden Eigenwerte eine der beiden Hälften mit -1 multipliziert werden.
Zur Lösung der Differentialgleichungen für Ψ_+ und Ψ_- können typische Verfahren verwendet werden. Hier wurde ein Runge-Kutta-Verfahren der Ordnung 4 verwendet. Zur Bestimmung der Nullstellen wurde ein einfaches Bisektionsverfahren verwendet.

2.2 Zeitabhängige Schrödinger-Gleichung

Zur Bestimmung der zeitabhängigen Lösungen wird die Gleichung (1) verwendet. Um die Genauigkeit zu erhöhen und die Wahrscheinlichkeit $|\Psi|^2$ über die gesamte Zeit zu erhalten wird das Crank-Nicolson-Verfahren verwendet. Es besteht aus einer Mischung von Euler-Verfahren, allgemein lautet die Formel:

$$\frac{\partial y}{\partial t} = f\left(y, x, t, \frac{\partial y}{\partial x}, \frac{\partial^2 y}{\partial x^2}\right) \tag{8}$$

$$\frac{y_j^{n+1} - y_j^n}{\Delta t} = \frac{1}{2}\left[f_j^{n+1}\left(y, x, t, \frac{\partial y}{\partial x}, \frac{\partial^2 y}{\partial x^2}\right) + f_j^n\left(y, x, t, \frac{\partial y}{\partial x}, \frac{\partial^2 y}{\partial x^2}\right)\right] \tag{9}$$

Angewendet auf die Schrödinger-Gleichung ergibt sich nach einer kurzen Rechnung die folgende Gleichung:

$$\left(\mathbb{1} + \frac{1}{2}i\Delta t\hat{H}\right)\Psi^{n+1} = \left(\mathbb{1} - \frac{1}{2}i\Delta t\hat{H}\right)\Psi^n \tag{10}$$

Die rechte Seite ist dabei bekannt und $\hat{H}$ kann nach der Diskretisierung von x als Matrix mit Einträgen auf den drei Diagonalen geschrieben werden.

$$\hat{H} = \begin{bmatrix} \frac{2}{\Delta x^2} + V_i & \frac{-1}{\Delta x^2} & 0 & 0 & 0 & & 0 & 0 & 0 \\ \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} + V_i & \frac{-1}{\Delta x^2} & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} + V_i & \frac{-1}{\Delta x^2} & 0 & & 0 & 0 & 0 \\ & & & & \vdots & & & & \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & \frac{2}{\Delta x^2} + V_i & \frac{-1}{\Delta x^2} \end{bmatrix} \tag{11}$$

Daraus ergibt sich ein lineares Gleichungssystem der Form

$$A_j x_{j-1} + B_j x_j + C_j x_{j+1} = F_j \qquad j = 1, ...N-2 \tag{12}$$

$$B_0 x_0 + C_0 x_1 = F_0 \tag{13}$$

$$A_{N-1} x_{N-2} + B_{N-1} x_{N-1} = F_{N-1} \tag{14}$$

x_0 kann umgeschrieben werden zu:

$$x_0 = \frac{F_0}{B_0} + \frac{-C_0}{B_0} x_1 =: \alpha_0 + \beta_0 x_1 \tag{15}$$

Und iterativ eingesetzt ergibt sich eine Gleichung für x_i

$$x_i = \frac{F_i - A_i\alpha_{i-1}}{B_i + A_i\beta_{i-1}} + \frac{-C_i}{B_i + A_i\beta_{i-1}} x_{i+1} =: \alpha_i + \beta_i x_{i+1} \tag{16}$$

Wegen $C_{N-1} = 0$ kann das System iterativ gelöst werden. Damit lässt sich die Schrödinger-Gleichung (1) numerisch lösen. Der Vorteil dieses Verfahrens ist der Erhalt der Wahrscheinlichkeit über die gesamte Zeit, da die Operatoren unitär sind.

3 Auswertung

3.1 Stationäre Schrödingergleichung

Die erste Aufgabe bestand darin, die Eigenwerte und Eigenfunktionen zu bestimmen. Dazu wurde das in 2.1 erläuterte Verfahren verwendet. Die Schrittgröße beträgt $\Delta x = 10^{-3}$ und zur Integration wurde das Runge-Kutta-Verfahren der Ordnung 4 verwendet. Für die Anfangsbedingungen wurde der Punkt $x_0 = 10$ gewählt. Die Ergebnisse sind in Abbildung 1 dargestellt. Wie zu erwarten sind abwechselnd symmetrische und antisymmetrische Lösungen zu sehen und jede Eigenfunktion hat einen Knotenpunkt mehr als die vorherige.

3.2 Zeitabhängige Schrödingergleichung

In der zweiten Aufgabe sollte die zeitabhängige Schrödingergleichung simuliert werden. Zuerst soll der Erhalt der Wahrscheinlichkeit für verschiedene Schrittgrößen Δx untersucht werden. Die Ergebnisse sind in Abbildung 2 für eine Gauss-Anfangsbedingung geplottet. Grundsätzlich sieht man, dass sogar für das große $\Delta x = 1$ die Wahrscheinlichkeit nahezu erhalten bleibt. Erst nach sehr langen Zeiten wird sich das dissipative Verhalten bemerkbar machen.
Interessant ist es, dass für kleine Δx die Wahrscheinlichkeit nicht fällt, sondern steigt. Vorallem der Anstieg bei $\Delta x = 5 \times 10^{-3}$ ist sehr stark. Für $\Delta x = 10^{-3}$ scheint die Wahrscheinlichkeit am stabilsten zu sein, auch wenn es verrauscht aussieht, das könnte von Rundungsfehlern stammen, die während der Simulation auftreten. Es macht also keinen Sinn kleinere Schrittgrößen zu wählen.
In der nächsten Aufgabe soll die Stationarität der Eigenfunktionen untersucht werden. Um die Stabilität am besten zu untersuchen werden Animationen der ersten 10 Sekunden erstellt. Sie sind unter dem Link `https://gitlab.com/Boundter/simulate_schroedinger/tree/master/plots` zu finden. Man kann deutlich erkennen, dass sie stabil sind und sich während der Zeit nicht bewegen. Nur an den Minima und Maxima kann eine leichte Schwankung gesehen werden. Das könnte an dem verwendeten Verfahren liegen, da es meist zu höheren harmonischen Schwingungen in der Lösung führt. Unter dem Link kann auch eine Animation der Entwicklung der Gauss-Funktion gefunden werden. Hier erkennt man deutlich eine zeitliche Entwicklung bzw. Oszillation zwischen den beiden Mulden.
In der letzten Aufgabe soll die Oszillation der Wahrscheinlichkeit von $|\Psi|^2(x < 0) := \int_{x<0} |\Psi|^2(x)dx$ untersucht werden. Die Anfangsbedingungen sind Superpositionen von Eigenfunktionen. Für die Simulation wurden immer Anfnagsbedingungen gewählt, sodass $E > 0$. Die Ergebnisse für einzelne Superpositionen sind in Abbildung 3 dargestellt. Es ist deutlich erkennbar, dass die Periode kleiner wird, wenn der Energieunterschied größer wird. In der Abbildung 4 ist die Periode der Schwingung in Abhängigkeit der Energie aufgetragen. Zu erwarten ist eine Schwingung mit der

Periode $T = 2\pi/\Delta E$[1] für $\hbar = 1$. Das entspricht genau der bestimmten Periode.

[1] Vladislav S. Olkhovsky and Sergei P. Maydanyuk; About evolution of particle transitions from one well to another in double-well potential; `http://arxiv.org/pdf/quant-ph/0311128.pdf`

4 Abbildungen

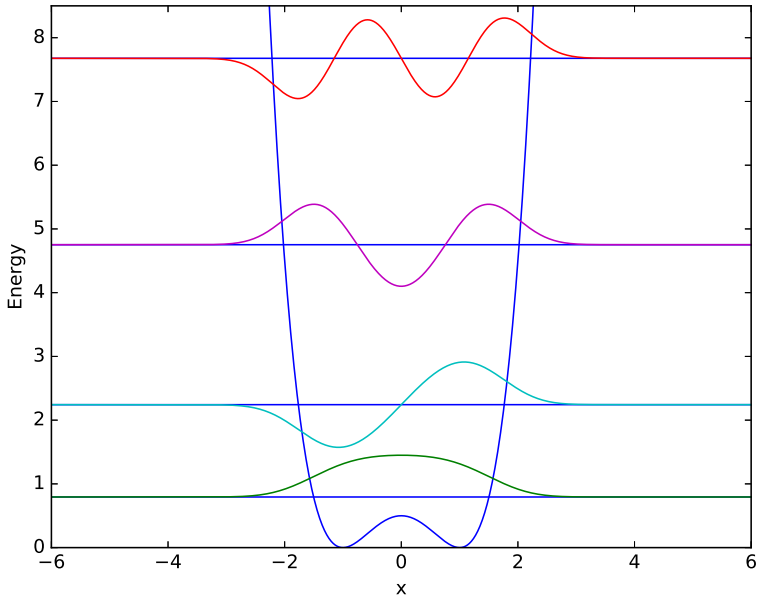


Abbildung 1: Eigenfunktionen und Eigenwerte der Schrödinger-Gleichung im Doppelmuldenpotential $V(x) = 1/2(1 - x^2)^2$.

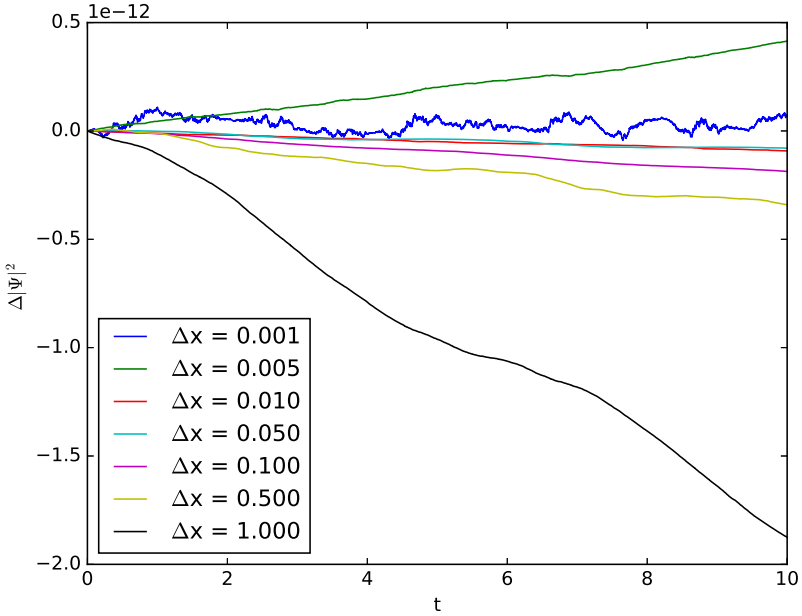
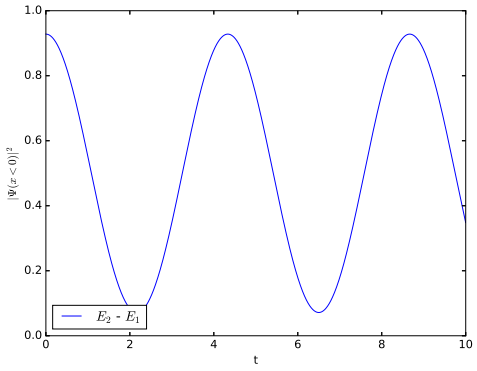
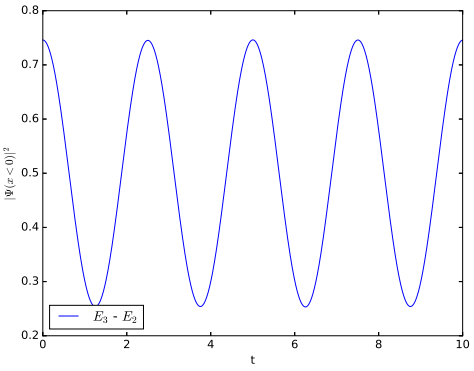


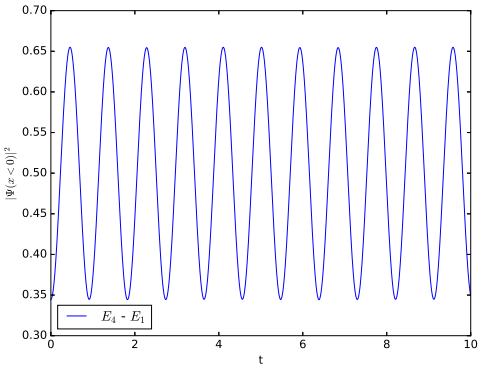
Abbildung 2: Änderung der Wahrscheinlichkeit $|\Psi|^2 = \int_{\mathbb{R}} |\Psi|^2(x)dx$ in Abhängigkeit von der Zeit und der Schrittgröße Δx in der Schrödingergleichung im Doppelmuldenpotential.



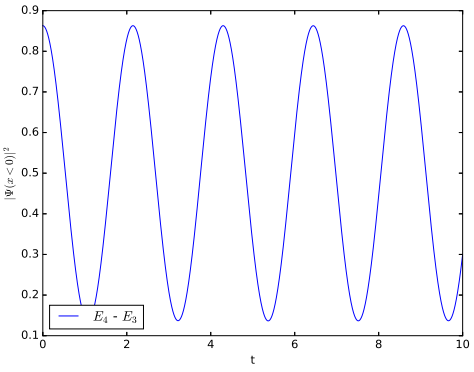
(a) $E_2 - E_1$



(b) $E_3 - E_2$



(c) $E_4 - E_1$



(d) $E_4 - E_3$

Abbildung 3: $|\Psi|^2(x < 0) = \int_{-\infty}^{0} |\Psi|^2(x)dx$ für Superpositionen von Eigenfunktionen der Schrödingergleichung im Doppelmuldenpotential.

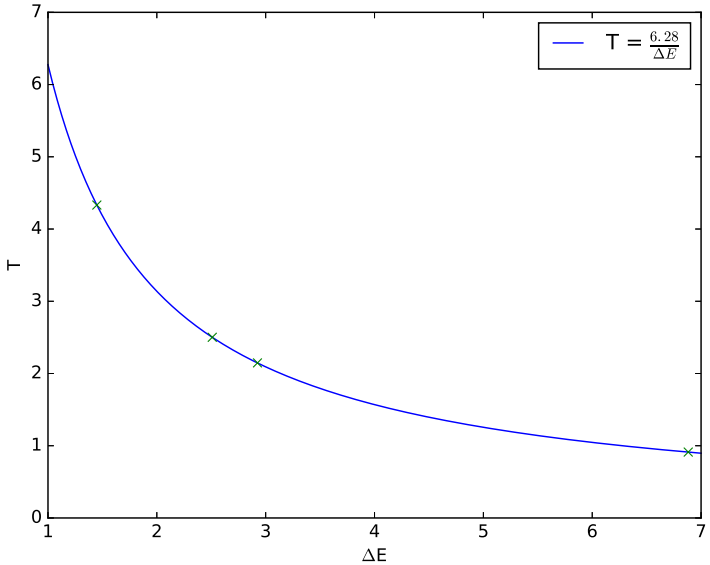


Abbildung 4: Periodizität der Wahrscheinlichkeit $|\Psi|^2(x < 0) = \int_{-\infty}^{0} |\Psi|^2(x)dx$ in Abhängigkeit der Energie der Schrödingergleichung im Doppelmuldenpotential.

5 Code

5.1 Wave.py - Klasse für Schrödinger-Wellen

```
import numpy as np
from scipy.integrate import ode
from math import copysign
import matplotlib.pyplot as plt

__all__ = ['SchroedingerWave', 'StationarySchroedinger', 'GeneralSchroedinger']

class SchroedingerWave:
  """
  Class of a typical solution for the Schroedinger-equation.

  Attributes:
  x: Spatial array
  Psi: Array of Psi(x)

  Methods:
  PlotWave(xRange, yRange, FileName, Label = ''):
   Plots the Wavefunction in the specified range and saves it to plots/FileName
  PlotAbsolute2(xRange, yRange, FileName, Label = ''):
   Plots |Psi(x)|^2 in the specified range and saves it to plots/FileName
  IntegrateAbsolute2(Begin, End):
   Integrates |Psi(x)|^2 in the range[Begin, End] and returns the value
  Normalize():
    Normalizes the wave by dividing by sqrt(IntegrateAbsolute2)
  """
  def __init__(self, x, Psi):
    self.x = x # np.array
    self.Psi = Psi # Psi(x)

  def __call__(self):
    return self.Psi

  def PlotWave(self, xRange, yRange, FileName, Label = ''):
    plt.figure()
    plt.xlabel('x'); plt.ylabel(r'$\Psi(x)$')
    plt.plot(self.x, self.Psi.real, label = 'Real Part%s' %Label)
    plt.plot(self.x, self.Psi.imag, label = 'Imag Part%s' %Label)
    plt.legend()
    plt.xlim(xRange); plt.ylim(yRange)
    plt.savefig('plots/%s' %FileName, format = 'eps', dpi = 1000)
    plt.close()

  def PlotAbsolute2(self, xRange, yRange, FileName, Label = ''):
    plt.figure()
    plt.xlabel('x'); plt.ylabel(r'$\vert\Psi(x)\vert^2$')
    plt.plot(self.x, np.absolute(self.Psi)**2, label = Label)
    if Label: plt.legend()
    plt.xlim(xRange); plt.ylim(yRange)
    plt.savefig('plots/%s' %FileName, format = 'eps', dpi = 1000)
    plt.close()

  def _FindNearestIndex(self, x):
    return (np.abs(self.x-x)).argmin()

  def IntegrateAbsolute2(self, Begin, End):
```

```
    BeginIndex = self._FindNearestIndex(Begin)
    EndIndex = self._FindNearestIndex(End) - 1
    dx = self.x[BeginIndex + 1: EndIndex + 1] - self.x[BeginIndex: EndIndex]
    dPsi2 = (np.absolute(self.Psi[BeginIndex + 1: EndIndex + 1])**2 + \
             np.absolute(self.Psi[BeginIndex: EndIndex])**2)/2
    Integral = np.sum(dx*dPsi2)
    return Integral

  def Normalize(self):
    self.Psi /= np.sqrt(self.IntegrateAbsolute2(self.x[0], self.x[-1]))

class StationarySchroedinger(SchroedingerWave):
  """
  Class for a Stationary Schroedinger-equation.

  Attributes:
  x: Spatial array
  V: Function for the Potential V(x)
  E: Energy of the Wave
  Psi: Psi(x) needs to be initialized by calling Integrate() with save
    parameter

  Methods:
  PlotWave(xRange, yRange, FileName, Label = ''):
   Plots the Wavefunction in the specified range and saves it to plots/FileName
  PlotAbsolute2(xRange, yRange, FileName, Label = ''):
   Plots |Psi(x)|^2 in the specified range and saves it to plots/FileName
  IntegrateAbsolute2(Begin, End):
   Integrates |Psi(x)|^2 in the range[Begin, End] and returns the value
  Normalize():
    Normalizes the wave by dividing by sqrt(IntegrateAbsolute2)
  Integrate(Start, End = 0., Scaling = 1e-20, save = False):
    Integrates the stationary Schroedinger-equation from Start to End using
    Runge-Kutta. Scaling helps to prevent overflow or underflow. The save
    parameter saves the solution in the Psi Attribute
  w(Boundary = 10.):
    Calculate w = Psi_-*Phi_+ - Phi_-*Psi+, where a root of w corrresponds to
    an eigenvalue E/ eigenfunction Psi
  FlipAntisymmetric():
    Flips an solution, if it is antisymmetric. This method should be called
    after an eigenfunction has been found, otherwise all solutions will be
    symmetric
  """
  def __init__(self, x, V, E):
    self.x = x
    self.Psi = np.zeros(len(x))
    self.V = V
    self.E = E

  def __call__(self):
    SchroedingerWave.__call__(self)

  def _InitialCondition(self, x, Scaling):
    Psi = Scaling*np.exp(-np.sqrt(self.E)*np.absolute(x))
    if x < 0:
      Phi = np.sqrt(self.E)*Psi
    else:
      Phi = -np.sqrt(self.E)*Psi
    return [Psi, Phi]
```

```
  def _SecondDerivative(self, x, y):
    Psi, Phi = y
    f = self.V(x) - self.E
    return [Phi, f*Psi]

  def Integrate(self, Start, End = 0., Scaling = 1e-20, save = False):
    InitialIndex = SchroedingerWave._FindNearestIndex(self, Start)
    FinalIndex = SchroedingerWave._FindNearestIndex(self, End)
    Iterator = -int(copysign(1., Start))
    Initial = self._InitialCondition(Start, Scaling)
    if save: self.Psi[InitialIndex] = Initial[0]
    Integrator = ode(self._SecondDerivative).set_integrator('dopri5')
    Integrator.set_initial_value(Initial, self.x[InitialIndex])
    for i in np.linspace(InitialIndex, FinalIndex, abs(FinalIndex - InitialIndex) + 1, dtype=in
      dx = self.x[i + Iterator] - self.x[i]
      Integrator.integrate(Integrator.t + dx)
      i += Iterator
      if save: self.Psi[i] = Integrator.y[0]
    return Integrator.y

  def w(self, Boundary = 10.):
    PsiPlus = self.Integrate(Boundary)
    PsiMinus = self.Integrate(-Boundary)
    return (PsiPlus[1]*PsiMinus[0] - PsiMinus[1]*PsiPlus[0])

  def FlipAntisymmetric(self):
    ZeroIndex = SchroedingerWave._FindNearestIndex(self, 0.)
    if abs(self.Psi[ZeroIndex]) <= 2e-3 and \
       int(copysign(1, self.Psi[ZeroIndex-1])) == int(copysign(1, self.Psi[ZeroIndex+1])):
      self.Psi[0:ZeroIndex] *= -1

class GeneralSchroedinger(SchroedingerWave):
  """
  Class of a typical solution for the Schroedinger-equation.

  Attributes:
  x: Spatial array
  Psi: Array of Psi(x)
  V: Spatial array of the Potential. It will be initialized by a function V(x)

  Methods:
  PlotWave(xRange, yRange, FileName, Label = ''):
   Plots the Wavefunction in the specified range and saves it to plots/FileName
  PlotAbsolute2(xRange, yRange, FileName, Label = ''):
   Plots |Psi(x)|^2 in the specified range and saves it to plots/FileName
  IntegrateAbsolute2(Begin, End):
   Integrates |Psi(x)|^2 in the range[Begin, End] and returns the value
  Normalize():
    Normalizes the wave by dividing by sqrt(IntegrateAbsolute2)
  IntegrateTime(dt):
    Integrates Psi one timestep dt by using the Crank-Nicolson-scheme
  """
  def __init__(self, x, Psi, V, t0 = 0.):
    self._x = x
    self.Psi = Psi.astype(np.complex_)
    self.V = V(x)
    self.t = t0
    self._Calcdx()
```

```
def __call__(self):
  SchroedingerWave.__call__(self)

def _Getx(self):
  return self._x

def _Setx(self, x):
  self._x = x
  self._Calcdx()

def _Calcdx(self):
  self.dx = np.mean(self._x[1:] - self._x[0:-1])

def _RightSide(self, dt):
  A = 0.5j*dt/self.dx**2
  B = 1 - 1j*dt/self.dx**2 - 0.5j*dt*self.V
  F = np.zeros(len(self.Psi), dtype = np.complex_)
  F[1:-1] = A*(self.Psi[0:-2] + self.Psi[2:]) + B[1:-1]*self.Psi[1:-1]
  F[0] = B[0]*self.Psi[0] + A*self.Psi[1]
  F[-1] = B[-1]*self.Psi[-1] + A*self.Psi[-2]
  return F

def _TestFunction(self, dt, F):
  A = -0.5j*dt/self.dx**2
  B = 1 + 1j*dt/self.dx**2 + 0.5j*dt*self.V
  FRef = np.zeros(len(self.Psi), dtype = np.complex_)
  FRef[1:-1] = A*(self.Psi[0:-2] + self.Psi[2:]) + B[1:-1]*self.Psi[1:-1]
  FRef[0] = B[0]*self.Psi[0] + A*Psi[1]
  FRef[-1] = B[-1]*self.Psi[-1] + A*Psi[-2]
  print(np.amax((FRef - F).real))


def IntegrateTime(self, dt):
  A = -0.5j*dt/self.dx**2
  B = 1 + 1j*dt/self.dx**2 + 0.5j*dt*self.V
  F = self._RightSide(dt)
  alpha = np.zeros(len(self.Psi), dtype = np.complex_)
  beta = np.zeros(len(self.Psi), dtype = np.complex_)
  alpha[0] = F[0]/B[0]; beta[0] = -A/B[0]
  for i in range(1, len(self.Psi)):
    alpha[i] = (F[i] - A*alpha[i-1])/(B[i] + A*beta[i-1])
    beta[i] = -A/(B[i] + A*beta[i-1])
  self.Psi[-1] = alpha[-1]
  for i in np.linspace(len(self.Psi) - 2, 0, len(self.Psi) + 1,\
              dtype = int):
    self.Psi[i] = alpha[i] + beta[i]*self.Psi[i+1]
  self.t += dt

x = property(_Getx, _Setx)
```

5.2 main2.py - Programm für die Simulation und zum plotten

```
#!/usr/bin/env python

import numpy as np
import wave as wv
import matplotlib.animation as animation
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

#########
# Input #
#########

#Task 1
dx = 1e-3 # Spatial step
Boundary = 10 # Solution in the interval [-Boundary, Boundary]
EMax = 15. # Maximum of E
# These dx will be used for chekcing of Probability conservation of Crank-Nico.
dxList = [1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1, 1]

#Task 2
dt = 1e-3 # Time step for the integration
tEnd = 10 # Stoptime


# Global variables
NumberRoots = 4 # How many roots to search
xPoints = 2*Boundary/dx # #points in the interval [-Boundary, Boundary]
if xPoints%2 == 0: # Make sure we have an uneven number of points, to get 0
  xPoints += 1
fps = 30 # Frames per second for the animation
StepsPerAnimation = int(1/(dt*fps)) # #timesteps per frame

#############
# Functions #
#############

def V(x):
  """
  Potential V(x) for the Schroedinger-equation
  """
  return 0.5*(1-x**2)**2

def Bisection(LowerE, UpperE, Wave):
  """
  Bisection of w(E) to find the eigenvalues
  """
  NewE = (LowerE + UpperE)/2
  Wave.E = NewE
  NewW = Wave.w()
  BisectionAccuracy = 1e-6
  while np.absolute(np.absolute(UpperE) - np.absolute(LowerE)) > BisectionAccuracy:
    if NewW < 0:
        LowerE = NewE
    else:
        UpperE = NewE
    NewE = (LowerE + UpperE)/2
    Wave.E = NewE
```

```
    NewW = Wave.w()
  return NewE

def FindEigenvalues(InitialE, EndE, dE):
  """
  Finds the eigenvalues E by using bisection in E. It will return NumberRoots
  of eigenvalues in a dictionary of the form {'eigenvalue0i':E}
  """
  Eigenvalues = {}
  x = np.linspace(-Boundary, Boundary, xPoints)
  Wave = wv.StationarySchroedinger(x, V, InitialE)
  LastValue = [Wave.E, Wave.w()]
  E = InitialE
  while E < EndE and len(Eigenvalues) < NumberRoots:
    Wave.E = E
    CurrentValue = [E, Wave.w()]
    print("Energy = %3g" %CurrentValue[0])
    if CurrentValue[1] < 0 and LastValue[1] > 0:
      Eigenvalues['eigenvalue%02i' %(len(Eigenvalues) + 1)] = \
      Bisection(CurrentValue[0], LastValue[0], Wave)
    elif CurrentValue[1] > 0 and LastValue[1] < 0:
      Eigenvalues['eigenvalue%02i' %(len(Eigenvalues) + 1)] = \
      Bisection(LastValue[0], CurrentValue[0], Wave)
    E += dE
    LastValue = CurrentValue
  if len(Eigenvalues) < NumberRoots:
    print('Not enough eigenvalues found. Enlarge Energy.')
    exit()
  return Eigenvalues

def Superposition(Key1, Key2, x):
  """
  Takes two eigenfunctions as arguments and integrates the over the time. The
  probability of begin <0 will be ploted.
  """
  print('%s and %s' % (Key1, Key2))
  plt.figure()
  plt.xlabel('t')
  plt.ylabel(r'$\vert \Psi  (x < 0) \vert ^2$')
  Psi = Eigenvalues[Key1][1].Psi - Eigenvalues[Key2][1].Psi
  Psi = wv.GeneralSchroedinger(x, Psi, V)
  Psi.Normalize()
  Time = np.arange(0, tEnd, dt)
  Probability = [Psi.IntegrateAbsolute2(-Boundary, 0)]
  for i in Time:
    Psi.IntegrateTime(dt)
    Probability.append(Psi.IntegrateAbsolute2(-Boundary, 0))
  Probability = Probability[:-1]
  plt.plot(Time, Probability, label = r'$E_%s$ - $E_%s$' %(Key1[-1], Key2[-1]))
  plt.xlim([0., tEnd])
  plt.legend(loc = 3)
  plt.savefig('plots/probability_left%s%s.eps' % (Key1[-1], Key2[-1]), \
              format = 'eps', dpi = 1000)
  plt.close()
  Period = Periodicity(Time, Probability)
  return (Period, Eigenvalues[Key1][0] - Eigenvalues[Key2][0])

def Periodicity(Time, Probability):
  """
```

```
  Finds the periodicity of the Probability by looking for the maximum in the
  range of 5 points
  """
  ProbOrder = sorted(Probability, reverse = True)
  PeriodTime = []
  for element in ProbOrder[:500]:
    Indx = Probability.index(element)
    if Indx < len(Probability) - 1 and \
      Indx > 1 and \
      Probability[Indx] > Probability[Indx - 1] and \
      Probability[Indx] > Probability[Indx + 1] and \
      Probability[Indx] > Probability[Indx - 2] and \
      Probability[Indx] > Probability[Indx + 2]:
      PeriodTime.append(Time[Indx])
  PeriodTime = np.asarray(PeriodTime); PeriodTime.sort()
  print(PeriodTime)
  PeriodTime = PeriodTime[1:] - PeriodTime[:-1]
  PeriodTime = np.mean(PeriodTime)
  return PeriodTime

def FitFunc(DeltaE, a):
  """
  Fit-function for the Periodicity of being left
  """
  return a/DeltaE

##############################################################################
# Animation
def init():
    VLine.set_data([], [])
    PsiLine.set_data([], [])
    time_text.set_text('')
    return (VLine, PsiLine, time_text)

def animate(j, Wave):
    for i in range(0, StepsPerAnimation):
      Wave.IntegrateTime(dt)
    Wave.IntegrateTime(dt)
    PsiLine.set_data(Wave.x, np.absolute(Wave.Psi)**2)
    VLine.set_data(x, Wave.V)
    time_text.set_text('time = %.2f' % (Wave.t))
    return(VLine, PsiLine, time_text)

def PlotAnimation(Wave, Key):
    anim = animation.FuncAnimation(fig, animate, init_func=init, blit=True, \
                                   frames = int(30*tEnd), fargs = (Wave, ))
    anim.save('plots/animation_%s.mp4' % Key, \
              extra_args=['-vcodec', 'libx264'], fps = fps)


##########
# Task 1 #
##########

# Find the eigenvalues and eigenfunctions and plot the eigenfunctions
Eigenvalues = FindEigenvalues(0.01, EMax, 0.1)
print(Eigenvalues)
x = np.linspace(-Boundary, Boundary, xPoints)
for Key in Eigenvalues:
```

```
    Wave = wv.StationarySchroedinger(x, V, Eigenvalues[Key])
    Wave.Integrate(Boundary, save = True); Wave.Integrate(-Boundary, save = True)
    Wave.Normalize()
    Wave.FlipAntisymmetric()
    Wave.PlotWave([-6., 6.], [-1., 1.], '%s.eps' %Key, \
                  Label = ': E = %.3g' %Eigenvalues[Key])
    Eigenvalues[Key] = [Eigenvalues[Key], Wave]

# Plot a scheme of the eigenfunctions and eigenvalues
plt.figure()
plt.xlabel('x')
plt.ylabel('Energy')
plt.plot(Eigenvalues['eigenvalue01'][1].x, \
         V(Eigenvalues['eigenvalue01'][1].x))
for Key in Eigenvalues:
    plt.axhline(y = Eigenvalues[Key][0], xmin =  -6.,xmax = 6.)
    plt.plot(Eigenvalues[Key][1].x, \
             Eigenvalues[Key][1].Psi + Eigenvalues[Key][0])
plt.xlim([-6., 6.])
plt.ylim([0., 8.5])
plt.savefig('plots/eigenfunctions.eps', format = 'eps', dpi = 1000)
plt.close()

##########
# Task 2 #
##########

# Animate the Eigenvalues to see, if they are stationary and plot them at t0
# and at tEnd.
for Key in Eigenvalues:
    Psi = wv.GeneralSchroedinger(Eigenvalues[Key][1].x, Eigenvalues[Key][1].Psi,\
                                 V)
    Psi.PlotAbsolute2([-6., 6.,], [-0.2, 1.], '%s_t0.eps' % Key)
    fig = plt.figure()
    ax = fig.add_subplot(111, autoscale_on=False, xlim=(-3, 3),\
                         ylim=(0, 1))
    PsiLine, = ax.plot([], [], lw=2)
    VLine,  = ax.plot([], [], lw=2)
    ax.set_xlabel('x')
    ax.set_ylabel(r'$\vert \Psi (x) \vert ^2$')
    time_text = ax.text(0.02, 0.95, '', transform=ax.transAxes)
    print('Eigenfunction %s' % Key)
    PlotAnimation(Psi, Key)
    Psi.PlotAbsolute2([-6., 6.,], [-0.2, 1.], '%s_t1.eps' % Key)

# Plot the Probability |Psi(x)|^2 over the whole spatial interval in
# dependence of the time and the stepsize dx for a Gaussian-wave
plt.figure()
plt.xlabel('t')
plt.ylabel(r'$\Delta \vert \Psi \vert ^2$')
for Deltax in dxList:
    x = np.arange(-10., 10., Deltax)
    Gauss = 1/np.sqrt(2*np.pi)*np.exp(-(x-1)**2/2)
    Psi = wv.GeneralSchroedinger(x, Gauss, V)
    Psi.Normalize()
    Time = np.arange(0, tEnd, dt)
    Probability = [1. - Psi.IntegrateAbsolute2(-Boundary, Boundary)]
    for i in Time:
        Psi.IntegrateTime(dt)
```

```
    Probability.append(1. - Psi.IntegrateAbsolute2(-Boundary, Boundary))
  Probability = Probability[:-1]
  plt.plot(Time, Probability, label = r'$\Delta$x = %.3f' % Deltax)
plt.xlim([0., tEnd])
plt.legend(loc = 3)
plt.savefig('plots/probability.eps', format = 'eps', dpi = 1000)
plt.close()

# Animate a Gaussian wave over the whole time range
x = np.linspace(-Boundary, Boundary, xPoints)
Gauss = 1/np.sqrt(2*np.pi)*np.exp(-(x-1)**2/2)
Psi = wv.GeneralSchroedinger(x, Gauss, V)
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-3, 3),\
                     ylim=(0, 1))
PsiLine, = ax.plot([], [], lw=2)
VLine,  = ax.plot([], [], lw=2)
ax.set_xlabel('x')
ax.set_ylabel(r'$\vert \Psi (x) \vert ^2$')
time_text = ax.text(0.02, 0.95, '', transform=ax.transAxes)
print('Gauss')
PlotAnimation(Psi, 'Gauss')

# Plot the Probability of being left of 0 for a superposition of two
# eigenvalues. The superposition is of the form E1 - E2
Period = []
Energies = []
y = Superposition('eigenvalue02', 'eigenvalue01', x)
Period.append(y[0]); Energies.append(y[1])
y = Superposition('eigenvalue04', 'eigenvalue01', x)
Period.append(y[0]); Energies.append(y[1])
y = Superposition('eigenvalue03', 'eigenvalue02', x)
Period.append(y[0]); Energies.append(y[1])
y = Superposition('eigenvalue04', 'eigenvalue03', x)
Period.append(y[0]); Energies.append(y[1])
Period = np.asarray(Period)
Energies = np.asarray(Energies)
popt, pcov = curve_fit(FitFunc, Energies, Period)
plt.figure()
plt.xlabel(r'$\Delta$E')
plt.ylabel('T')
ELin = np.linspace(1., 7., 1000)
plt.plot(ELin, FitFunc(ELin, popt[0]), \
         label = r'T = $\frac{%.2f}{\Delta E}$' % popt[0])
plt.plot(Energies, Period, 'x')
plt.legend()
plt.savefig('plots/periodicity.eps', format = 'eps', dpi = 1000)
plt.close()
```