Moleküldynamik auf dem Computer

Simulationen zur Reaktion $^{4.1}H + Cl_2 \longrightarrow {}^{4.1}HCl + Cl$

Richard Möhn

31. Dezember 2011

Inhaltsverzeichnis

Aufgabe 1: Morsepotential und harmonische Näherung 2

Aufgabe 2: Harmonischer Oszillator und numerische Näherungsverfahren 3

Aufgabe 3: Morseoszillator 5

Aufgabe 4: LEPS-Potential 7

Aufgabe 5: Trajektorien 9

Aufgabe 6: Schwingungszustände des Reaktionsprodukts 12

Quelltexte 14

Aufgabe 1: Morsepotential und harmonische Näherung

Zu dieser Aufgabe gehören die Abbildungen 1 und 2 sowie Listing 1 (Seite 14).

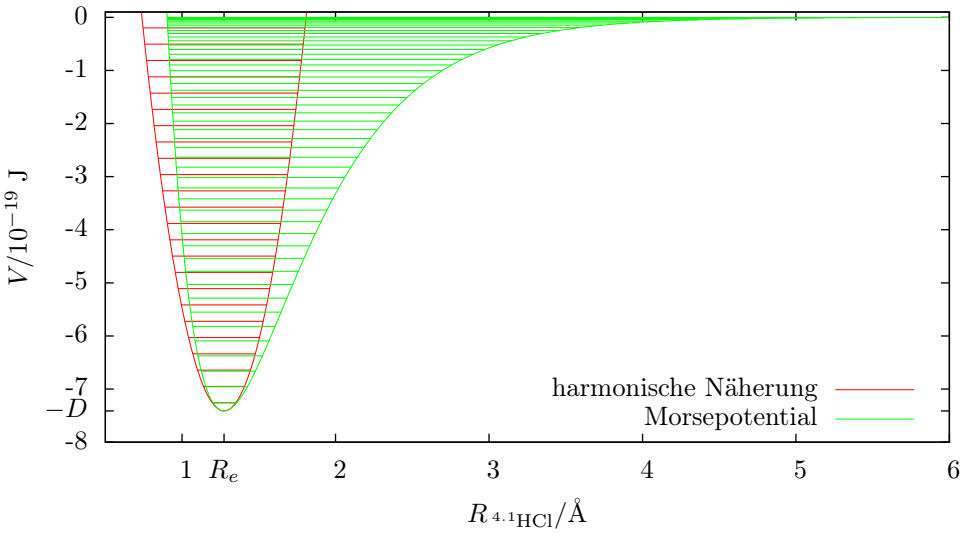

Abbildung 1: Morsepotential und harmonische Näherung für $^{4.1}$HCl in Abhängigkeit vom Atomabstand. Die waagerechten Linien sind die Schwingungszustände. Die Nummer des höchsten Schwingungszustands $v_{\max}$ ist 47.

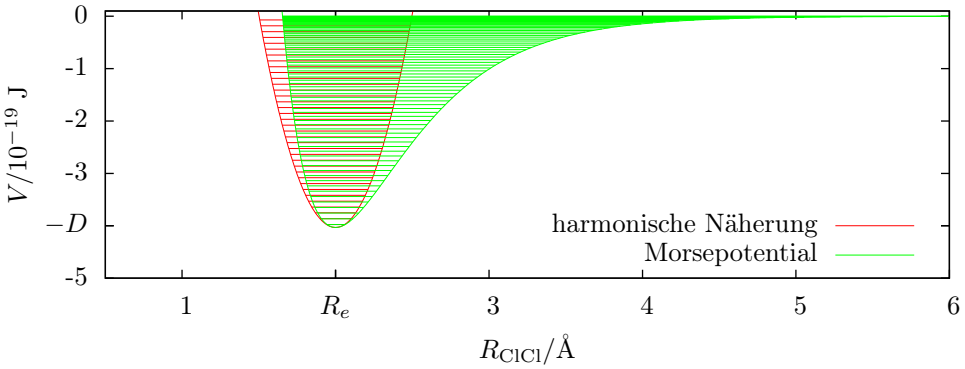

Abbildung 2: Morsepotential und harmonische Näherung für Cl_2 in Abhängigkeit vom Atomabstand. Die Nummer des höchsten Schwingungszustands $v_{\max}$ ist 71.

Aufgabe 2: Harmonischer Oszillator und numerische Näherungsverfahren

Zu dieser Aufgabe gehören die Abbildungen 3 und 4 sowie Listing 2 (Seite 17).

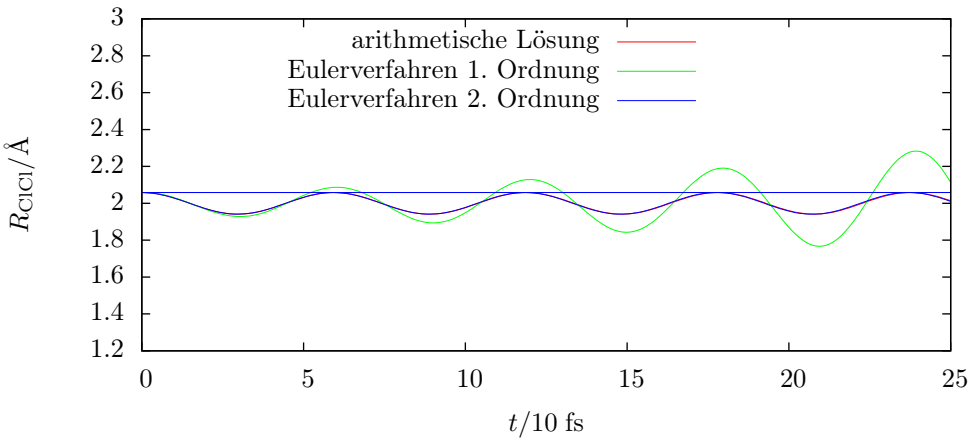

Abbildung 3: Cl_2-Molekül als harmonischer Oszillator bei Schwingungszustand $v = 0$. Die Schrittweite der Propagation ist $\frac{T_v}{50} \approx 0{,}119 \cdot 10$ fs (T_v ist die Schwingungsdauer des harmonischen Oszillators). (Das Eulerverfahren 2. Ordnung entspricht dem Runge-Kutta-Verfahren 2. Ordnung.) Die waagerechte blaue Linie erschien nach Neukompilierung im Jahre 2014. Sie gehört nicht dazu und ich weiß nicht, wo sie herkommt. Das Gleiche gilt für farbige waagerechte Linien in den folgenden drei Bildern.

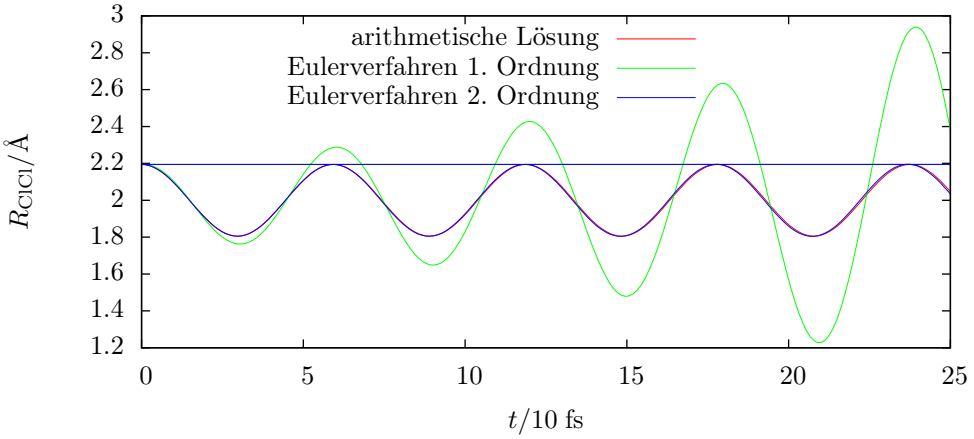

Abbildung 4: Cl_2-Molekül als harmonischer Oszillator bei Schwingungszustand $v = 5$. Die Schrittweite der Propagation ist wieder $\frac{T_v}{50}$.

Aufgabe 3: Morseoszillator

Zu dieser Aufgabe gehören die Abbildungen 5, 6, 7 und 8 sowie Listing 2 (Seite 17).

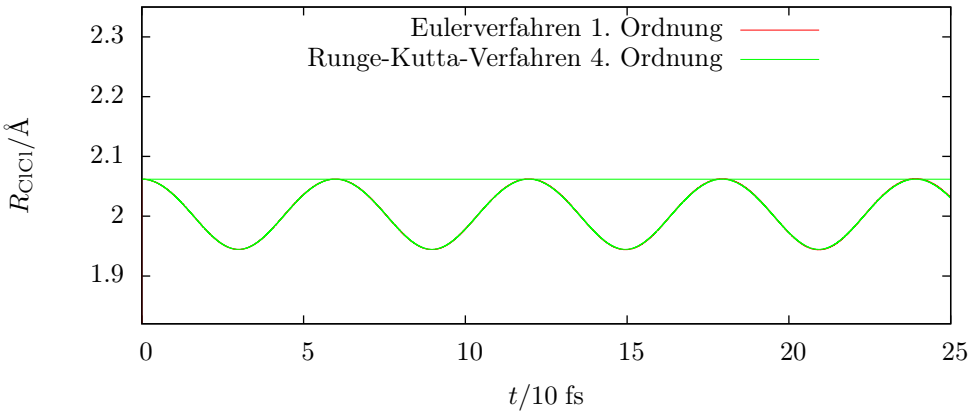

Abbildung 5: Cl_2-Molekül als Morseoszillator bei Schwingungszustand $v = 0$. Die Schrittweite der Propagation ist $\frac{T_v}{6000} \approx 9{,}895 \cdot 10^{-3}$fs. Wegen der kleinen Schrittweite ist auch die Näherung durch das Eulerverfahren erster Ordnung relativ genau. Die numerisch ermittelte Schwingungsdauer beträgt $5{,}978 \cdot 10$ fs.

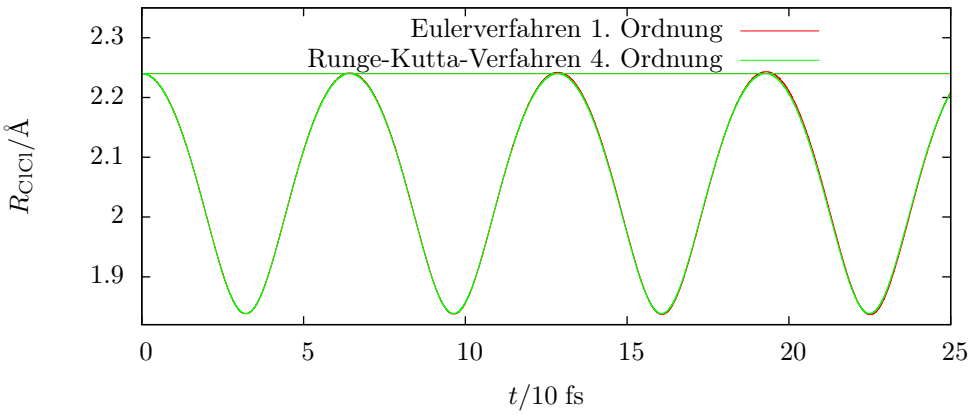

Abbildung 6: Cl_2-Molekül als Morseoszillator bei Schwingungszustand $v = 5$. Die Schrittweite der Propagation ist wieder $\frac{T_v}{6000}$. Die numerisch ermittelte Schwingungsdauer beträgt $6{,}427 \cdot 10$ fs.

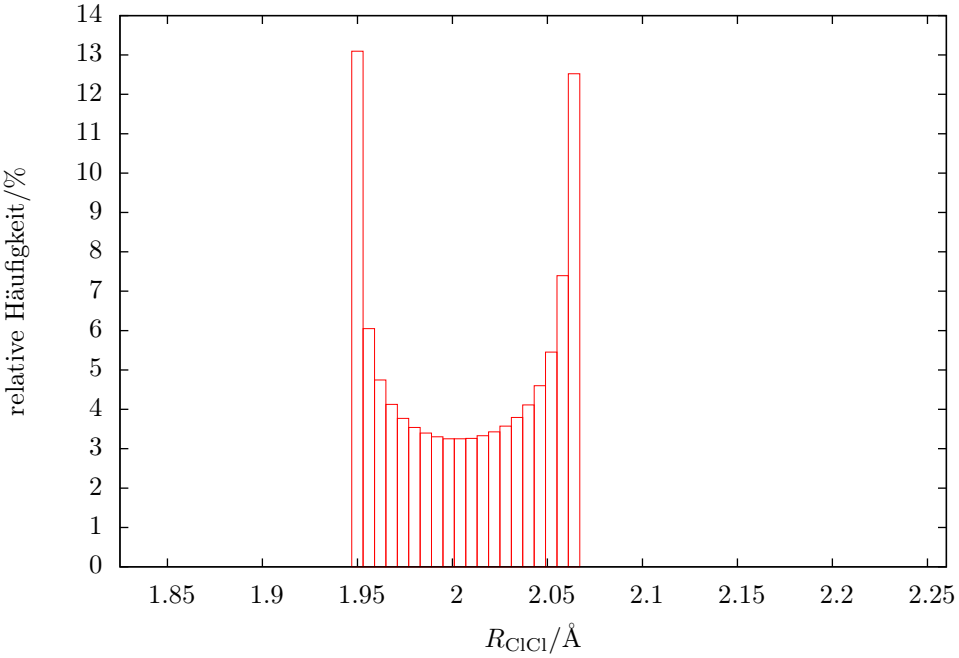

Abbildung 7: Histogramm für die relativen Häufigkeiten der Radien beim Morseoszillator Cl_2 mit $v = 0$. Die Intervallbreite beträgt 0,006 Å.

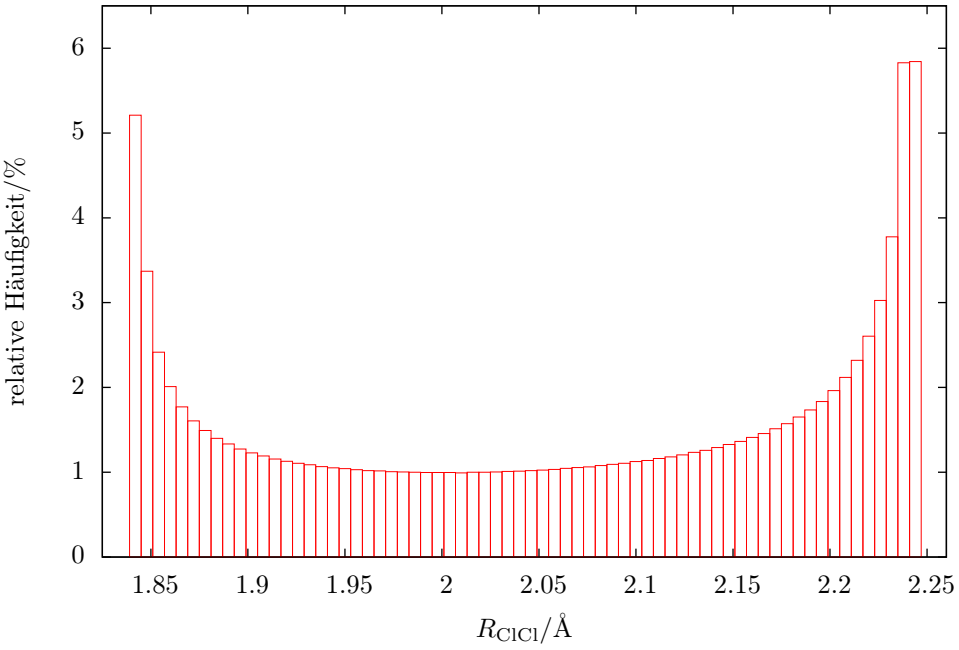

Abbildung 8: Histogramm für die relativen Häufigkeiten der Radien beim Morseoszillator Cl_2 mit $v = 5$. Die Intervallbreite beträgt wieder 0,006 Å.

Aufgabe 4: LEPS-Potential

Zu dieser Aufgabe gehören die Abbildungen 9 und 10 sowie Listing 4 (Seite 29).

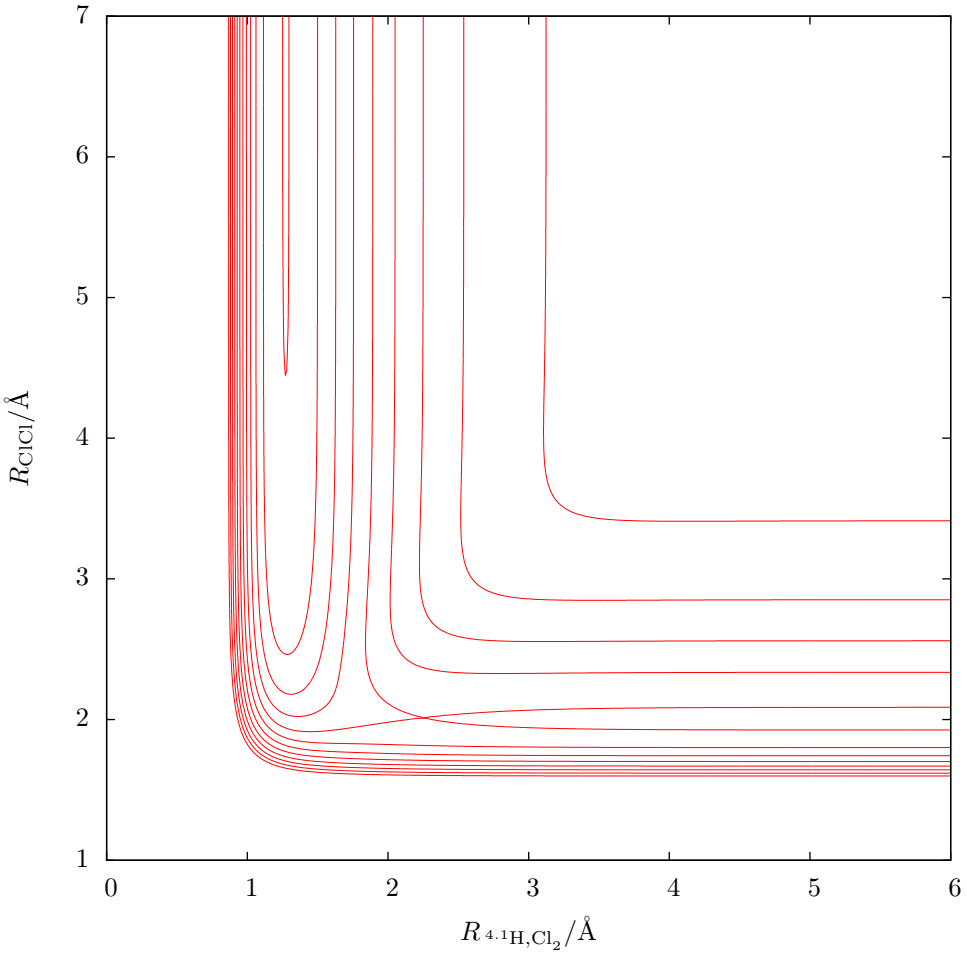

Abbildung 9: LEPS-Potential mit Bindungskoordinaten. Höhenlinien ab $-10.87 \cdot 10^{-19}$ J alle $0.87 \cdot 10^{-19}$ J.

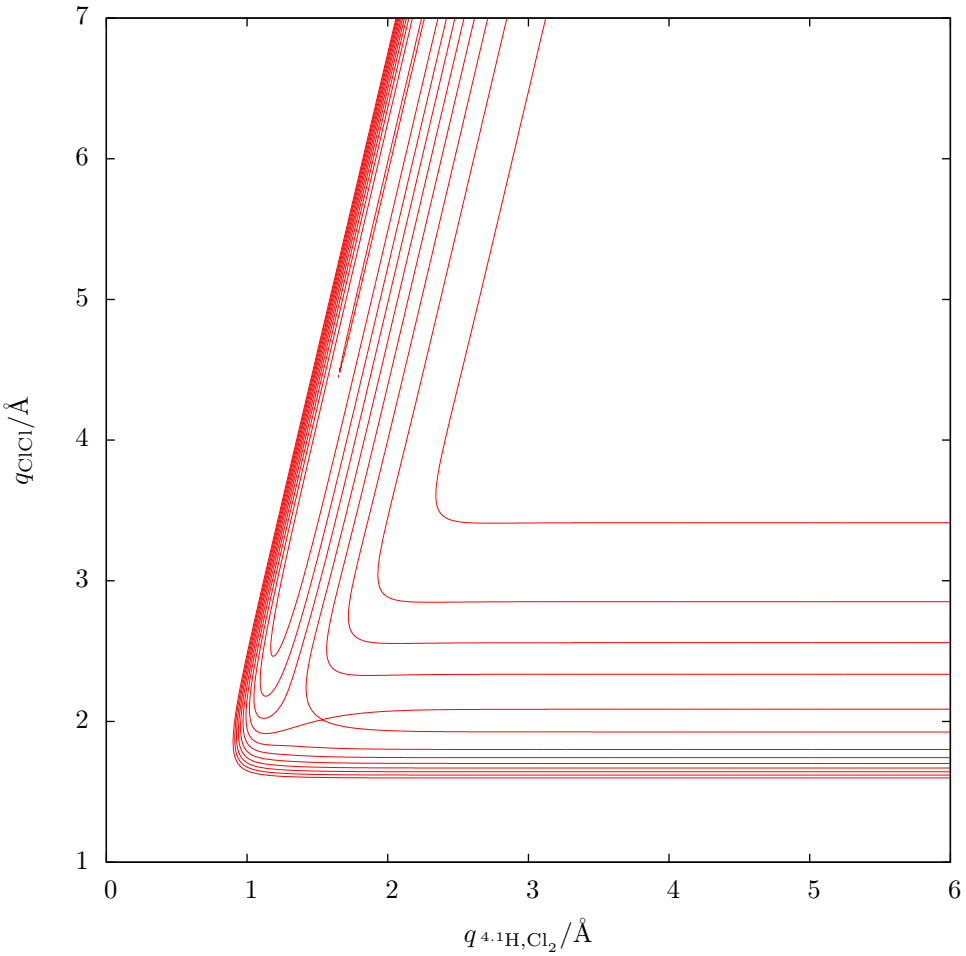

Abbildung 10: LEPS-Potential mit massengewichteten Koordinaten. Höhenlinien ab $-10.87 \cdot 10^{-19}$ J alle $0.87 \cdot 10^{-19}$ J. Der Sattelpunkt (Schnittpunkt der Höhenlinien) liegt bei $q_{^{4.1}\mathrm{H,Cl_2}} = 1{,}530$ Å, $q_{\mathrm{ClCl}} = 2{,}010$ Å bei einem Potential $V_{\mathrm{LEPS}} = -3{,}928 \cdot 10^{-19}$ J. Der Bereichswinkel ist 76,7°.

Aufgabe 5: Trajektorien

Zu dieser Aufgabe gehören die Abbildungen 11, 12 und 13 sowie Listing 4 (Seite 29).

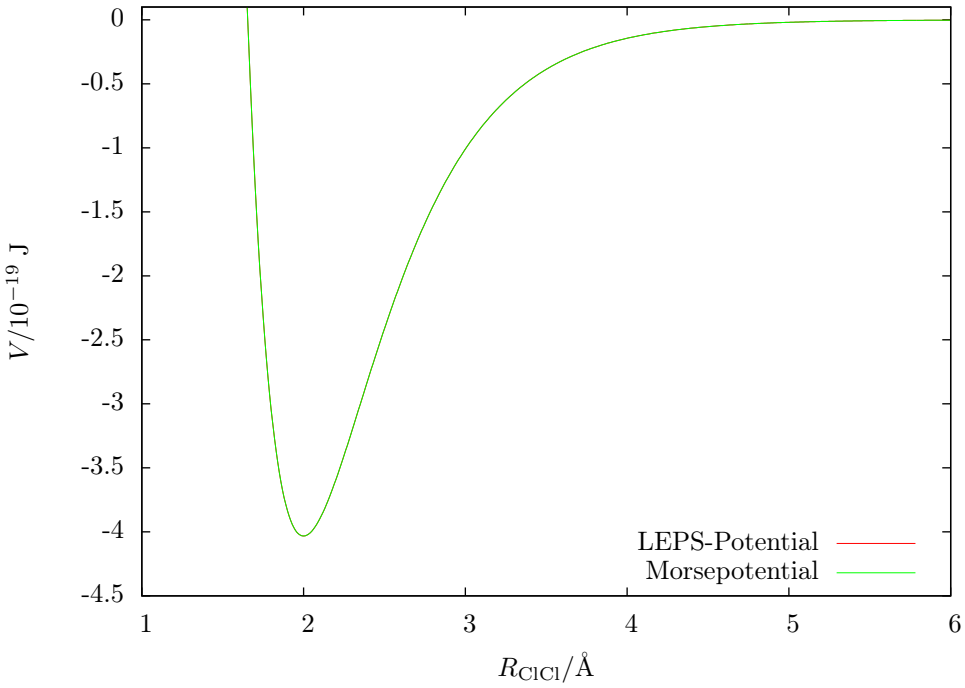

Abbildung 11: Schnitt durch das LEPS-Potential bei $q_{4.1\mathrm{H,Cl_2}} = 8$ Å parallel zur Ordinatenachse und Morsepotential von Cl_2 mit gleichem Kurvenverlauf

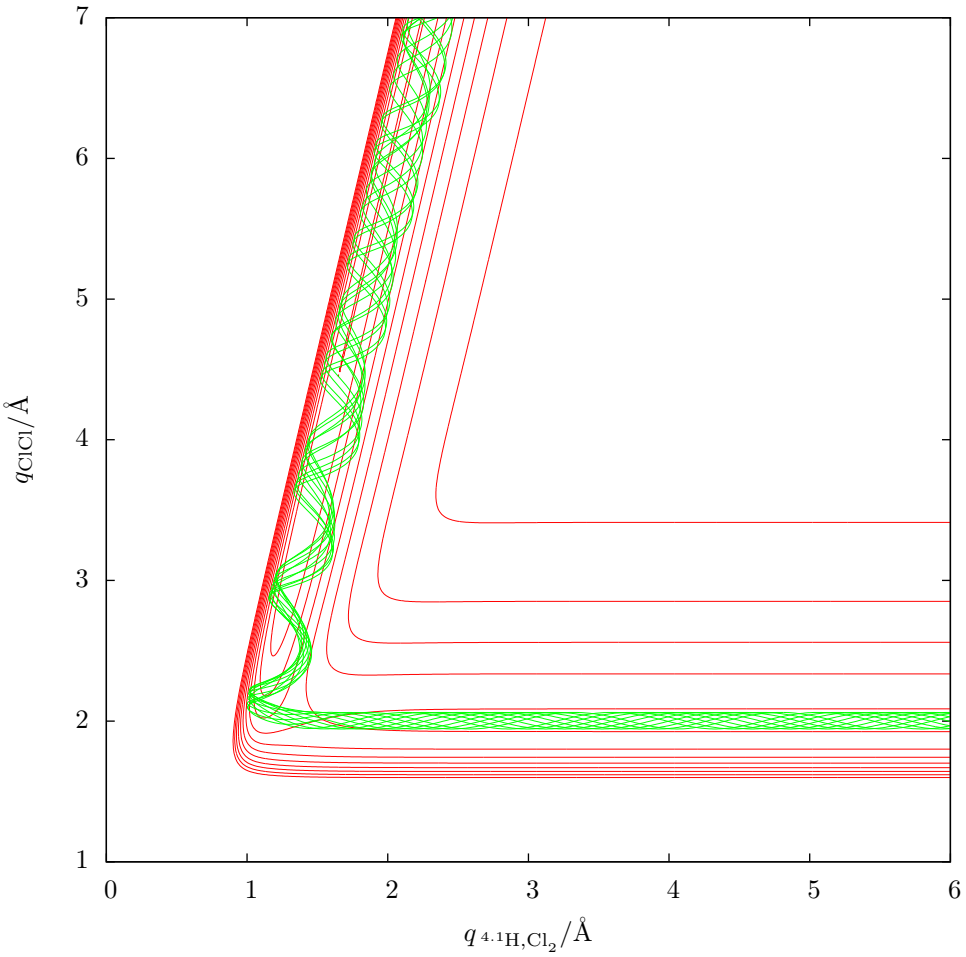

Abbildung 12: Zehn Trajektorien mit $v_{\mathrm{Cl}_2} = 0$ und Anfangsimpuls $p_x = -7{,}136 \cdot 10^{-23}$ kg m/s (Anfangsenergie $E_0 = -3{,}1 \cdot 10^{-19}$ J). Alle sind reaktiv. (LEPS-Potential siehe Abbildung 10.)

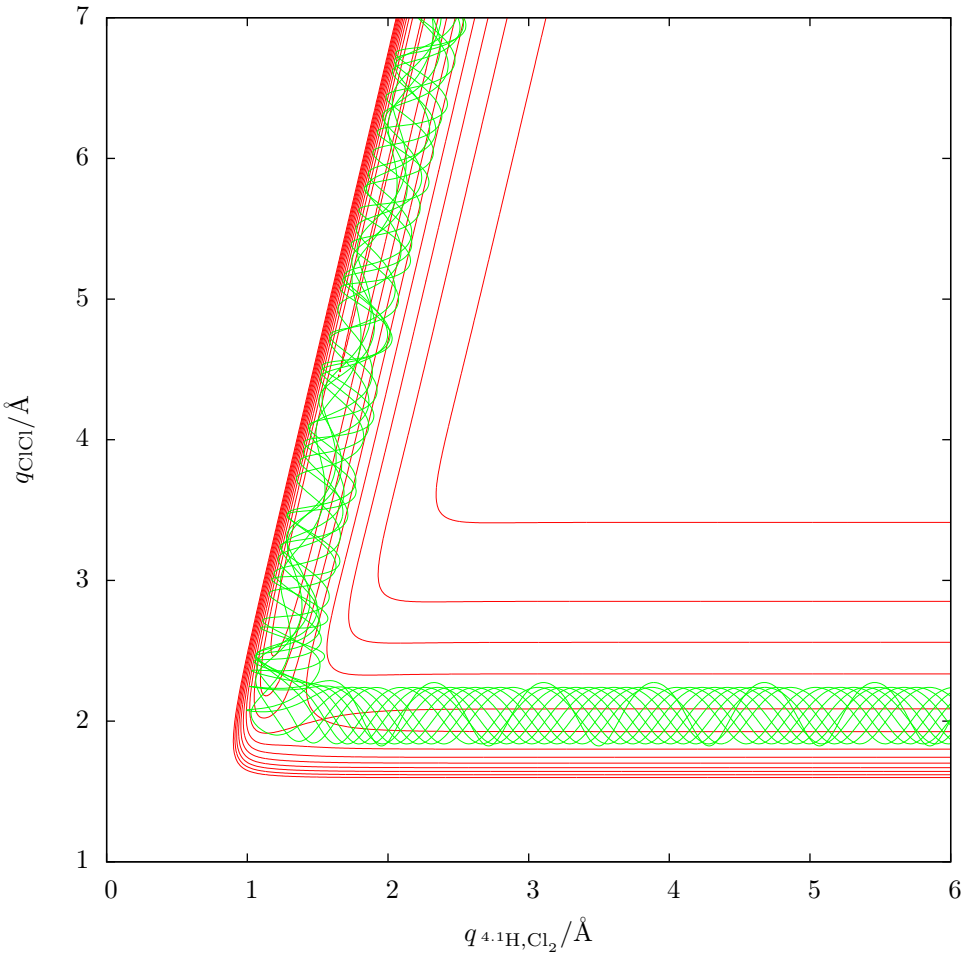

Abbildung 13: Zehn Trajektorien mit $v_{\mathrm{Cl}_2} = 5$ und Anfangsimpuls $p_x = -4{,}457 \cdot 10^{-23}$ kg m/s (Anfangsenergie $E_0 = -3{,}1 \cdot 10^{-19}$ J). Neun sind reaktiv. (LEPS-Potential siehe Abbildung 10.)

Aufgabe 6: Schwingungszustände des Reaktionsprodukts

Zu dieser Aufgabe gehören die Abbildungen 14, 15, 16 und 17 sowie die Listings 4 (Seite 29) und 7 (Seite 44).

Die Abbildungen zeigen die Energieverläufe ausgewählter Trajektorien der behandelten Reaktion in Abhängigkeit von der Zeit. Man kann erkennen, in welchem Schwingungszustand, je nach Ausgangsbedingungen, das Reaktionsprodukt $^{4.1}$HCl geboren wird.

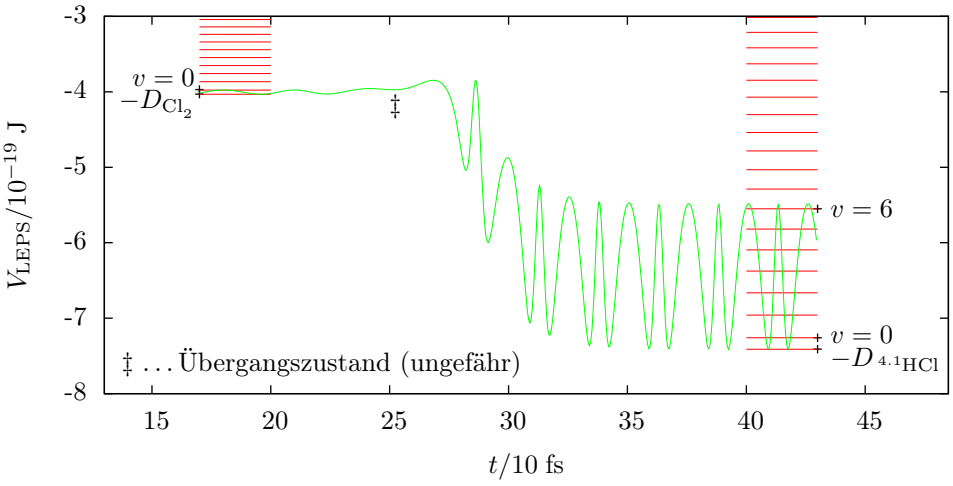

Abbildung 14: Energieverlauf der Trajektorie mit $v_{\mathrm{Cl_2}} = 0$ und $q_{^{4.1}\mathrm{H,Cl_2},0} = 8{,}000$ Å

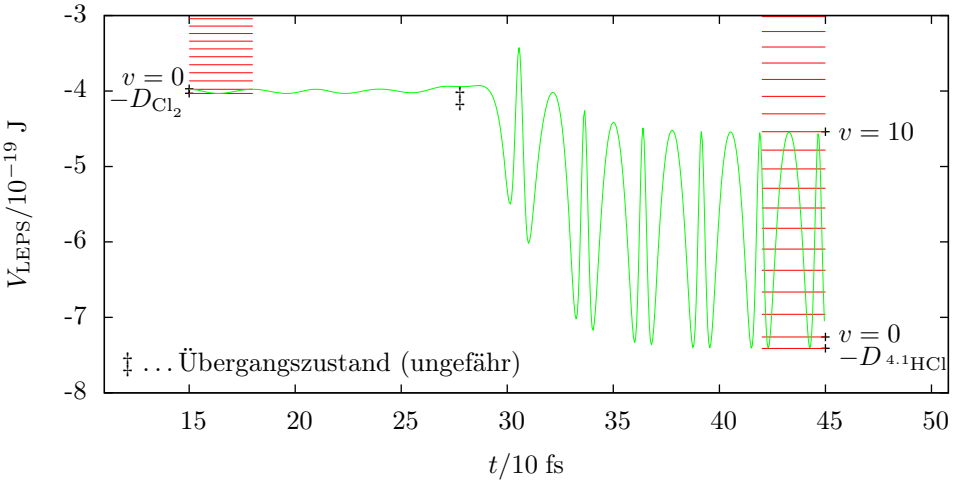

Abbildung 15: Energieverlauf der Trajektorie mit $v_{\mathrm{Cl_2}} = 0$ und $q_{^{4.1}\mathrm{H,Cl_2},0} = 8{,}534$ Å

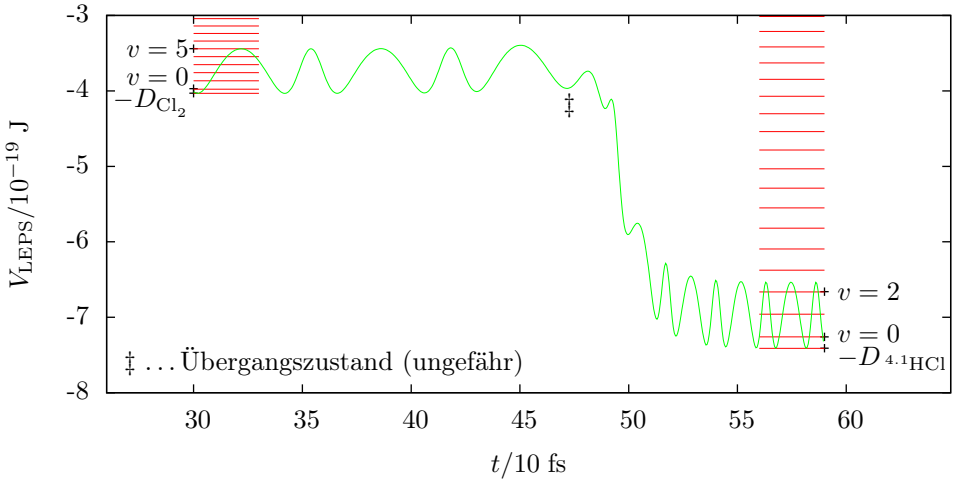

Abbildung 16: Energieverlauf der Trajektorie mit $v_{\mathrm{Cl_2}} = 5$ und $q_{4.1\mathrm{H,Cl_2},0} = 8{,}628$ Å

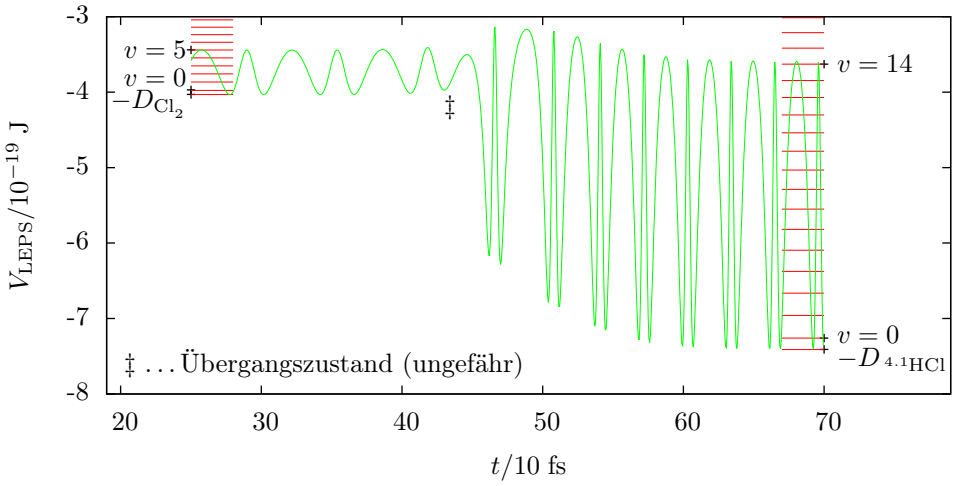

Abbildung 17: Energieverlauf der Trajektorie mit $v_{\mathrm{Cl_2}} = 5$ und $q_{4.1\mathrm{H,Cl_2},0} = 8{,}359$ Å

Quelltexte

Listing 1: morse-harm.f90

```
! Berechnet Potentialkurven und Schwingungszustände für die Moleküle von
! Ausgangsstoff und Reaktionsprodukt.

program morse_harm
use konst_umr
use teilchenparam
implicit none

! Achtung: Um das alte Programm an das neue System anzupassen und damit
! flexibler zu machen, wurden die alten Variablen mit den Werten aus
! teilchenparam belegt. Die Variablennamen enden daher immer noch auf HF und
! F2, obwohl durch Umbelegung von mol_1 und mol_2 andere Moleküle verwendet
! werden können. (Auch in den Dateinamen!)
type(Mol2), parameter :: &
    mol_1 = MCl, &
    mol_2 = Cl2  ! zu benutzende Moleküle
real(8), parameter :: &
! Werte für HF
    D_HF    = mol_1%D,    & ! Topftiefe
    beta_HF = mol_1%beta, & ! Morsekonstante
    R_e_HF  = mol_1%R_e,  & ! Gleichgewichtsradius
    my_HF   = mol_1%my,   & ! reduzierte Masse
! Werte für F2
    D_F2    = mol_2%D,    & ! Topftiefe
    beta_F2 = mol_2%beta, & ! Morsekonstante
    R_e_F2  = mol_2%R_e,  & ! Gleichgewichtsradius
    my_F2   = mol_2%my!   & ! reduzierte Masse

real(8) :: &
    k_HF,  &
    k_F2
    ! Kraftkonstanten der zu benutzenden Moleküle

real(8) :: &
! Schleifenvariablen
    R,              & ! aktueller Radius
    R_0   = 0,      & ! Anfangswert für Radius
    delta = 0.01,   & ! Schrittweite
    V_hF2,          & ! aktuelles harmonisches Potential für F2
    V_hHF,          & ! aktuelles harmonisches Potential für HF
    V_mF2,          & ! aktuelles Morsepotential für F2
    V_mHF,          & ! aktuelles Morsepotential für HF
    R_t,            & ! Differenz zwischen Umkehrpunkt- und
                      ! Gleichgewichtsradius
    R_tr,           & ! Radius des rechten Umkehrpunkts
    R_tl,           & ! Radius des linken Umkehrpunkts
! diverse
    omega   ! Kreisfrequenz der Schwingung

integer :: &
```

```
51      v_max, & ! höchster Schwingungszustand
52      v,     & ! aktueller Schwingungszustand
53      i        ! Iterator
54
55  ! Kraftkonstanten berechnen
56  k_HF = kraftkonst(mol_1)
57  k_F2 = kraftkonst(mol_2)
58
59  !! öffnen der Dateien !!!!!!!!!!!!!!!
60  open(unit=13, file='potkurv-harm-HF.dat', status='unknown', form='formatted')
61  open(unit=14, file='potkurv-morse-HF.dat', status='unknown', form='formatted')
62  open(unit=15, file='potkurv-harm-F2.dat', status='unknown', form='formatted')
63  open(unit=16, file='potkurv-morse-F2.dat', status='unknown', form='formatted')
64
65  !! Berechnung der Potentialkurven !!!!!!!!!!!!!!!
66  ! Potential/Radius-Werte berechnen
67  do i = 0,1000
68      ! nächsten Radius-Wert berechnen
69      R = R_0 + i * delta
70
71      ! Potential-Werte mit dem harmonischen Oszillator für HF und F2 berechnen
72      V_hHF = -D_HF + 0.5 * 2*D_HF*beta_HF**2 * (R - R_e_HF)**2
73      V_hF2 = -D_F2 + 0.5 * 2*D_F2*beta_F2**2 * (R - R_e_F2)**2
74
75      ! Morsepotential-Werte für HF und F2 berechnen
76      V_mHF = -D_HF + D_HF * (exp(-beta_HF * (R - R_e_HF)) - 1)**2
77      V_mF2 = -D_F2 + D_F2 * (exp(-beta_F2 * (R - R_e_F2)) - 1)**2
78
79      ! berechnete Werte in Datei schreiben
80      write(13,*) R, V_hHF
81      write(14,*) R, V_mHF
82      write(15,*) R, V_hF2
83      write(16,*) R, V_mF2
84  enddo
85
86  ! Leerzeile in die Dateien schreiben. -- Gnuplot braucht immer zwei.
87  write(13,*)
88  write(14,*)
89  write(15,*)
90  write(16,*)
91  write(13,*)
92  write(14,*)
93  write(15,*)
94  write(16,*)
95
96  !! Berechnung der Schwingungszustände !!!!!!!!!!!!!!!
97
98  ! Nummer des höchsten Schwingungszustands für HF berechnen
99  omega = sqrt(2*D_HF / my_HF) * beta_HF
100 v_max = floor(2*D_HF / (hquer*omega) - 0.5)
101 write(*,*) v_max
102
103 ! Berechnung der Schwingungszustände für HF für die Morsekurve
104 do v=0,v_max
```

```
105     ! Berechnung der Umkehrradien für die Morsekurve
106     R_t   = sqrt(1/D_HF * ((v + 0.5) * hquer * omega &
107                           - (v + 0.5)**2 * (hquer * omega)**2 / (4 * D_HF)))
108         ! Achtung! Dieses R_t ist nicht so, wie es in der Deklaration steht.
109     R_tr  = R_e_HF - 1/beta_HF * log(1 + R_t)
110     R_tl  = R_e_HF - 1/beta_HF * log(1 - R_t)
111     ! Berechnung der Energien für die Morsekurve
112     V_mHF = -D_HF + (v + 0.5) * hquer * omega &
113           - (v + 0.5)**2 * (hquer * omega)**2 / (4 * D_HF)
114
115     ! Ausgeben der Werte für die Morsekurve
116     write(14,*) R_tr, V_mHF
117     write(14,*) R_tl, V_mHF
118     write(14,*)
119     write(14,*)
120         ! Gnuplot braucht immer zwei Leerzeilen
121 enddo
122
123 ! Berechnung der Schwingungszustände für HF für die harmonische Kurve
124 v_max = v_max / 2
125 do v=0,v_max
126     ! Berechnung der Umkehrradien für die harmonische Kurve
127     R_t   = sqrt(2 * (v + 0.5) * hquer * omega / k_HF)
128     R_tr  = R_e_HF + R_t
129     R_tl  = R_e_HF - R_t
130
131     ! Berechnung der Energiewerte für die harmonische Kurve
132     V_hHF = -D_HF + 0.5 * k_HF * (R_tr - R_e_HF)**2
133
134     ! Ausgeben der Werte für die harmonische Kurve
135     write(13,*) R_tr, V_hHF
136     write(13,*) R_tl, V_hHF
137     write(13,*)
138     write(13,*)
139         ! Gnuplot braucht immer zwei Leerzeilen
140 enddo
141
142 ! Nummer des höchsten Schwingungszustands für F2 berechnen
143 omega = sqrt(2*D_F2 / my_F2) * beta_F2
144 v_max = floor(2*D_F2 / (hquer*omega) - 0.5)
145 write(*,*) v_max
146
147 ! Berechnung der Schwingungszustände für F2 für die Morsekurve
148 do v=0,v_max
149     ! Berechnung der Umkehrradien für die Morsekurve
150     R_t   = sqrt(1/D_F2 * ((v + 0.5) * hquer * omega &
151                           - (v + 0.5)**2 * (hquer * omega)**2 / (4 * D_F2)))
152         ! Achtung! Dieses R_t ist nicht so, wie es in der Deklaration steht.
153     R_tr  = R_e_F2 - 1/beta_F2 * log(1 + R_t)
154     R_tl  = R_e_F2 - 1/beta_F2 * log(1 - R_t)
155     ! Berechnung der Energien für die Morsekurve
156     V_mF2 = -D_F2 + (v + 0.5) * hquer * omega &
157           - (v + 0.5)**2 * (hquer * omega)**2 / (4 * D_F2)
158
```

```
159     ! Ausgeben der Werte für die Morsekurve
160     write(16,*) R_tr, V_mF2
161     write(16,*) R_tl, V_mF2
162     write(16,*)
163     write(16,*)
164         ! Gnuplot braucht immer zwei Leerzeilen
165 enddo
166
167 ! Berechnung der Schwingungszustände für F2 für die harmonische Kurve
168 v_max = v_max / 2
169 do v=0,v_max
170     ! Berechnung der Umkehrradien für die harmonische Kurve
171     R_t   = sqrt(2 * (v + 0.5) * hquer * omega / k_F2)
172     R_tr  = R_e_F2 + R_t
173     R_tl  = R_e_F2 - R_t
174
175     ! Berechnung der Energiewerte für die harmonische Kurve
176     V_hF2 = -D_F2 + 0.5 * k_F2 * (R_tr - R_e_F2)**2
177
178     ! Ausgeben der Werte für die harmonische Kurve
179     write(15,*) R_tr, V_hF2
180     write(15,*) R_tl, V_hF2
181     write(15,*)
182     write(15,*)
183         ! Gnuplot braucht immer zwei Leerzeilen
184 enddo
185
186 !! Schließen der Dateien !!!!!!!!!!!!!!!
187 close(13)
188 close(14)
189 close(15)
190 close(16)
191
192
193 end program morse_harm
```

Listing 2: svp-t.f90

```
1  ! Ausgabe von Ort-Zeit- und Geschwindigkeit-/Impuls-Zeit-Werten für den
2  ! harmonischen Oszillator
3
4  program svp_t
5  use teilchenparam
6  use schwingungen
7  use utils
8  use konst_umr
9  use rk4_mod
10 implicit none
11
12 !! DECLARATIONS !!!!!!!!!!!!!!!
13 real(8) :: &
14     T_v,      & ! Periodendauer des harmonischen Oszillators
15     omega!    & ! Kreisfrequenz des harmonischen Oszillators
16
```

```
integer :: &
    v,        & ! zu benutzendes Schwingungsniveau
    last_ind    ! jeweils letzter Index der vom Propagator gefüllten Arrays

type(Mol2) :: mol = Cl2 ! zu benutzendes Molekül

real(8) :: &
    t_0 = 0,        & ! Zeitanfangswert
    t_end,          & ! Zeitendwert
    delta_t,        & ! Schrittweite der Propagation
    delta_r = 0.006, & ! Intervallbreite im Histogramm
    mors_period!    & ! numerisch zu ermittelnde Periodendauer für den
                      ! Morseoszillator

! Array für zu berechnende Punkte
type(SPKT), dimension(:), allocatable :: punkte
    ! Falls das Array nach der Propagation verwendet wird, wird es unmittelbar
    ! danach verwendet. Ein Array reicht also für alle.

! Dateinamen
character(len=19) :: &
    h_arit_v0_file = 'svp-t-h-v0-arit.dat', &
    h_arit_v5_file = 'svp-t-h-v5-arit.dat', &
    h_eul1_v0_file = 'svp-t-h-v0-eul1.dat', &
    h_eul1_v5_file = 'svp-t-h-v5-eul1.dat', &
    h_eul2_v0_file = 'svp-t-h-v0-eul2.dat', &
    h_eul2_v5_file = 'svp-t-h-v5-eul2.dat', &
    m_eul1_v0_file = 'svp-t-m-v0-eul1.dat', &
    m_eul1_v5_file = 'svp-t-m-v5-eul1.dat', &
    m_ruk4_v0_file = 'svp-t-m-v0-ruk4.dat', &
    m_ruk4_v5_file = 'svp-t-m-v5-ruk4.dat'

! Kreisfrequenz und Periodendauer berechnen
omega = kreisfreq(mol)
T_v   = 2 * Pi / omega
write(*,*) 'Periodendauer für harmonischen Oszillator:', T_v


! Schleifenparameter berechnen
t_end   = 5 * T_v
delta_t = T_v / 50
write(*,*) 'harm. Schrittweite:', delta_t

! Achtung: Array punkte wird bei jedem Aufruf von propagator oder
! spkt_harm_array neu belegt.

! Berechnungen für Aufgabe 1: harmonischer Oszillator von Molekül 2 mit
! analytischer Berechnung, Propagation mit Euler erster Ordnung und
! Propagation mit Euler zweiter Ordnung (entspricht Runge-Kutta zweiter
! Ordnung).
v = 0
call spkt_harm_array(punkte, mol, v, t_0, t_end, delta_t, h_arit_v0_file)
call propagator(punkte, last_ind, next_spkt_harm_eul1, mol, &
    first_spkt_harm(mol, v), t_end, delta_t, h_eul1_v0_file)
```

```
71  call propagator(punkte, last_ind, next_spkt_harm_eul2, mol, &
72      first_spkt_harm(mol, v), t_end, delta_t, h_eul2_v0_file)
73
74  v = 5
75  call spkt_harm_array(punkte, mol, v, t_0, t_end, delta_t, h_arit_v5_file)
76  call propagator(punkte, last_ind, next_spkt_harm_eul1, mol, &
77      first_spkt_harm(mol, v), t_end, delta_t, h_eul1_v5_file)
78  call propagator(punkte, last_ind, next_spkt_harm_eul2, mol, &
79      first_spkt_harm(mol, v), t_end, delta_t, h_eul2_v5_file)
80
81  ! Berechnungen für Aufgabe 2: Morseoszillator von Molekül 2 mit Propagation
82  ! mit Euler erster Ordnung und Runge-Kutta vierter Ordnung
83  delta_t = T_v / 6000
84  write(*,*) 'Morse-Schrittweite:', delta_t
85      ! kleinere Schrittweite für Morseoszillator, damit Histogramm ordentlich
86      ! wird
87
88  ! Periodendauer des Morseoszillators numerisch ermitteln. (Muss genau bekannt
89  ! sein, damit das Histogramm hinhaut.)
90  v = 0
91  call propagator(punkte, last_ind, next_spkt_mors_ruk4, mol, &
92      first_spkt_M(mol, v), t_end, delta_t, m_ruk4_v0_file)
93  mors_period = obtain_spkt_period(punkte, 0, last_ind)
94
95  ! über genau fünf Perioden propagieren
96  t_end = 5 * mors_period
97  call propagator(punkte, last_ind, next_spkt_mors_eul1, mol, &
98      first_spkt_M(mol, v), t_end, delta_t, m_eul1_v0_file)
99  call propagator(punkte, last_ind, next_spkt_mors_ruk4, mol, &
100     first_spkt_M(mol, v), t_end, delta_t, m_ruk4_v0_file)
101
102 write(*,*) 'Periodendauer für mol 2 mit v=0:', mors_period
103
104 ! Histogramm ausgeben
105 call calc_histogram(punkte, 0, last_ind, delta_r, 'histogramm-v0.dat')
106
107 ! Periodendauer ermitteln
108 v = 5
109 call propagator(punkte, last_ind, next_spkt_mors_ruk4, mol, &
110     first_spkt_M(mol, v), t_end, delta_t, m_ruk4_v5_file)
111 mors_period = obtain_spkt_period(punkte, 0, last_ind)
112
113 ! über genau fünf Perioden propagieren
114 t_end = 5 * mors_period
115 call propagator(punkte, last_ind, next_spkt_mors_eul1, mol, &
116     first_spkt_M(mol, v), t_end, delta_t, m_eul1_v5_file)
117 call propagator(punkte, last_ind, next_spkt_mors_ruk4, mol, &
118     first_spkt_M(mol, v), t_end, delta_t, m_ruk4_v5_file)
119
120 write(*,*) 'Periodendauer für mol 2 mit v=5:', mors_period
121
122 ! Histogramm ausgeben
123 call calc_histogram(punkte, 0, last_ind, delta_r, 'histogramm-v5.dat')
124 end program svp_t
```

Listing 3: schwingungen.f90

```
! Berechnung von Radien und Energiewerten für Schwingungszustände sowie
! von Ort-Zeit-, Geschwindigkeit-Zeit- und Impuls-Zeit-Werten für
! Molekülschwingungen

module schwingungen
use teilchenparam
use konst_umr
use potentialfkten
use rk4_mod
use utils
implicit none

!! DECLARATIONS !!!!!!!!!!!!!!!
!! Typ schw_zust: enthält alle Parameter eines Schwingungszustands !!!!!!!!!!!
type :: SZST
    real(8) &
        R_tl,   & ! Radius des linken Umkehrpunkts
        R_tr,   & ! Radius des rechten Umkehrpunkts
        E_v       ! Energie des Schwingungszustands
end type SZST

!! Typ SPKT: enthält die Daten zu einem Punkt einer Schwingung
type :: SPKT
    real(8) :: &
        t,  & ! Zeitpunkt
        R,  & ! Ort (Radius)
        R_, & ! Geschwindigkeit R'
        P     ! Impuls
end type SPKT

contains

!! Funktion szst_h: Berechnung eines Schwingungszustands des harmonischen
!! Oszillators für gegebenes Molekül und Zustandsnummer
type(SZST) function szst_h(moldata, v)
implicit none
type(Mol2) :: moldata   ! Daten des Moleküls
real(8)    :: d_R_e     ! von R_e zu addierender/abzuziehender Wert
integer    :: v         ! gewünschter Schwingungszustand
    d_R_e = sqrt(2 * (v+0.5) * hquer * kreisfreq(moldata) &
                 / kraftkonst(moldata))

    szst_h%R_tl = moldata%R_e - d_R_e
    szst_h%R_tr = moldata%R_e + d_R_e
    szst_h%E_v  = V_h(moldata, szst_h%R_tl)
end function szst_h


!! Funktion szst_M: Berechnung eines Schwingungszustands es Morseoszillators
!! für gegebenes Molekül und Zustandsnummer
type(SZST) function szst_M(moldata, v)
implicit none
type(Mol2) :: moldata        ! Daten des zu berechnenden Moleküls
```

```
54  integer    :: v              ! gewünschter Schwingungszustand
55  real(8)    :: wurzel, omega  ! Dummy, Kreisfrequenz
56
57      ! Omega erhalten
58      omega  = kreisfreq(moldata)
59
60      ! Auslagerung der Wurzelberechnung
61      wurzel = sqrt(1/moldata%D * ((v + 0.5) * hquer * omega &
62                                   - (v + 0.5)**2 * (hquer * omega)**2 &
63                                      / (4 * moldata%D)))
64
65      ! Berechnung der Werte des Schwingungszustands
66      szst_M%R_tl = moldata%R_e - 1/moldata%beta * log(1 + wurzel)
67      szst_M%R_tr = moldata%R_e - 1/moldata%beta * log(1 - wurzel)
68      szst_M%E_v  = -moldata%D + (v + 0.5) * hquer * omega &
69                  - (v + 0.5)**2 * (hquer * omega)**2 / (4 * moldata%D)
70  end function szst_M
71
72
73  !! Funktion spkt_harm_arith: liefert eine Datenstruktur SPKT mit
74  !! den Schwingungswerten eines Moleküls zu einem Zeitpunkt beim
75  !! Schwingungszustand v nach dem Modell des harmonischen Oszillators.
76  !! Zur Berechnung wird die arithmetische Lösung benutzt.
77  type(SPKT) function spkt_harm_arith(moldata, v, t)
78  implicit none
79  type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
80  type(SZST) :: szst_1
81  real(8) :: &
82      t,       & ! Zeitpunkt
83      omega,   & ! Kreisfrequenz der Schwingung
84      R_tr!,    & ! Radius des rechten Umkehrpunkts
85  integer :: v   ! Schwingungszustand
86
87      ! Kreisfrequenz ermitteln
88      omega = kreisfreq(moldata)
89
90      ! Radius des rechten Umkehrpunkts ermitteln
91      szst_1 = szst_h(moldata, v)
92      R_tr   = szst_1%R_tr
93
94      ! gefragte Werte berechnen
95      spkt_harm_arith%t  = t   ! (!)
96      spkt_harm_arith%R  = moldata%R_e + (R_tr - moldata%R_e) * cos(omega * t)
97      spkt_harm_arith%R_ =   -omega    * (R_tr - moldata%R_e) * sin(omega * t)
98      spkt_harm_arith%P  = -moldata%my * omega * (R_tr - moldata%R_e) &
99                           * sin(omega * t)
100 end function spkt_harm_arith
101
102
103 !! Funktion first_spkt_M: liefert eine Datenstruktur SPKT mit den Schwingungs-
104 !! anfangswerten eines Moleküls für die üblichen Anfangsbedingungen bei dem
105 !! angegebenen Schwingungsniveau
106 !! t  = 0
107 !! R  = R_tr
```

```
!! R_ = 0
!! P  = 0
type(SPKT) function first_spkt_M(moldata, v)
implicit none
type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
type(SZST) :: zustand ! zwischendurch erhaltener Schwingungszustand
integer    :: v       ! Nummer des zu berechnenden Schwingungszustandes

    ! Schwingungszustand erhalten
    zustand = szst_M(moldata, v)

    ! Rückgabe generieren
    first_spkt_M%t  = 0
    first_spkt_M%R  = zustand%R_tr
    first_spkt_M%R_ = 0
    first_spkt_M%P  = 0
end function first_spkt_M


!! Funktion first_spkt_harm: gibt den ersten Schwingungspunkt (t=0) für einen
!! harmonischen Oszillator zurück
type(SPKT) function first_spkt_harm(moldata, v)
implicit none
type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
integer    :: v       ! gewünschtes Schwingungsniveau

    first_spkt_harm = spkt_harm_arith(moldata, v, 0d0)
end function first_spkt_harm

!! Funktion next_spkt_harm_eul1: propagiert die Schwingungsdaten des
!! harmonischen Oszillators zum Zeitpunkt t auf einen Zeitpunkt t+x
!! unter Benutzung der Euler'schen Näherung erster Ordnung
type(SPKT) function next_spkt_harm_eul1(moldata, old_spkt, delta_t)
implicit none
type(SPKT) :: old_spkt    ! Daten des vorherigen Punktes
type(Mol2) :: moldata     ! Daten des zu berechnenden Moleküls
real(8)    :: k, delta_t  ! Kraftkonstante, Zeitschritt

    ! Kraftkonstante ermitteln
    k = kraftkonst(moldata)

    ! gefragte Werte berechnen
    next_spkt_harm_eul1%t  = old_spkt%t + delta_t
    next_spkt_harm_eul1%R  = old_spkt%R + old_spkt%P/moldata%my * delta_t
    next_spkt_harm_eul1%P  = old_spkt%P - k * (old_spkt%R - moldata%R_e) &
                                        * delta_t
    next_spkt_harm_eul1%R_ = next_spkt_harm_eul1%P / moldata%my
end function next_spkt_harm_eul1


!! Funktion next_spkt_harm_eul2: propagiert die Schwingungsdaten des
!! harmonischen Oszillators zum Zeitpunkt t auf einen Zeitpunkt t+x
!! unter Benutzung der Euler'schen Näherung zweiter Ordnung
type(SPKT) function next_spkt_harm_eul2(moldata, old_spkt, delta_t)
```

```
162 implicit none
163 type(SPKT) :: old_spkt    ! Daten des vorherigen Punktes
164 type(Mol2) :: moldata     ! Daten des zu berechnenden Moleküls
165 real(8)    :: k, delta_t  ! Kraftkonstante, Zeitschritt
166
167     ! Kraftkonstante ermitteln
168     k = kraftkonst(moldata)
169
170     ! gefragte Werte berechnen
171     next_spkt_harm_eul2%t  = old_spkt%t + delta_t
172     next_spkt_harm_eul2%R  = old_spkt%R + old_spkt%P/moldata%my * delta_t &
173                              - delta_t**2/2 * k/moldata%my &
174                                * (old_spkt%R - moldata%R_e)
175     next_spkt_harm_eul2%P  = old_spkt%P - k * (old_spkt%R - moldata%R_e) &
176                                           * delta_t &
177                              - delta_t**2/2 * k * old_spkt%P / moldata%my
178     next_spkt_harm_eul2%R_ = next_spkt_harm_eul2%P / moldata%my
179 end function next_spkt_harm_eul2
180
181
182 ! Funktion next_spkt_mors_eul2: propagiert die Schwingungsdaten des
183 ! Morseoszillators zum Zeitpunkt t auf einen Zeitpunkt t+x unter Benutzung des
184 ! Eulerverfahrens zweiter Ordnung
185 ! Achtung: Hier stimmt etwas nicht. Allerdings wird die Funktion sowieso nicht
186 ! gebraucht.
187 type(SPKT) function next_spkt_mors_eul2(moldata, old_spkt, delta_t)
188 implicit none
189 type(SPKT) :: old_spkt    ! Daten des vorherigen Punktes
190 type(Mol2) :: moldata     ! Daten des zu berechnenden Moleküls
191 real(8)    :: delta_t     ! Zeitschritt
192
193     ! gefragte Werte berechnen
194     next_spkt_mors_eul2%t = old_spkt%t + delta_t
195     next_spkt_mors_eul2%R = old_spkt%R + old_spkt%P/moldata%my * delta_t &
196                              - delta_t**2/2 / moldata%my &
197                                           * V_M_(moldata, old_spkt%R)
198     next_spkt_mors_eul2%P = old_spkt%P &
199                              + delta_t * V_M_(moldata, old_spkt%R) &
200                              - delta_t**2/2 * V_M__(moldata, old_spkt%R) &
201                                           * old_spkt%P / moldata%my
202     next_spkt_mors_eul2%R_= next_spkt_mors_eul2%P / moldata%my
203 end function next_spkt_mors_eul2
204
205
206 ! Funktion next_spkt_mors_eul1: propagiert die Schwingungsdaten des
207 ! Morseoszillators zum Zeitpunkt t auf einen Zeitpunkt t+x unter Benutzung des
208 ! Eulerverfahrens erster Ordnung
209 type(SPKT) function next_spkt_mors_eul1(moldata, old_spkt, delta_t)
210 implicit none
211 type(SPKT) :: old_spkt    ! Daten des vorherigen Punktes
212 type(Mol2) :: moldata     ! Daten des zu berechnenden Moleküls
213 real(8)    :: delta_t     ! Zeitschritt
214
215     ! gefragte Werte berechnen
```

```
216     next_spkt_mors_eul1%t  = old_spkt%t + delta_t
217     next_spkt_mors_eul1%R  = old_spkt%R + old_spkt%P / moldata%my * delta_t
218     next_spkt_mors_eul1%P  = old_spkt%P - V_M_(moldata, old_spkt%R) * delta_t
219     next_spkt_mors_eul1%R_ = next_spkt_mors_eul1%P / moldata%my
220 end function next_spkt_mors_eul1
221
222 ! Mit generischen Euler-1- und Euler-2-Funktionen wie bei RK4 wäre das noch
223 ! viel hübscher gewesen...
224
225 !! Funktion next_spkt_mors_ruk4: Wrapperfunktion für RK4-Subroutine, die sich
226 !! wie die Funktionen aus schwingungen.f90 verhält. -- Propagiert einen
227 !! Schwingungspunkt des Morseoszillators nach dem Runge-Kutta-Verfahren
228 !! vierter Ordnung.
229 type(SPKT) function next_spkt_mors_ruk4(moldata, old_spkt, delta_t)
230 implicit none
231 type(SPKT) :: old_spkt    ! Daten des vorherigen Punktes
232 type(Mol2) :: moldata     ! Daten des zu berechnenden Moleküls
233 real(8)    :: t, delta_t  ! Zeit, Zeitschritt
234 integer, parameter     :: n = 2 ! Anzahl der Differentialgleichungen
235 real(8), dimension(n)  :: y     ! Array mit den Werten der abhängigen
236                                  ! Variablen
237
238     ! Zeit des alten Schwingungspunktes kopieren. -- Würde sonst verändert.
239     t = old_spkt%t
240
241     ! Array mit den Werten des vorherigen Punktes belegen
242     y = (/ old_spkt%R, old_spkt%P /)
243
244     ! ruk4-Subroutine aufrufen. Achtung: Das Ding modifiziert seine Parameter!
245     call rk4(moldata, t, y, delta_t, n)
246
247     ! neue Daten übernehmen
248     next_spkt_mors_ruk4%t = t
249     next_spkt_mors_ruk4%R = y(1)
250     next_spkt_mors_ruk4%P = y(2)
251     next_spkt_mors_ruk4%R_= y(2) / moldata%my
252 end function next_spkt_mors_ruk4
253
254 !! Subroutine propagator: Propagiert einen Punkt einer Schwingung mit dem !
255 !! angegebenen Verfahren bis zum Zeitpunkt t. Die ! Ergebnisse werden in eine
256 !! Datei geschrieben oder populieren das Array ! punkte.
257 subroutine propagator(punkte, iters, prop_func, moldata, startpunkt, t_end, &
258     delta_t, filename)
259 implicit none
260 type(SPKT), dimension(:), allocatable, intent(out) :: punkte
261     ! Array, das mit den Schwingungspunkten populiert wird
262 type(SPKT), external :: prop_func
263     ! propagierende Funktion
264 type(Mol2), intent(in) :: moldata
265     ! Daten des zu berechnenden Moleküls
266 type(SPKT), intent(in) :: startpunkt
267     ! Punkt, an dem die Propagation beginnen soll
268 real(8), intent(in) :: t_end
269     ! Zeit, bis zu der propagiert werden soll
```

```
integer, intent(out) :: iters
    ! Anzahl der Iterationen. -- Wird zurückgegeben, damit nachfolgende
    ! Programmteile beispielsweise obtain_spkt_period mit passendem Start- und
    ! Endindex aufrufen können.
real(8), intent(in) :: delta_t
    ! Schrittweite der Propagation
character(len=*), intent(in), optional :: filename
character(len=36) :: ERROR = 'ERROR in subroutine call propagator:'
    ! message for the user
integer :: allocate_stat ! status of the allocation operation
! Schleifenvariablen
integer :: i ! Iterator
real(8) :: t ! aktuelles Zeitargument

    ! Fehlenden Parameter iters berechnen.
    iters = iterationen(startpunkt%t, t_end, delta_t)

    ! Array punkte festlegen
    allocate(punkte(0:iters), STAT=allocate_stat)
    if (allocate_stat > 0) write(*,*) ERROR, 'Allocation failed.'

    ! Anfangspunkt in punkte speichern
    punkte(0) = startpunkt

    ! Propagieren
    do i=1,iters
        ! Zeitargument berechnen
        t = i * delta_t

        ! Array populieren
        punkte(i) = prop_func(moldata, punkte(i-1), delta_t)
    enddo

    ! auf Wunsch Daten in Datei schreiben
    if (present(filename)) then
        open(unit=45, file=filename, action='write', status='replace', &
             form='formatted')
            ! In der Hoffnung, dass niemand anderes FD 45 benutzt. -- 42 wäre
            ! gefährlich.
        call write_spkt_array(punkte, 0, iters, 45)
        close(45)
    end if
end subroutine propagator


!! Subroutine spkt_harm_array: Populiert ein Array mit den Werten des
!! harmonischen Oszillators unter Nutzung der arithmetischen Lösung
subroutine spkt_harm_array(punkte, moldata, v, t_0, t_end, delta_t, filename)
implicit none
type(SPKT), dimension(:), allocatable, intent(out) :: punkte
    ! Array, das mit den Schwingungspunkten populiert wird
type(Mol2), intent(in) :: moldata
    ! Daten des zu berechnenden Moleküls
real(8), intent(in) :: t_0, t_end
```

```
324     ! Anfangs- und Endzeit der Berechnung
325 integer :: iters, v
326     ! Anzahl der Iterationen, gewünschtes Schwingungsniveau
327 real(8), intent(inout), optional :: delta_t
328     ! Schrittweite der Zeitwerte
329 character(len=*), intent(in), optional :: filename
330 character(len=36) :: ERROR = 'ERROR in subroutine call propagator:'
331     ! message for the user
332 integer :: allocate_stat ! status of the allocation operation
333 ! Schleifenvariablen
334 integer :: i ! Iterator
335 real(8) :: t ! aktuelles Zeitargument
336
337     ! Fehlenden Parameter iters berechnen.
338     iters = iterationen(t_0, t_end, delta_t)
339
340     ! Array punkte festlegen
341     allocate(punkte(0:iters), STAT=allocate_stat)
342     if (allocate_stat > 0) write(*,*) ERROR, 'Allocation failed.'
343
344     ! Propagieren
345     do i=0,iters
346         ! Zeitargument berechnen
347         t = i * delta_t
348
349         ! Array populieren
350         punkte(i) = spkt_harm_arith(moldata, v, t)
351     enddo
352
353     ! auf Wunsch Daten in Datei schreiben
354     if (present(filename)) then
355         open(unit=45, file=filename, action='write', status='replace', &
356              form='formatted')
357             ! In der Hoffnung, dass niemand anderes FD 45 benutzt. -- 42 wäre
358             ! gefährlich.
359         call write_spkt_array(punkte, 0, iters, 45)
360         close(45)
361     end if
362 end subroutine spkt_harm_array
363
364 ! Subroutine find_local_spkt_min: durchsucht eine Liste von SPKTen nach einem
365 ! lokalen Minimum (Radius/Auslenkung) und gibt den Minimumspunkt und dessen
366 ! Index zurück.
367 subroutine find_local_spkt_min(punkte, startind, endind, minpunkt, index)
368 implicit none
369 type(SPKT), dimension (:), intent(in) :: punkte
370     ! Array, in dem gesucht werden soll
371 integer, intent(in) :: startind, endind
372     ! Arrayindizes, bei denen die Suche startet bzw. erfolglos beendet wird
373 type(SPKT), intent(out) :: minpunkt
374     ! zurückgegebener Minimumspunkt
375 integer, intent(out) :: index
376     ! Index des Minimumspunktes im Eingabearray
377
```

```
integer :: i
    ! Iterator

    ! finden und zurückgeben des Minimums -- Die Ordinate eines lokalen
    ! Minimums ist kleiner als die Ordinaten beider Nachbarpunkte.
    do i=startind,endind
        ! Falls Minimumspunkt gefunden, Ausgabevariablen belegen und abhauen
        if ((punkte(i)%R < punkte(i-1)%R) &
             .and. (punkte(i)%R < punkte(i+1)%R)) then
            minpunkt = punkte(i)
            index    = i
            return
        end if
    end do

   ! Ausführung erreicht diesen Punkt nur, wenn kein Minimum gefunden wurde.
   write(*,*) 'ERROR in subroutine find_local_spkt_min: No local minimum', &
              ' found.'
        ! Aber kein Stop! -- Es könnte ja noch woanders lokale Minima geben.
end subroutine find_local_spkt_min

! Funktion obtain_spkt_period: Gibt die Periodendauer eines Arrays von SPKTen
! (Schwingung) zurück
real(8) function obtain_spkt_period(punkte, startind, endind)
implicit none
integer :: startind, endind
    ! Arrayindizes, zwischen denen gesucht werden soll
type(SPKT), dimension(startind:endind) :: punkte
    ! Array, in dem gesucht werden soll
type(SPKT) :: spkt_min_1, spkt_min_2
    ! im Array zu findende Minimumspunkte
integer :: index
    ! Variable, in der der Index des ersten Minimumspunktes gespeichert wird.
    ! (Der zweite ist belanglos.)

    ! zwei Minimumspunkte (Radius/Auslenkung) finden
    call find_local_spkt_min(punkte, startind, endind, spkt_min_1, index)
    call find_local_spkt_min(punkte, index + 1, endind, spkt_min_2, index)

    ! Zeitdifferenz zwischen Minimumspunkten ist gleich der Periodendauer
    obtain_spkt_period = spkt_min_2%t - spkt_min_1%t
end function obtain_spkt_period

! Subroutine calc_histogram: berechnet aus einem Array von SPKTen ein
! Histogramm mit der Häufigkeitsverteilung der Radien einer Schwingung
!
! Arbeitsweise: Die Radien der SPKTe werden durch die Intervallbreite geteilt
! und aufgerundet. Die entstehenden ganzzahligen Werte werden als Indizes für
! das Array mit den absoluten Häufigkeiten der Radien in den Intervallen
! benutzt. So bekommt man eine lineare Laufzeit.
subroutine calc_histogram(punkte, startind, endind, delta_r, filename)
implicit none
integer :: &
    startind, endind
```

```
432     ! Grenzen des punkte-Arrays. Es gibt hier eine Inkonsistenz unter meinen
433     ! Programmen, die mit den ekligen statischen Arrays zu tun hat: Mal sind
434     ! startind und endind nur Anfangs- und Endwert der Iteration über ein
435     ! Array, mal werden sie tatsächlich zur Allokation verwendet. In diesem
436     ! Bereich können noch mehr Probleme auftreten, aber wir beten alle zu
437     ! Athene, dass sie das nicht tun. (Eine Fehlerursache ist bspw., dass ich
438     ! meine Arrayindizes mit 0 angefangen habe, aber Fortran normalerweise bei
439     ! 1 anfängt.)
440 type(SPKT), dimension(startind:endind) :: punkte
441 real(8) :: &
442     delta_r, &
443     ! Breite der Intervalle für die Aufenthaltshäufigkeit
444     r_max, &
445     r_min
446     ! größter und kleinster Radius im punkte-Array
447 character(len=*) :: filename
448     ! Name der Datei für die Histogrammdaten
449 real(8), dimension(startind:endind) :: radien
450 integer, dimension(:), allocatable :: haeufigktn
451     ! Array, die Häufigkeiten in den Intervallen
452 integer :: &
453     i, &
454     ! Iterator
455     interval_num, &
456     ! Anzahl der Intervalle im Histogramm
457     i_min, i_max, &
458     ! kleinster und größter Intervallindex
459     interv_ind
460     ! Zwischenspeicher für den aktuellen Intervallindex in der Zählschleife
461
462     ! Radien der SPKTe in eigenes Array speichern. (minval und maxval
463     ! funktionieren nicht mit SPKTen)
464     do i = startind, endind
465         radien(i) = punkte(i)%R
466     enddo
467
468     ! Anzahl der Intervalle berechnen, die bei einer Radiusschrittweite von
469     ! delta_r nötig sind, um alle Punkte vom kleinsten bis zum größten Radius
470     ! zu erfassen.
471     r_max = maxval(radien)
472     r_min = minval(radien)
473     interval_num = ceiling((r_max - r_min) / delta_r)
474
475     ! Grenzen des Intervallarrays berechnen
476     i_min = ceiling(r_min / delta_r)
477     i_max = i_min + interval_num
478
479     ! Array für Häufigkeiten allozieren und mit Nullen belegen
480     allocate( haeufigktn(i_min:i_max) )
481     haeufigktn = (/ (0, i = i_min, i_max) /)
482
483     ! über Radien iterieren
484     do i = startind, endind
485         ! normieren und aufrunden
```

```
486         interv_ind = ceiling(radien(i) / delta_r)
487
488         ! Häufigkeitswert des entsprechenden Intervalls inkrementieren
489         haeufigktn(interv_ind) = haeufigktn(interv_ind) + 1
490     end do
491
492     ! Ausgabedatei öffnen
493     open(unit=51, file=filename, action='write', status='replace', &
494          form='formatted')
495
496     write(*,*) 'Punkte insgesamt:', sum(haeufigktn)
497
498     ! Häufigkeiten zusammen mit Intervallschritten ausgeben
499     do i = i_min, i_max
500         write(51, *) i * delta_r, haeufigktn(i)
501     end do
502
503     ! Ausgabedatei schließen
504     close(51)
505 end subroutine calc_histogram
506
507
508 !! Funktion write_spkt_array: Schreibt ein Array von SPKTen von Index start
509 !! bis Index ende zeilenweise an das angegebene Filehandle
510 subroutine write_spkt_array(array, start, ende, deskriptor)
511 implicit none
512 integer, intent(in) :: start, ende, deskriptor
513     ! Start- und Endindex des auszugebenden Stücks Array, Filedeskriptor
514 type(SPKT), intent(in), dimension(:) :: array
515     ! auszugebendes Array
516 integer :: i
517     ! Schleifenvariable
518
519     ! ausgeben
520     do i=start,ende
521         write(deskriptor,*) array(i)
522     enddo
523 end subroutine write_spkt_array
524
525 end module schwingungen
```

An dieser stelle fehlt ein Listing für die Datei `rk4-mod.f90`, da diese Code von Axel Schild, einem der Kursleiter, enthält. Es handelt sich dabei um eine Implementation des Runge-Kutta-Verfahrens vierter Ordnung.

Listing 4: leps-gen.f90

```
1 ! Programm, das die LEPS-Potential-Gitter generiert
2
3 program leps_gen
4 use konst_umr
5 use teilchenparam
6 use leps_mod
7 use traj_mod
8 use utils
```

```
9  implicit none
10
11 !! DECLARATIONS !!!!!!!!!!!!!!
12 real(8), parameter :: &
13     r_0     = 0,   &
14     r_end   = 8,   & ! Ausdehnung des Gitters
15     delta_r = 0.01,& ! Abstand der Gitterlinien
16     V_L_max = 3,   & ! Leps-Potentialwert, auf den alle gesetzt werden
17                      ! sollen, die größer sind. (Muss höher sein als der
18                      ! Maximalwert bei den Höhenlinien von Gnuplot, da diese
19                      ! sonst gewellt sind.)
20     delta_V = 0.3 * eV, &
21                      ! Differenz zwischen LEPS- und Morsepotential, ab der der
22                      ! Bereich im Gitter als asymptotisch angesehen wird.
23     delta_R_as = 0.1 ! Differenz zwischen Bindungslänge und
24                      ! Gleichgewichtsradius Bereich im Gitter als
25                      ! asymptotisch angesehen wird.
26 type(LGPT), dimension(:,:), allocatable :: &
27     leps_grid_unw, & ! Array, in dem das ungewichtete Gitter gespeichert wird
28     leps_grid_wei    ! Array, in dem das gewichtete Gitter gespeichert wird
29 type(Mol2), parameter :: &
30     mol_1 = MCl, &    ! die zu verwendenden Moleküle -- mol_2 ist Ausgangs-
31     mol_2 = Cl2       ! stoff, HF Reaktionsprodukt.
32 character(len=17) :: &
33     leps_grid_unw_file = 'leps-grid-unw.dat', &
34     leps_grid_wei_file = 'leps-grid-wei.dat'
35                     ! Dateien, in die die LEPS-Daten geschrieben werden
36 integer :: i_max, j_max
37                     ! Grenzen der Gitterarrays
38
39 type(TPKT) :: trajstart_v0, trajstart_v5
40     ! Startpunkte der Trajektorien für v=0 und v=5
41 real(8), parameter :: &
42     x_0          = 8, &
43     ! Abszissenwert für den Anfang der ersten Trajektorienpropagation
44     E_0          = -3.1, &
45     ! Energieanfangswert für die Trajektorenpropagationen
46     delta_t      = 0.05, &
47     ! Schrittweite der Trajektorienpropagation
48     deriv_delta  = 0.01, &
49     ! Schrittweite der Näherung der Ableitung des LEPS-Potentials
50     periodend_v0 = 5.9725344560839551, &
51     periodend_v5 = 6.4237398424282590
52     ! Periodendauern von Cl2 für v=0 und v=5 (aus svp-t)
53 integer, parameter :: tpkt_anz = floor(1000 / delta_t)
54     ! Ungefähre Anzahl der Punkte einer Trajektorie. -- Gehe davon aus, dass
55     ! eine Reaktion nicht länger als 200 Zeiteinheiten dauert.
56 type(TPKT), dimension(0:tpkt_anz) :: trajpunkte_v0, trajpunkte_v5
57     ! Array für eine Trajektorie
58 character(len=11), parameter :: &
59     traj_file_v0 = 'traj-v0.dat', &
60     traj_file_v5 = 'traj-v5.dat'
61     ! Dateien, in denen die Trajektoriendaten für v=0 und v=5 gespeichert
62     ! werden
```

```
63  type(LGPT) :: y_asympt
64      ! LEPS-Gitterpunkt, der als Beginn des asymptotischen Bereiches
65      ! ausgerechnet wurde
66  real(8) :: x_dist_v0, x_dist_v5
67      ! Weg, den das Teilchen mol_2 während einer Schwingungsperiode in
68      ! x-Richtung zurücklegt
69  integer :: i
70      ! Iterator
71
72  ! ungewichtetes Gitter berechnen
73  call leps_grid(leps_grid_unw, leps, mol_1, mol_2, r_0, r_end, &
74                 delta_r, V_L_max, leps_grid_unw_file)
75  ! gewichtetes Gitter berechnen
76  call leps_grid(leps_grid_wei, leps_wei, mol_1, mol_2, r_0, r_end, &
77                 delta_r, V_L_max, leps_grid_wei_file, i_max, j_max)
78
79
80  write(*,*) 'Asympt. Bereich in x- und y-Richtung für gewichtetes Gitter.'
81  write(*,*) find_asympt_ber(leps_grid_wei, i_max, j_max, 'x', mol_2, delta_V, &
82                             delta_R)
83  y_asympt = find_asympt_ber(leps_grid_wei, i_max, j_max, 'y', mol_1, delta_V, &
84                             delta_R)
85  write(*,*) y_asympt
86      ! Sollte nicht zu ernst genommen werden.
87
88  write(*,*) 'Sattelpunkt im gewichteten Gitter.'
89  write(*,*) find_sattelpunkt(leps_grid_wei, i_max, j_max)
90  write(*,*) 'Sattelpunkt im ungewichteten Gitter.'
91  write(*,*) find_sattelpunkt(leps_grid_unw, i_max, j_max)
92
93  ! möglicherweise vorhandene Dateien mit Trajektoriendaten löschen
94  call delete_file(traj_file_v0)
95  call delete_file(traj_file_v5)
96
97  ! erste Startpunkte für Propagationen finden
98  trajstart_v0 = gen_traj_start(mol_2, x_0 = x_0, v = 0, E_0 = E_0)
99  trajstart_v5 = gen_traj_start(mol_2, x_0 = x_0, v = 5, E_0 = E_0)
100
101 ! Strecke berechnen, die das Teilchen während einer Schwingungsperiode in x
102 ! zurücklegt (für v=0 und v=5)
103 x_dist_v0 = abs(trajstart_v0%p_x * periodend_v0 / mol_2%my)
104 x_dist_v5 = abs(trajstart_v5%p_x * periodend_v5 / mol_2%my)
105
106 ! für jeden Schwingungszustand zehn Trajetorien berechnen
107 do i = 0, 9
108     ! Startpunkte für nächste Propagation finden
109     trajstart_v0 = gen_traj_start(mol_2, x_0 = x_0 + i/11d0 * x_dist_v0, &
110                                   v = 0, E_0 = E_0)
111     trajstart_v5 = gen_traj_start(mol_2, x_0 = x_0 + i/11d0 * x_dist_v5, &
112                                   v = 5, E_0 = E_0)
113
114     ! Trajektorien berechnen
115     call calc_traj(trajpunkte_v0, tpkt_anz, mol_1, mol_2, trajstart_v0, &
116                    r_end, delta_t, deriv_delta, traj_file_v0)
```

```
117     call calc_traj(trajpunkte_v5, tpkt_anz, mol_1, mol_2, trajstart_v5, &
118                    r_end, delta_t, deriv_delta, traj_file_v5)
119 end do
120
121 end program leps_gen
```

Listing 5: leps-mod.f90

```
1  ! Funktionen zur Berechnung des LEPS-Potentials
2
3  module leps_mod
4
5  use teilchenparam
6  use potentialfkten
7  use utils
8  IMPLICIT NONE
9
10 !! Typ LGPT: Punkt im LEPS-Potentialgitter
11 type :: LGPT
12     real(8) :: &
13         x,  & ! Position in x-Richtung
14         y,  & ! Position in y-Richtung
15         V_L   ! LEPS-Potential in dem Punkt
16 end type LGPT
17
18
19 ! überladen der Vergleichsoperation zur Nutzung mit LGPTen
20 interface operator (>)
21     module procedure lgpt_gt
22 end interface
23
24 interface operator (==)
25     module procedure lgpt_eq
26 end interface
27
28 interface operator (<)
29     module procedure lgpt_lt
30 end interface
31
32 CONTAINS
33
34 ! ------------------------------------------------------------------
35 ! -- leps
36 ! --
37 ! --   IN      :     DAB          : A-B distance (in Angstrom)
38 ! --                 DBC          : B-C distance (in Angstrom)
39 ! --                 DAC          : A-C distance (in Angstrom)
40 ! --
41 ! --   RET: LEPS potential value (in kJ/mol)
42 ! ++        Achtung: Einheit hängt ab von der Einheit von D!
43 ! --
44 ! -- Purpose:
45 ! --   Calculate three-body A-B-C LEPS potential for the given
46 ! --   distances (DAB, DBC, DAC)
```

```
47  ! --
48  ! -- References:
49  ! --
50  ! --   [1]    :   Jonathan et al., Mol. Phys. vol. 24 (1972) 1143
51  ! --   [2]    :   Jonathan et al., Mol. Phys. vol. 43 (1981) 215
52  ! ------------------------------------------------------------------
53  REAL(8) FUNCTION leps( mol_1, mol_2, DAB, DBC )
54
55    REAL(8) :: DAB, DBC, DAC    ! -- input
56      ! Abstände der Teilchen
57
58    type(Mol2), intent(in) :: mol_1, mol_2
59      ! Daten der zu berechnenden Moleküle
60
61    ! komische Variablen in dieser Funktion
62    real(8), dimension(1:3) ::  D, beta, REQ, delta
63
64    REAL(8) :: Q(3), J(3), S(3)
65    INTEGER :: k
66
67    ! in dieser Funktion genutzte Variablen mit den Molekülparametern
68    ! initialisieren
69    D     = (/ mol_1%D,     mol_2%D,     mol_1%D /)
70    beta  = (/ mol_1%beta,  mol_2%beta,  mol_1%beta /)
71    REQ   = (/ mol_1%R_e,   mol_2%R_e,   mol_1%R_e /)
72    delta = (/ mol_1%delta, mol_2%delta, mol_1%delta /)
73
74    ! wir verzichten auf nicht-kollineare Konstellationen
75    DAC = DAB + DBC
76
77    ! -- calculate integrals
78    CALL integrals( D(1), beta(1), Delta(1), DAB - REQ(1), Q(1), J(1) )
79    CALL integrals( D(2), beta(2), Delta(2), DBC - REQ(2), Q(2), J(2) )
80    CALL integrals( D(3), beta(3), Delta(3), DAC - REQ(3), Q(3), J(3) )
81
82    ! -- put it al together
83    S = 1.0D0 + Delta
84    leps = Q(1) / S(1) + Q(2) / S(2) + Q(3) / S(3)  &
85         - SQRT( J(1)**2 / S(1)**2 + J(2)**2 / S(2)**2 &
86                 + J(3)**2 / S(3)**2 &
87                 - J(1) * J(2) / S(1) / S(2) &
88                 - J(1) * J(3) / S(1) / S(3) &
89                 - J(2) * J(3) / S(2) / S(3) )
90  END FUNCTION  ! -- leps
91
92
93  ! ------------------------------------------------------------------
94  ! -- integrals
95  ! --
96  ! --   IN      :     D           : energy of dissociation
97  ! --                 beta        : beta parameter
98  ! --                 Delta       : Sato parameter
99  ! --                 R           : position (rel. to equilibrium)
100 ! --
```

```
101 ! --   OUT     :     Q            : Coulomb integral
102 ! --                 J            : Exchange integral
103 ! --
104 ! -- Purpose:
105 ! --   Calculate coulomb- and exchange integrals.
106 ! --   The dissociation energy of the tripplet state is set
107 ! --   to: D^3 = 0.5 * D^1.
108 ! ----------------------------------------------------------------------
109 SUBROUTINE integrals(D, beta, Delta, R, Q, J)
110
111   REAL(8) :: D, beta, Delta, R   ! -- input
112   REAL(8) :: Q, J                ! -- output
113
114   REAL(8) :: A, E1, E3
115
116   ! -- morse potentials
117   A = EXP( -beta * R )
118   ! ---- binding
119   E1 = D * ( A*A - 2.0D0 * A )
120   ! ---- anti-binding
121   E3 = 0.5D0 * D * ( A*A + 2.0D0 * A )
122
123
124   ! -- integrals
125   Q = 0.5D0 * ( E1 * (1.0D0 + Delta) + E3 * (1.0D0 - Delta) )
126   J = 0.5D0 * ( E1 * (1.0D0 + Delta) - E3 * (1.0D0 - Delta) )
127
128 END SUBROUTINE  ! -- integrals
129
130
131 !! Funktion leps_wei: Berechnet das LEPS-Potential, nachdem es die Koordinaten
132 !! nach Masse gewichtet hat.
133 real(8) function leps_wei(mol_1, mol_2, X, Y)
134
135     implicit none
136     ! Die Reaktion sieht so aus: A + BB -> AB + B
137     !                            1 + 23 -> 12 + 3
138     ! Es gilt also: m(AB) = m(A) + m(B), m(BB) = 2 * m(B)
139     !            => m(A)  = m(AB) - m(B) = m(AB) - 1/2 m(BB)
140     !           und m(B)  = 1/2 m(BB)
141     type(Mol2), intent(in) :: mol_1, mol_2
142         ! Daten der zu berechnenden Moleküle
143     real(8), intent(in) :: X, Y
144         ! Koordinaten des zu berechnenden Punktes
145     real(8) :: &
146         m1, m3, m23, m123, d12, d23, d13, d123
147
148     m1  = mol_1%M - 0.5 * mol_2%M
149     m3  = 0.5 * mol_2%M
150     m23 = mol_2%my
151     m123 = m1*(mol_2%M)/(m1+mol_2%M)
152
153     d23  = Y
154     d123 = sqrt(m23/m123)*X
```

```
155     d12  = d123-m3/(mol_2%M)*d23
156     d13  = d12+d23
157
158     leps_wei = leps(mol_1, mol_2, d12, d23)
159 end function leps_wei
160
161 ! Funktion leps_wei_x_: Berechnet Näherung (Fehler ist Schrittweite^4) der
162 ! Ableitung in x-Richtung des gewichteten LEPS-Potentials
163 real(8) function leps_wei_x_(mol_1, mol_2, x, y, delta_x)
164 implicit none
165 type(Mol2), intent(in) :: mol_1, mol_2
166     ! Daten der zu berechnenden Moleküle
167 real(8) :: &
168     x, y, &
169     ! Koordinaten des zu berechnenden Punktes
170     delta_x
171     ! Schrittweite
172
173     leps_wei_x_ = ( &
174                             leps_wei(mol_1, mol_2, x - 2*delta_x, y) &
175                       - 8 * leps_wei(mol_1, mol_2, x -   delta_x, y) &
176                       + 8 * leps_wei(mol_1, mol_2, x +   delta_x, y) &
177                       -     leps_wei(mol_1, mol_2, x + 2*delta_x, y) &
178                   ) / 12 / delta_x
179 end function leps_wei_x_
180
181 ! Funktion leps_wei_y_: Berechnet Näherung (Fehler ist Schrittweite^4) der
182 ! Ableitung in y-Richtung des gewichteten LEPS-Potentials
183 ! (Ugh, Copy & Paste.)
184 real(8) function leps_wei_y_(mol_1, mol_2, x, y, delta_y)
185 implicit none
186 type(Mol2), intent(in) :: mol_1, mol_2
187     ! Daten der zu berechnenden Moleküle
188 real(8) :: &
189     x, y, &
190     ! Koordinaten des zu berechnenden Punktes
191     delta_y
192     ! Schrittweite
193
194     leps_wei_y_ = ( &
195                             leps_wei(mol_1, mol_2, x, y - 2*delta_y) &
196                       - 8 * leps_wei(mol_1, mol_2, x, y -   delta_y) &
197                       + 8 * leps_wei(mol_1, mol_2, x, y +   delta_y) &
198                       -     leps_wei(mol_1, mol_2, x, y + 2*delta_y) &
199                   ) / 12 / delta_y
200 end function leps_wei_y_
201
202 ! Subroutine leps_grid: Berechnet ein zweidimensionales Array von LGPTen und
203 ! gibt es auf Wunsch in eine Datei aus. Je nach dem, welche leps_func
204 ! verwendet wird, sind die Koordinaten massengewichtet oder nicht.
205 subroutine leps_grid(gitter, leps_func, mol_1, mol_2, r_0, r_end, &
206                      delta_r, V_L_max, filename, i_max, j_max)
207 type(LGPT), dimension(:,:), intent(out), allocatable :: gitter
208     ! zweidimensionales Array, das die LGPTe enthält
```

```
209 real(8) :: leps_func
210     ! Funktion, mit der das LEPS-Potential berechnet werden soll
211 type(Mol2), intent(in) :: mol_1, mol_2
212     ! Daten der zu berechnenden Moleküle
213 real(8), intent(in) :: r_0, r_end, delta_r
214     ! Anfangs- und Endradien des Gitters und Abstand der Gitterlinien
215 real(8), intent(in) :: V_L_max
216     ! Potentialwert, ab dem das Potential der Gitterpunkte auf den Maximalwert
217     ! gesetzt werden sollen
218 character(len=*), intent(in), optional :: filename
219     ! Name der Datei, in die die Daten geschrieben werden sollen
220 integer, intent(out), optional :: i_max, j_max
221
222 REAL(8) :: A, B, V_L
223 REAL(8) :: A1, A2, B1, B2
224     ! das gleiche von Axel, bloß einzeln für x- und y-Richtung
225 INTEGER :: N1, N2
226     ! Anzahl der Linien, die er in jede Richtung produziert, bzw. Ausdehnung
227     ! des Gitterarrays.
228 INTEGER :: j, k
229     ! Iteratorvariablen
230
231     ! Gittergrenzen festlegen
232     A1 = r_0
233     A2 = r_end
234     B1 = r_0
235     B2 = r_end
236
237     ! Anzahl der Linien berechnen
238     N1 = iterationen(r_0, r_end, delta_r)
239     N2 = N1
240
241     ! Größe von gitter festlegen
242     allocate(gitter(1:N1, 1:N2))
243
244     ! Gitter populieren
245     DO j = 1, N1
246
247         A = (j - 1) * (A2 - A1) / (N1 - 1) + A1
248
249         DO k = 1, N2
250
251             B = (k - 1) * (B2 - B1) / (N2 - 1) + B1
252
253             V_L = leps_func( mol_1, mol_2, A, B )
254
255             ! Potential auf Maximalwert setzen, wenn dieser überschritten wird
256             IF ( V_L .GT. V_L_max ) THEN
257                 V_L = V_L_max
258             END IF
259
260             ! Punkt in gitter schreiben
261             gitter(j, k) = LGPT(A, B, V_L)
262         END DO  ! -- k
```

```
263     END DO  ! -- j
264
265     ! auf Verlangen des Benutzers Punkte in Datei ausgeben
266     if (present(filename)) then
267         open(unit=46, file=filename, action='write', status='replace', &
268              form='formatted')
269         call write_lgpt_array(gitter, N1, N2, 46)
270         close(46)
271     end if
272
273     ! auf Wunsch des Benutzers Grenzen des Gitterarrays zurückgeben
274     if (present(i_max) .and. present(j_max)) then
275         i_max = N1
276         j_max = N2
277     end if
278 end subroutine
279
280
281 !! Funktion find_local_lgpt_min: durchsucht eine Liste von LGPTen nach einem
282 !! lokalen Minimum und gibt den Minimumspunkt zurück
283 type(LGPT) function find_local_lgpt_min(punkte, startind, endind)
284 implicit none
285 type(LGPT), dimension(:) :: punkte
286 integer :: &
287     startind, & ! Arrayindex, bei dem die Suche startet
288     endind,   & ! Arrayindex, bei dem die Suche beendet wird
289     i           ! Iterator
290
291     ! finden und zurückgeben des Minimums -- Die Ordinate eines lokalen
292     ! Minimums ist kleiner als die Ordinaten beider Nachbarpunkte.
293     do i=startind,endind
294         if ((punkte(i)%V_L < punkte(i-1)%V_L) &
295              .and. (punkte(i)%V_L < punkte(i+1)%V_L)) then
296             find_local_lgpt_min = punkte(i)
297             return
298         end if
299     enddo
300
301     ! Ausführung erreicht diesen Punkt nur, wenn kein Minimum gefunden wurde.
302   !write(*,*) 'ERROR in function find_local_lgpt_min: No local minimum found.'
303         ! Aber kein Stop! -- Es könnte ja noch woanders lokale Minima geben.
304 end function find_local_lgpt_min
305
306
307 ! Funktion find_sattelpunkt: durchsucht ein zweidimensionales Feld von LGPTen
308 ! nach einem Sattelpunkt (Energiebarriere)
309 type(LGPT) function find_sattelpunkt(gitter, i_max, j_max)
310 implicit none
311 type(LGPT), dimension(:,:) :: gitter
312     ! enthält die Punkte der LEPS-Potentialfläche, unter denen der Sattelpunkt
313     ! gefunden werden soll.
314 integer :: i_max, j_max, i, j
315     ! Ausdehnung des Arrays mit den Punkten (Startindizes immer 1) und
316     ! Iteratoren
```

```
317
318     ! finden und zurückgeben des Sattelpunkts -- in x-Richtung (i) ist er
319     ! Maximum, in y-Richtung (j) ist er Minimum
320     do i=2,i_max
321         do j=2,j_max
322             ! Testen, ob es sich um ein Maximum handelt
323             if ((gitter(i,j) < gitter(i, j-1)) &
324                 .and. (gitter(i,j) < gitter(i,   j+1)) &
325                 .and. (gitter(i,j) > gitter(i-1, j))   &
326                 .and. (gitter(i,j) > gitter(i+1, j))) then
327
328                 find_sattelpunkt = gitter(i,j)
329                 return
330             end if
331         enddo
332     enddo
333
334 !    write(*,*) 'Kein Sattelpunkt gefunden.'
335 !    stop
336 end function find_sattelpunkt
337
338
339 !! Funktion write_lgpt_array: Schreibt ein zweidimensionales Array in den
340 !! Grenzen i_min, i_max, j_min, j_max an den angegebenen Dateideskriptor
341 subroutine write_lgpt_array(array, i_max, j_max, deskriptor)
342 implicit none
343 type(LGPT), intent(in), dimension(:,:) :: array
344     ! auszugebendes Array
345 integer, intent(in) :: i_max, j_max
346     ! Grenzen der Ausgabe
347 integer, intent(in) :: deskriptor
348     ! Filedeskripter, zu dem geschrieben werden soll
349 integer :: i, j
350     ! Iteratoren
351
352     ! über eine Dimension iterieren
353     do i=1,i_max
354         ! über zweite Dimension iterieren und in die Datei schreiben
355         do j=1,j_max
356             write(deskriptor,*) array(i,j)
357         enddo
358
359         ! nachdem eine Arrayzeile/-spalte abgearbeitet ist, Leerzeile
360         write(deskriptor,*)
361     enddo
362 end subroutine write_lgpt_array
363
364
365 !! Funktion find_asympt_ber: durchsucht ein gegebenes Array von LGPTen nach
366 !! dem Anfang des Bereiches, der nach den gegebenen Kriterien asymptotisch ist
367 type(LGPT) function find_asympt_ber(inp_gitter, i_max, j_max, testrichtung, &
368     moldata, delta_V, delta_R)
369 implicit none
370 integer, intent(in) :: i_max, j_max
```

```
371     ! obere Grenzen des Nachschauens im Array (untere sind per Konvention 1)
372 type(LGPT), intent(in), dimension(i_max,j_max) :: inp_gitter
373     ! Array, das die Gitterdaten enthält
374 type(LGPT), dimension(i_max,j_max) :: gitter
375     ! Arbeitskopie des Arrays, das die Gitterdaten enthält
376 character(len=1) :: testrichtung
377     ! Gitter wird entweder in x-Richtung (Schnitte parallel zur y-Achse) oder
378     ! in y-Richtung (Schnitte parallel zur x-Achse) durchlaufen
379 type(Mol2) :: moldata
380     ! Daten des zu berechnenden Moleküls (muss mit schnittrichtung abgestimmt
381     ! sein!)
382 real(8) :: delta_V, delta_R
383     ! Werte für die Differenzen zwischen LEPS- und Morsepotential und zwischen
384     ! Gleichgewichts- und aktuellem Bindungsabstand, ab denen der Bereich als
385     ! asymptotisch angesehen wird
386
387 type(LGPT) :: minpunkt
388     ! In der Schleife verwendete temporäre Variable. Wenn auf sie die
389     ! Kriterien für den asymptotischen Bereich zutreffen, wird sie zum
390     ! Funktionswert.
391 real(8) :: R
392     ! In der Schleife verwendete temporäre Variable für den Abstand, in dessen
393     ! Richtung nicht getestet wird.
394 integer :: i, j
395     ! Iteratoren
396
397     ! Wenn in y-Richtung getestet wird, ist es am einfachsten, die Matrix zu
398     ! transponieren und auf die gleiche Weise durchzugehen, wie in x-Richtung
399     if (testrichtung .eq. 'y') then
400         gitter = transpose(inp_gitter)
401     else
402         gitter = inp_gitter
403     end if
404
405     ! auf der x-Achse entlangwandern
406     do i=1,i_max
407     ! lokales Minimum in den Schnitten entlang der y-Richtung finden
408         minpunkt = find_local_lgpt_min(gitter(i,:), 2, j_max-1)
409             ! Der Anfang der Suche muss zwei sein, weil die Suchfunktion links
410             ! des ersten Elements zu schauen versucht und sonst kläglich
411             ! versagen würde. -- Am Rand gibt es sowieso keine lokalen Minima.
412
413         ! Je nach dem, wieherum die Schnitte erfolgen, wird ein anderer
414         ! Abstand benötigt.
415         if (testrichtung .eq. 'x') then
416             R = minpunkt%y
417         else
418             R = minpunkt%x
419         end if
420
421         ! Auf Asymptotizität (!) testen.
422         if ((abs(minpunkt%V_L - V_M(moldata, R)) .le. delta_V) &
423            .and. (abs(R - moldata%R_e) .le. delta_R)) then
424            find_asympt_ber = minpunkt
```

```
425             return
426         end if
427 
428     enddo
429 
430     ! Es wurde nichts gefunden.
431     write(*,*) 'ERROR in Funktion find_asympt_ber: Kein asymptotischer', &
432                'Bereich gefunden'
433     stop
434 end function find_asympt_ber
435 
436 
437 ! Funktion lgpt_gt: Testet, ob das Potential des ersten LGPT größer als das
438 ! des zweiten ist.
439 logical function lgpt_gt(lgpt1, lgpt2)
440 implicit none
441 type(LGPT), intent(in) :: lgpt1, lgpt2
442     ! LGPTe, deren Potentialwerte verglichen werden sollen
443 
444     ! Prüfen und zurückgeben
445     if (lgpt1%V_L > lgpt2%V_L) then
446         lgpt_gt = .true.
447     else
448         lgpt_gt = .false.
449     end if
450 end function lgpt_gt
451 
452 
453 ! Funktion lgpt_eq: Testet, ob das Potential des ersten LGPT gleich dem des
454 ! zweiten ist.
455 logical function lgpt_eq(lgpt1, lgpt2)
456 implicit none
457 type(LGPT), intent(in) :: lgpt1, lgpt2
458     ! LGPTe, deren Potentialwerte verglichen werden sollen
459 
460     ! Prüfen und zurückgeben
461     if (lgpt1%V_L == lgpt2%V_L) then
462         lgpt_eq = .true.
463     else
464         lgpt_eq = .false.
465     end if
466 end function lgpt_eq
467 
468 
469 ! Alle anderen Vergleichsoperatoren können mittels der beiden obenstehenden
470 ! definiert werden.
471 
472 ! Funktion lgpt_lt: Testet, ob das Potential des ersten LGPT kleiner als das
473 ! des zweiten ist.
474 logical function lgpt_lt(lgpt1, lgpt2)
475 implicit none
476 type(LGPT), intent(in) :: lgpt1, lgpt2
477     ! LGPTe, deren Potentialwerte verglichen werden sollen
478 
```

```
479     ! Prüfen und zurückgeben
480     if (.not. (lgpt_gt(lgpt1, lgpt2) .or. lgpt_eq(lgpt1, lgpt2))) then
481         lgpt_lt = .true.
482     else
483         lgpt_lt = .false.
484     end if
485 end function lgpt_lt
486
487
488 END module leps_mod
```

Listing 6: traj-mod.f90

```
1  ! Bereitstellung von Funktionen zur Berechnung von Trajektorien
2
3  module traj_mod
4  use leps_mod
5  use rk5_mod
6  use teilchenparam
7  use schwingungen
8  use potentialfkten
9  implicit none
10
11 ! Typ TPKT: Punkt einer Trajektorie
12 type :: TPKT
13     real(8) :: &
14         t,   & ! Zeitpunkt
15         x,   & ! Abszisse
16         y,   & ! Ordinate
17         p_x, & ! Impuls in x-Richtung
18         p_y, & ! Impuls in y-Richtung
19         V_L    ! LEPS-Potential in dem Punkt
20 end type TPKT
21
22
23 CONTAINS
24
25 ! Subroutine calc_traj: Berechnet Trajektorie für angegebene Anfangswerte und
26 ! Reaktion. Die Ergebnisse populieren das Array punkte und werden auf Wunsch
27 ! in eine Datei geschrieben.
28 ! (Diese Subroutine ist analog einer Kombination von next_spkt_mors_ruk4 und
29 ! propagator aus schwingungen.f90.)
30 subroutine calc_traj(punkte, punkte_dim, mol_1, mol_2, startpunkt, y_end, &
31     delta_t, deriv_delta, filename)
32 implicit none
33 integer, intent(in) :: punkte_dim
34     ! ungefähre Anzahl von Punkten der Trajektorie (Array muss vorher
35     ! alloziert werden)
36 type(TPKT), dimension(0:punkte_dim), intent(out) :: punkte
37     ! Array, das mit den Punkten der Trajektorie populiert wird
38 type(Mol2), intent(in) :: mol_1, mol_2
39     ! Daten der zu berechnenden Moleküle
40 type(TPKT), intent(in) :: startpunkt
41     ! Punkt, an dem die Propagation beginnen soll
```

```
42 real(8), intent(in) :: &
43     y_end,    & ! Abszissenwert, bei dem die Propagation abgebrochen werden
44                 ! soll
45                 ! (asymptotischer Bereich)
46     delta_t,  & ! Schrittweite der Propagation
47     deriv_delta ! Schrittweite der Näherung der Ableitung des LEPS-Potentials
48 character(len=*), intent(in), optional :: filename
49     ! Name der Datei, in die die Trajektorienpunkte geschrieben werden sollen
50 character(len=35) :: ERROR = 'ERROR in subroutine call calc_traj:'
51     ! message for the user
52 integer :: i
53     ! Iterator
54 real(8) :: t = 0
55     ! aktuelles Zeitargument
56 real(8), dimension(4) :: &
57     y, &
58     ! Array mit den Werten der abhängigen Variablen
59     yd = (/1d0, 1d0, 1d0, 1d0/)
60
61     ! Array für rk5 mit Werten des Anfangspunktes belegen
62     y = (/ startpunkt%x, startpunkt%y, startpunkt%p_x, startpunkt%p_y /)
63
64     ! startpunkt als ersten Punkt eintragen
65     punkte(0) = startpunkt
66
67     ! i zurücksetzen. Warum behält diese fürchterliche Sprache das i bei,
68     ! wenn man in der Deklaration schreibt = 0?
69     i = 0
70
71     ! Propagieren, solange nicht im asymptotischen Bereich (y-Richtung),
72     do while ((y(2) .le. y_end) &
73               ! nicht wieder über Ausgangspunkt zurück,
74               .and. (y(1) .le. startpunkt%x) &
75               ! Array nicht voll.
76               .and. (i .lt. punkte_dim))
77         ! i inkrementieren
78         i = i + 1
79
80         ! Zeitargument berechnen
81         t = i * delta_t
82
83         ! rk5-Subroutine aufrufen. Achtung: Das Ding modifiziert seine
84         ! Parameter!
85         call rk5(mol_1, mol_2, t, y, yd, delta_t, 4, 1d-8, 1, deriv_delta)
86
87         ! Werte im Trajektorienpunkt speichern
88         punkte(i)%t   = t
89         punkte(i)%x   = y(1)
90         punkte(i)%y   = y(2)
91         punkte(i)%p_x = y(3)
92         punkte(i)%p_y = y(4)
93         punkte(i)%V_L = leps_wei(mol_1, mol_2, y(1), y(2))
94
95 !        write(*,*) punkte(i)
```

```
96      end do
97
98      ! auf Wunsch in Datei schreiben
99      if (present(filename)) then
100         ! Datei öffnen
101         open(unit=55, file=filename, action='write', status='unknown', &
102              form='formatted', access='append')
103             ! In der Hoffnung, dass niemand anderes FD 55 benutzt. -- 42 wäre
104             ! gefährlich.
105
106         ! Daten schreiben
107         call write_tpkt_array(punkte, 1, i, 55)
108             ! i ist das letzte in der Schleife benutzte i.
109
110         ! Datei schließen (wer hätte das gedacht?)
111         close(55)
112     end if
113 end subroutine calc_traj
114
115 ! Funktion write_tpkt_array: Schreibt ein Array von TPKTen in Index start bis
116 ! Index ende zeilenweise an den angegebenen Filedeskriptor.
117 subroutine write_tpkt_array(array, start, ende, deskriptor)
118 implicit none
119 integer, intent(in) :: start, ende, deskriptor
120     ! Start- und Endindex des auszugebenden Stücks Array, Filedeskriptor
121 type(TPKT), intent(in), dimension(:) :: array
122     ! auszugebendes Array
123 integer :: i
124     ! Schleifenvariable
125
126     ! ausgeben
127     do i=start,ende
128         write(deskriptor,*) array(i)
129     enddo
130
131     ! Leerzeilen ans Ende setzen
132     write(deskriptor,*)
133     write(deskriptor,*)
134 end subroutine write_tpkt_array
135
136 ! Funktion gen_traj_start: generiert einen Startpunkt für die Propagation zur
137 ! Trajektorie
138 type(TPKT) function gen_traj_start(moldata, x_0, v, E_0)
139 implicit none
140 type(Mol2) :: moldata
141     ! Daten des zu berechnenden Moleküls
142 real(8) :: &
143     x_0, & ! Abszissenanfangswert (Abstand zwischen B und B)
144     E_0    ! Anfangswert für Gesamtenergie
145 integer :: v
146     ! Schwingungsniveau von B_2
147 type(SZST) :: mol_szst
148     ! Schwingungszustand des gewünschten Moleküls
149
```

```
150     ! Schwingungszustand herausfinden
151     mol_szst = szst_M(moldata, v)
152     !write(*,*) mol_szst
153
154     ! Startparameter eintragen
155     gen_traj_start%t   = 0
156     gen_traj_start%x   = x_0
157     gen_traj_start%y   = mol_szst%R_tr
158         ! rechter Turningpoint soll benutzt werden
159     gen_traj_start%p_x = -sqrt(2 * moldata%my * (E_0 - mol_szst%E_v))
160     gen_traj_start%p_y = 0
161     gen_traj_start%V_L = V_M(moldata, mol_szst%R_tr)
162         ! Im asymptotischen Bereich entspricht das Morsepotential dem
163         ! LEPS-Potential. Außerdem wird der Wert sowieso nicht verwendet.
164 end function gen_traj_start
165
166
167 end module
```

An dieser stelle fehlt ein Listing für die Datei `rk5-mod.f90`, da diese Axel Schilds Implementierung des Runge-Kutta-Verfahrens vierter Ordnung mit Korrekturen fünfter Ordnung (?) enthält.

Listing 7: niv-gen.f90

```
1  ! gibt die Schwingungszustandsliniendaten für die Zusatzaufgabe für Gnuplot
2  ! aus
3
4  program niv_gen
5  use schwingungen
6  use teilchenparam
7  use utils
8  implicit none
9
10 ! DECLARATIONS !!!!!!!!!!!!!!!
11 type(Mol2), parameter :: &
12     mol_1 = MCl, &
13     mol_2 = Cl2
14         ! zu verwendende Moleküle
15 real(8), parameter, dimension(4) :: &
16     t_1_start = (/ 42, 40, 67, 56 /), &
17     t_1_end   = (/ 45, 43, 70, 59 /), &
18     t_2_start = (/ 15, 17, 25, 30 /), &
19     t_2_end   = (/ 18, 20, 28, 33 /)
20         ! Da die Energie des Moleküls in Abhängigkeit von der Zeit aufgeführt
21         ! wird, müssen die Anfangs- und Endzeitwerte für die waagerechten
22         ! Striche der Schwingungsniveaus per Hand angepasst werden.
23 character(len=15), parameter, dimension(4) :: &
24     niv_file = (/ 'nivs-v0-v11.dat', 'nivs-v0-v06.dat', 'nivs-v5-v13.dat', &
25                   'nivs-v5-v02.dat' /)
26         ! Datei, in die die Daten geschrieben werden sollen
27 integer :: &
28     v, &
29         ! Iterator -- aktueller Schwingungszustand
30     i, &
```

```
31          ! Iterator für die verschiedenen Niveau-Dateien
32      desc
33          ! Filedeskriptor für Ausgabedateien
34  type(SZST) :: cur_szst
35          ! aktueller Schwingungszustand
36
37  ! in die verschiedenen Niveau-Dateien schreiben
38  do i = 1, 4
39      ! Ausgabedatei öffnen
40      open(unit=desc, file=niv_file(i), action='write', status='replace', &
41          form='formatted')
42
43      ! Berechnung für mol_1
44      ! Gleichgewichtsradius als unterste Linie
45      call write_gpl_line_sg(t_1_start(i), t_1_end(i), -mol_1%D, desc)
46
47      ! Schwingungszustände
48      do v = 0, calc_v_max(mol_1)
49          ! Schwingungszustand berechnen
50          cur_szst = szst_M(mol_1, v)
51
52          ! als Linie für Gnuplot schreiben
53          call write_gpl_line_sg(t_1_start(i), t_1_end(i), cur_szst%E_v, desc)
54      end do
55
56      ! Berechnung für mol_2
57      ! Gleichgewichtsradius als unterste Linie
58      call write_gpl_line_sg(t_2_start(i), t_2_end(i), -mol_2%D, desc)
59
60      ! Schwingungszustände
61      do v = 0, calc_v_max(mol_2)
62          ! Schwingungszustand berechnen
63          cur_szst = szst_M(mol_2, v)
64
65          ! als Linie für Gnuplot schreiben
66          call write_gpl_line_sg(t_2_start(i), t_2_end(i), cur_szst%E_v, desc)
67      end do
68
69      ! Ausgabedatei schließen
70      close(13)
71  end do
72
73  end program niv_gen
```

Listing 8: konst-umr.f90

```
1   ! Bereitstellung von Naturkonstanten und Umrechnungsfaktoren für die
2   ! Manz'schen Einheiten
3
4   module konst_umr
5   implicit none
6   save
7
8   !! DECLARATIONS !!!!!!!!!!!!!!!
```

```
real(8), parameter :: &
! Naturkonstanten
    hquer   = 0.10546,      & ! h
    Pi      = acos(-1.),    & ! Pi
    N_A     = 6.02214129d23,& ! Avogadro-Konstante in 1/mol, Quelle:
                ! http://physics.nist.gov/cgi-bin/cuu/Value?na, 2011-10-11
! Umrechnungsfaktoren
    eV      = 1.602177,     & ! 1 eV = 1.602177 * 10^-19 J
    u       = 1.66054,      & ! 1 u  = 1.66054 * 10^-27 kg
    kJjmol  = 1d22 / N_A,   & ! 1 kJ/mol = 10^22 * 10^-19 J * N_A (wir rechnen
                              ! mit einzelnen Teilchen°
    rad     = Pi / 180!     & ! 1 rad  = 2 * Pi / 360°


end module konst_umr
```

Listing 9: potentialfkten.f90

```
! Bereitstellung der Funktionen für Morsepotential und harmonisches Potential

module potentialfkten
use teilchenparam
implicit none
save

!! DECLARATIONS !!!!!!!!!!!!!!!

CONTAINS

!! Funktion V_M: berechnet das Morsepotential eines Moleküls beim Radius R
real(8) function V_M(moldata, R)
implicit none
type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
real(8)    :: R       ! gefragter Radius

V_M = -moldata%D + moldata%D * (exp(-moldata%beta * (R-moldata%R_e)) - 1)**2
return
end function

!! Funktion V_M_: berechnet die erste Ableitung des Morsepotentials eines
!! Moleküls beim Radius R
real(8) function V_M_(moldata, R)
implicit none
type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
real(8) :: &
    R,  & ! gefragter Radius
    b     ! Dummyariable

    b    = -moldata%beta * (R - moldata%R_e)
    V_M_ = 2 * moldata%D * moldata%beta * (exp(b) - exp(2 * b))
end function V_M_

!! Funktion V_M__: berechnet die zweite Ableitung des Morsepotentials eines
!! Moleküls beim Radius R
```

```
37 real(8) function V_M__(moldata, R)
38 implicit none
39 type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
40 real(8) :: &
41     R,  & ! gefragter Radius
42     b     ! Dummyariable
43
44     b    = -moldata%beta * (R - moldata%R_e)
45     V_M__ = 2 * moldata%D * moldata%beta &
46           * (2 * moldata%beta * exp(2 * b) - exp(b))
47 end function V_M__
48
49 !! Funktion V_h: berechnet das harmonische Potential eines Moleküls beim
50 !! Radius R
51 real(8) function V_h(moldata, R)
52 implicit none
53 type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
54 real(8)    :: R       ! gefragter Radius
55
56 V_h = -moldata%D + moldata%D * moldata%beta**2 * (R - moldata%R_e)**2
57 return
58 end function
59
60
61 end module potentialfkten
```

Listing 10: teilchenparam.f90

```
1  ! Bereitstellung diverser Werte (Eigenschaften) der verwendeten Teilchen
2
3  module teilchenparam
4  use konst_umr
5  implicit none
6
7  !! DECLARATIONS !!!!!!!!!!!!!!!
8  !! Typ Atom: enthält alle Parameter eines Atoms
9  type :: Atom
10     real(8) :: &
11         m   ! Masse
12 end type Atom
13
14 ! Typ Mol2: enthält alle Parameter eines zweiatomigen Moleküls
15 type :: Mol2
16     real(8) :: &
17        M,      & ! Gesamtmasse
18        my,     & ! reduzierte Masse
19        D,      & ! Tiefe des Potentialtopfs
20        R_e,    & ! Gleichgewichtsradius des Oszillators
21        beta,   & ! Morsekonstante
22        delta!  & ! Sato-Parameter
23 !       k         ! Kraftkonstante
24 end type Mol2
25
26 ! alle Atome deklarieren
```

```
27 type(Atom), parameter :: &
28     H  = Atom(1.0079 * u),  & ! Wasserstoff
29     F  = Atom(19.000 * u),  & ! Fluor
30     Cl = Atom(34.969 * u),  & ! Chlor
31     M  = Atom(4.116  * u)!  & ! Myonisches Helium/4.1-H
32
33 ! alle zweiatomigen Moleküle deklarieren
34 type(Mol2), parameter :: &
35     HF = Mol2( & ! Fluorwasserstoff
36             H%m + F%m,               &
37             H%m * F%m / (H%m + F%m), &
38             6.12 * eV,               &
39             0.9171,                  &
40             2.219,                   &
41             0d0                      &
42         ),    &
43     F2 = Mol2( &      ! Fluormolekül
44             F%m + F%m,               &
45             F%m * F%m / (F%m + F%m), &
46             1.63 * eV,               &
47             1.418,                   &
48             2.920,                   &
49             -0.35                    &
50         ),    &
51     Cl2 = Mol2( & ! Chlormolekül
52             Cl%m + Cl%m,                 &
53             Cl%m * Cl%m / (Cl%m + Cl%m), &
54             242.848 * kJjmol,            &
55             1.9998,                      &
56             2.0080,                      &
57             -0.113                       &
58         ),    &
59     MCl = Mol2( & ! Chlorwasserstoff mit 4.1-H
60             M%m + Cl%m,                &
61             M%m * Cl%m / (M%m + Cl%m), &
62             446.332 * kJjmol,          &
63             1.2732,                    &
64             1.8693,                    &
65             0.067                      &
66         )
67
68 CONTAINS
69
70 !! Funktion red_mass: reduzierte Masse eines zweiatomigen Moleküls
71 !! aus der Masse der einzelnen Atome berechnen
72 real(8) function red_mass(m1, m2)
73 implicit none
74 real(8) :: m1, m2   ! Massen der beiden Atome
75     red_mass = m1 * m2 / (m1 + m2)
76 end function red_mass
77
78 !! Funktion kraftkonst: Kraftkonstante eines zweiatomigen Moleküls berechnen
79 real(8) function kraftkonst(moldata)
80 implicit none
```

```
81  type(Mol2) :: moldata   ! Daten des zu berechnenden Moleküls
82      kraftkonst = 2 * moldata%D * moldata%beta**2
83  end function kraftkonst
84
85  !! Funktion kreisfreq: Kreisfrequenz eines zweiatomigen Moleküls berechnen
86  real(8) function kreisfreq(moldata)
87  implicit none
88  type(Mol2) :: moldata   ! Daten des zu berechnenden Moleküls
89      kreisfreq = sqrt(2*moldata%D / moldata%my) * moldata%beta
90  end function kreisfreq
91
92  !! Funktion periodend: Periodendauer eines zweiatomigen Moleküls berechnen
93  real(8) function periodend(moldata)
94  implicit none
95  type(Mol2) :: moldata   ! Daten des zu berechnenden Moleküls
96      periodend = 2 * Pi / kreisfreq(moldata)
97  end function periodend
98
99  ! Funktion calc_v_max: Gibt die Nummer des höchsten Schwingungszustandes
100 ! (Morseoszillator) eines Moleküls zurück.
101 integer function calc_v_max(moldata)
102 implicit none
103 type(Mol2) :: moldata ! Daten des zu berechnenden Moleküls
104
105     calc_v_max = floor(2*moldata%D / (hquer * kreisfreq(moldata)) - 0.5)
106 end function calc_v_max
107
108 end module teilchenparam
```

Listing 11: utils.f90

```
1   ! Bereitstellung verschiedener Hilfsfunktionen
2
3   module utils
4   implicit none
5
6   !! DECLARATIONS !!!!!!!!!!!!!!!
7
8   CONTAINS
9
10  !! Funktion schrittweite: berechnet die Schrittweite für eine Schleife bei
11  !! gegebenen Anfangs-, Endwert und Iterationszahl
12  real(8) function schrittweite(start, ende, schritte)
13  implicit none
14  real(8) :: &
15      start,   & ! Anfangswert der Schleife
16      ende!,    & ! Endwert der Schleife
17  integer :: schritte   ! Anzahl der Interationen in der Schleife
18
19      schrittweite = (ende - start) / real(schritte)
20  end function schrittweite
21
22  !! Funktion iterationen: berechnet die Anzahl Iterationen für eine Schleife
23  !! bei gegebenen Anfangs-, Endwert und Schrittweite
```

```
24 integer function iterationen(start, ende, delta_x)
25 implicit none
26 real(8) :: &
27     start,  & ! Anfangswert der Schleife
28     ende,   & ! Endwert der Schleife
29     delta_x   ! Schrittweite
30
31     iterationen = ceiling( (ende-start) / delta_x )
32 end function iterationen
33
34 ! Subroutine delete_file: löscht eine Datei
35 subroutine delete_file(filename)
36 implicit none
37 character(len=*) :: filename
38     ! Name der zu löschenden Datei
39
40     ! Datei öffnen...
41     open  (unit=83, file=filename, status="unknown")
42     ! und beim Schließen löschen.
43     close (unit=83, status="delete")
44 end subroutine delete_file
45
46 ! Subroutine write_gpl_line_sg: Schreibt einen Datensatz für eine Strecke in
47 ! Gnuplot an eine angegebene Datei
48 subroutine write_gpl_line_sg(x_start, x_end, y, deskriptor)
49 implicit none
50 real(8) :: &
51     x_start, x_end, &
52     ! Anfangs- und Endabszisse der Strecke
53     y
54     ! Ordinate der Strecke
55 integer :: deskriptor
56     ! Dateideskriptor, an den geschrieben werden soll
57
58     ! Anfangs- und Endpunkt schreiben
59     write(deskriptor,*) x_start, y
60     write(deskriptor,*) x_end,   y
61
62     ! Datensatz mit Leerzeile beenden
63     write(deskriptor,*)
64 end subroutine write_gpl_line_sg
65
66 end module utils
```